WebSphere® Application Server for z/OS, Version 7.0

Developing WebSphere applications

SA23-2257-00

Note

Before using this information, be sure to read the general information under “Notices” on page 1703.

Compilation date: September 23, 2008

Contents

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an e-mail form appears.
  3. Fill out the e-mail form as instructed, and click on **Submit feedback** .
- To send comments on PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com** or fax them to 919-254-5250.

  Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Changes to serve you more quickly

Print sections directly from the information center navigation

PDF books are provided as a convenience format for easy printing, reading, and offline use. The information center is the official delivery format for IBM WebSphere Application Server documentation. If you use the PDF books primarily for convenient printing, it is now easier to print various parts of the information center as needed, quickly and directly from the information center navigation tree.

To print a section of the information center navigation:

1. Hover your cursor over an entry in the information center navigation until the **Open Quick Menu** icon is displayed beside the entry.
2. Right-click the icon to display a menu for printing or searching your selected section of the navigation tree.
3. If you select **Print this topic and subtopics** from the menu, the selected section is launched in a separate browser window as one HTML file. The HTML file includes each of the topics in the section, with a table of contents at the top.
4. Print the HTML file.

For performance reasons, the number of topics you can print at one time is limited. You are notified if your selection contains too many topics. If the current limit is too restrictive, use the feedback link to suggest a preferable limit. The feedback link is available at the end of most information center pages.

Under construction!

The Information Development Team for IBM WebSphere Application Server is changing its PDF book delivery strategy to respond better to user needs. The intention is to deliver the content to you in PDF format more frequently. During a temporary transition phase, you might experience broken links. During the transition phase, expect the following link behavior:

- Links to Web addresses beginning with http:// work
- Links that refer to specific page numbers within the same PDF book work
- The remaining links will *not* work. You receive an error message when you click them

Thanks for your patience, in the short term, to facilitate the transition to more frequent PDF book updates.

Chapter 1. Web applications

Tuning URL invocation cache

The URL invocation cache holds information for mapping request URLs to servlet resources. A cache of the requested size is created for each worker thread that is available to process a request. The default size of the invocation cache is 50. If more than 50 unique URLs are actively being used (each JavaServer Page is a unique URL), you should increase the size of the invocation cache.

Before you begin

A larger cache uses more of the Java™ heap, so you might also need to increase the maximum Java heap size. For example, if each cache entry requires 2KB, maximum thread size is set to 25, and the URL invocation cache size is 100; then 5MB of Java heap are required.

The invocation cache is now Web container based instead of thread-based, and shared for all Web container threads.

About this task

To change the size of the invocation cache:

1. In the administrative console, click **Servers** → **Server Types** → **WebSphere application servers** and select the application server that you are tuning.
2. Click **Java and Process Management**.
3. Click **Process Definition** under Additional Properties, and then select either **control** or **servant** depending on whether you want this property defined in the control or the servant.
4. Click **Java Virtual Machine** under Additional Properties.
5. Click **Custom Properties** under Additional Properties.
6. Specify **invocationCacheSize** in the Name field and the size of the cache in the Value field. The default size for the invocation cache is 500 entries. Since the invocation cache is no longer thread-based, the invocation cache size specified by the user is multiplied by ten to provide similar function from previous releases. For example, if you specify an invocation cache size of 50, the Web container will create a cache size of 500.
7. Click **Apply** and then **Save** to save your changes.
8. Stop and restart the application server.

Results

The new cache size is used for the URL invocation cache.

Developing servlet applications using asynchronous request dispatcher

Web modules can dispatch requests concurrently on separate threads. Requests can be dispatched by the server or client.

Before you begin

For additional information about the AsyncRequestDispatcherConfig and the AsyncRequestDispatcher interfaces, review the com.ibm.websphere.webcontainer.async package in the application programming interfaces (API) documentation. The generated API documentation is available in the information center table of contents from the path Reference > APIs - Application Programming Interfaces.

Review the asynchronous request dispatcher application (ARD) design considerations topic before completing the following steps.

About this task

Concurrent dispatching can improve servlet response time. If operations are dependant on each other, do not enable asynchronous request dispatching, therefore, select Disabled. Concurrent dispatching might result in errors when operations are dependant. Select Server side to enable the server to aggregate requests dispatched concurrently. Select Client side to enable the client to aggregate requests dispatched concurrently.

1. Logically separate resource intensive operations.
2. Develop servlets that use an asynchronous request dispatcher to include these operations.
3. Enable asynchronous request dispatching on an application server.
4. Deploy the application in an application server that has asynchronous request dispatching enabled.
5. Select an aggregation type for the application that needs ARD.
6. Optional: Configure the AsyncRequestDispatcherWorkManager work manager that is used for the request dispatch threads.
7. Restart the application server.

What to do next

Restart the modified applications if already installed or start newly installed applications to enable ARD on each application.

Asynchronous request dispatcher

Asynchronous request dispatcher (ARD) can improve Servlet response time when slow operations can be logically separated and performed concurrently with other operations required to complete the response. ARD enables Java™ servlet programmers to perform standard javax.servlet.RequestDispatcher include calls for the same request concurrently on separate threads. These javax.servlet.RequestDispatcher include calls are completed sequentially on the same thread. ARD is also useful in low CPU, long wait situations like waiting for a database connection.

If there are large CPU or memory requirements, ARD alone does not alleviate those issues. However, in combination with the remote request dispatcher, operations driven by one servlet request that can be performed concurrently on multiple application servers, alleviating resource demand on a single server and decreasing the risk of a system down situation.

Servlets, portlets, and JavaServer Pages (JSP) files can all utilize ARD. This functionality is an extension beyond the requirements of the Java Servlet Specification , which only describes synchronous request dispatching. ARD requires a new channel, called the ARD channel, between the HTTP and Web container channels to form a new channel chain. These new chains correspond only to the existing default host chains and reuse the same ports.

Each include can write output to the client and because ordering is important for valid results, there must be some aggregation of the data written. Typically, a servlet writes data to a buffer and once full, it is flushed to client. For server-side aggregation, the ARD channel cannot flush until any includes that had placeholders written to the current buffer are finished.

Client-side aggregation of the asynchronous include is also supported. Web 2.0 programmers often use Asynchronous JavaScript™ and XML (Ajax) in the Web browser of the client to dynamically retrieve and aggregate remote resources. Unfortunately, this puts the burden on the programmer to aggregate the contents and learn new technologies. Client-side aggregation automatically adds the necessary JavaScript

to dynamically update the page. For non-JavaScript clients, you can switch ARD to server-side aggregation, which gives equivalent results. You can deny non-JavaScript clients when using client-side aggregation.

ARD uses the Web container APIs to plug in unique request dispatching logic. It interacts with WCCM to read in configuration information for enablement status per enterprise application as well as a global appserver setting. You can use the administrative console and wsAdmin to enable or disable ARD. Servlets, portlets, and JSP files can all utilize ARD.

Related tasks

"Developing servlet applications using asynchronous request dispatcher" on page 1
Web modules can dispatch requests concurrently on separate threads. Requests can be dispatched by the server or client.

Related reference

"Asynchronous request dispatcher application design considerations"
Asynchronous request dispatcher (ARD) is not a one-size-fits-all solution to servlet programming. You must evaluate the needs of your application and the caveats of using ARD. Switching all includes to start asynchronously is not the solution for every scenario, but when used wisely, ARD can increase response time. This article contains important details about the ARD implementation and issues to consider when you design an application that leverages ARD.

Related information

Asynchronous request dispatcher application design considerations

Asynchronous request dispatcher (ARD) is not a one-size-fits-all solution to servlet programming. You must evaluate the needs of your application and the caveats of using ARD. Switching all includes to start asynchronously is not the solution for every scenario, but when used wisely, ARD can increase response time. This article contains important details about the ARD implementation and issues to consider when you design an application that leverages ARD.

Asynchronous request dispatcher client-side implementation

- JavaScript is dynamically written to the response output.
- This JavaScript results in Ajax requests back to a server-side results provider.
- Because of the Asynchronous Input/Output (AIO) features of the channel, the Ajax request does not tie up a thread and instead is notified for completion through an include callback.
- The client only makes one request at a time for the asynchronous includes because of browser limitations in the number of connections.
- Original connection has to be valid for the lifetime of the includes. It cannot be reused for the Ajax requests.
- Comment nodes, such as following,

  ```
  <!--uniquePlaceholderID--><!--1-->
  ```

  are placed in the browser object model since comment nodes have no effect on the page layout.
- Whenever a complete fragment exists, a response can be sent to the client and the comment node with the same ID is replaced. Requests are made until all the fragments are retrieved.
- Verify applications on all supported browsers when using client-side aggregation. Object oriented JavaScript principles are used so that applications only need avoid using the method name getDynamicDataIBMARD. Any previously specified window.onload is started before the ARD onload method.

Asynchronous request dispatcher channel results service

Requests for include data from the asynchronous JavaScript code are sent to known Uniform Resource Identifiers, URIs also known as URLs, that the ARD channel can intercept to prevent traveling through Web container request handling. These URIs are unique for the each server restart.

For example, `/IBMARD01234567/asyncInclude.js` is the URI for the JavaScript that forces the retrieval of the results, and `/IBMARD01234567/IBMARDQueryStringEntries?=12000` is used to retrieve the results for the entry with ID 12000.

To prevent unauthorized results access, unique IDs are generated for the service URI and for the ARD entries. A common ID generator is shared among the session and ARD, so uniqueness is configurable through session configuration. Session IDs are considered secure, but they are not as secure as using a Lightweight Third-Party Authentication (LTPA) token.

Custom client-side aggregation

If you want to perform your own client-side aggregation, the isUseDefaultJavascript method must return as false. The isUseDefaultJavascript method is part of the AsyncRequestDispatcherConfig method, which is set on the AsyncRequestDispatcher or for the AsyncRequestDispatcherConfigImpl.getRef method. The AsyncRequestDispatcherConfigImpl.getRef method is the global configuration object. You might want to perform your own client-side aggregation if the back button functionality is problematic. You must remove the results from the generic results service to prevent memory leaks, so that multiple requests with the same response results through an XMLHttpRequest fail. To facilitate proper location of position, placeholders are still written in the code as

```
<!--uniquePlaceholderID--><!--x-->
```

where x is the order of the includes. The endpoint to retrieve results are retrieved from the request attribute com.ibm.websphere.webcontainer.ard.endpointURI.

When making a request to the endpoint, ARD sends as many response fragments as possible when the request is made. Therefore, the client needs to re-request if all fragments are not initially returned. Trying to display the results directly in a browser without using an XMLHttpRequest can result in errors related to non well-formed XML. The response data is returned in the following format with a content type of text/xml:

```
<div id="2"><BR>Servlet 3--dispatcher3 requesting Servlet3 to sleep for 0 seconds at: 1187967704265
<BR> Servlet 3--Okay, all done!  This should print pop up: third at: 1187967704281 </div>
```

For additional information about the AsyncRequestDispatcherConfig and the AsyncRequestDispatcher interfaces, review the com.ibm.websphere.webcontainer.async package in the application programming interfaces (API) documentation. The generated API documentation is available in the information center table of contents from the path **Reference** → **APIs - Application Programming Interfaces**.

Server-side aggregation

Like client-side aggregation, server-side aggregation uses the ARD channel as a results service. The ARD channel knows which asynchronous includes have occurred for certain set of buffers. Those buffers can then be searched for an include placeholder. Because of the issues of JSP buffering, the placeholder for the include might not be in the searched buffers. If this occurs, the next set of buffers must also look for any include placeholders missed in the previous set. ARD attempts to iteratively aggregate as includes return so that response content can be sent to the client as soon as possible.

Asynchronous beans

An AsynchBeans work manager is used to start the includes. If the number of currently requested includes is greater than the work manager maximum thread pool size and this size is not growable, it starts the work on the current thread and skips the placeholder write. Utilizing AsynchBeans suppports propagation

of the J2EE context of the original thread including work area, internationalization, application profile, z/OS® operation system work load management, security, transaction, and connection context.

Timer

A single timer is used for ARD and timer tasks are created for all the timeout types of ARD requests. Tasks registered with the timer are not guaranteed to run at the exact time specified because the timer runs on a single thread, therefore one timeout might have to wait for the other timeout actions to complete. The timer is used as a last resort.

Remote request dispatcher

Optionally, ARD can be used in concert with the remote request dispatcher. The remote request dispatcher was introduced in WebSphere® Application Server Network Deployment 6.1. The remote request dispatcher runs the include on a different application server in a core group by serializing the request context into a SOAP message and using Web services to call the remote server. This is useful when the expense of creating and sending a SOAP message through Web services is outweighed by issuing the request locally. For more information, see this developerWorks article.

Exceptions

In the case of an exception in an included servlet, the Web container goes through the error page definitions mapped to exception types. So an error page defined in the deployment descriptor shows up as a portion of the aggregated page. Insert logic into the error page itself if behavior is different for an include. Because the include runs asynchronously, there is no guarantee that the top level servlet is still in service, therefore the exception is not propagated back from an asynchronous include like a normal include. Other includes finish so that partial pages can be displayed.

If the ARD work manager runs out of worker threads, the include is processed like a synchronous include. This is the default setting, but the work manager can also grow such that it does not result in this condition. This change in processing is invisible to the user during processing but is noted once in the system logs as a warning message and the rest of the time in the trace logs when enabled. Other states that can trigger the include to occur synchronously are reaching the maximum percentage of expired requests over a time interval and reaching the maximum size of the results store.

There are cases where exceptions happen outside of the scope of normal error page handling. For example, work can be rejected by the work manager. A timer can expire waiting for an include response to return. The ARD channel, acting as a generic service to retrieve the results, might receive an ID that is not valid. In these cases, there is no path to the error page handling because the context is missing, such as ServletRequest, ServletResponse, and ServletContext, for the request to work. To mitigate these issues, you can use the AsyncRequestDispatcherConfig interface to provide custom error messages. Defaults are provided and internationalized as needed.

Exceptions can also occur outside the scope of the request the custom configuration was set on, such as on the subsequent client-side XMLHttpRequests. In this case, the global configuration must be altered. This can be retrieved through com.ibm.wsspi.ard.AsyncRequestDispatcherConfigImpl.getRef().

Include start
: The work manager provides a timeout for how long to wait for an include to start. Since this typically happens immediately, there is not a programmatic way to enable this. However, this is configurable in the work manager settings. By default, you will not encounter this because of the maximum thread check before scheduling the work. Work can be retried if setRetriable(true) is called on the in use AsyncRequestDispatcherConfig.

Include finish
: The initiated timeout starts after the work is accepted. It can be configured through the console or programmatically through the AsyncRequestDispatcherConfig.setExecutionTimeoutOverride

method; The default value is 60000 ms, or one minute. In place of the include results, the message from the AsyncRequestDispatcherConfig.setExecutionTimeoutMessage is sent. If this intiated timeout is reached, but the actual include results are ready when the data can be flushed, preference is given to the actual results. Also, this does not apply to insertFragmentBlocking calls which always wait until the include is completed.

Expiration of results

Since the client-side has to hold the results in a service to send for the Ajax request, we want a way to expire the results if the client goes down and never retrieves the entry. The default of a minute is sufficient for a typical request because the Ajax request would come in immediately after sending the response. The timer can be configured programmatically via the setExpirationTimeoutOverride method the of AsyncRequestDispatcherConfig. The message from the getOutputRetrievalFailureMessage method of AsyncRequestDispatcherConfig is displayed when someone tries to access an entry that has expired and been removed from cache. This message is the same message that is sent to someone requesting a result with an ID that never existed.

Includes versus fragments

Consider which operations can be done asynchronously and when they can start. Ideally, all the includes are completed when the getFragment calls are made at the beginning of the request so that the includes can have more time to complete, and upon inserting the fragments, there would be less extra buffering and aggregating if they have completed. However, simply calling an asynchronous include is easier because it follows the same pattern as a normal request dispatcher include.

Web container

ServletContext

When doing cross-context includes, the context that is a target of the include must also have ARD enabled because the Web application must have been initialized for ARD for its servlet context to have valid methods to retrieve an AsyncRequestDispatcher. The aggregation type is determined by the original context's configuration because you cannot mix aggregation types.

ServletRequest

You must clone the request for each include. Otherwise, conflicts between threads might occur. Because applications can wrap the default request objects, your wrappers must implement the com.ibm.wsspi.webcontainer.servlet.IServletRequest interface, which has one method, the public Object clone method, which creates the CloneNotSupportedException.

Unwrapping occurs until a request wrapper that implements this interface is found. Non-implementing wrappers are lost; however, a servlet filter configured for the include can rewrap the response.

Changes made to the ServletRequest are not propagated back to the top level servlet unless transferState on the AsyncRequestDispatcherConfig is enabled and insertFragmentBlocking is called.

ServletResponse

A wrapped response extending com.ibm.websphere.servlet.response.StoredResponse is created by ARD and sent to the includes because the response output must be retrievable beyond the lifecycle of the original response.

Internal headers set in asynchronous includes are not supported due to lifecycle restrictions unless transferState on the AsyncRequestDispatcher config is enabled and insertFragmentBlocking is called. Normal headers are not supported in a synchronous include as specified by the servlet specification.

Include filters can rewrap the new response and must flush upon completion.

ServletInputStream
An application reading parameters using getParameter is not problematic.Parsing of parameters is forced before the first asynchronous include to prevent concurrent access to the input stream.

HttpSession
Initial getSession calls that result in a Set-Cookie header must be called from the top level servlet because it is unpredictable when the includes are started and if the headers have already been flushed. The exception is when transferState on the AsyncRequestDispatcherConfig is enabled and an insertFragmentBlocking is called. This normally creates an exception when you add the header.

Filters If there is a filter for an include, the filter is issued on the asynchronous thread.

Nested asynchronous includes
Nested asynchronous includes are not supported because they complicate aggregation. However, an asynchronous include can have nested synchronous includes. Any attempt to perform a nested asynchronous include reverts back to a synchronous include.

Transactions

Every asynchronous bean method is called using its own transaction, much like container-managed transactions in typical enterprise beans. The runtime starts a local transaction before invoking the method. The asynchronous bean method can start its own global transaction if this transaction is possible for the calling J2EE component.

If the asynchronous bean method creates an exception, any local transactions are rolled back. If the method returns normally, any incomplete local transactions are completed according to the unresolved action policy configured for the bean. If the asynchronous bean method starts its own global transaction and does not commit this global transaction, the transaction is rolled back when the method returns.

Connection management

An asynchronous bean method can use the connections that its creating servlet obtained using java:comp resource references. However, the bean method must access those connections using a get, use or close pattern. There is no connection caching between method calls on an asynchronous bean. The connection factories or data sources can be cached, but the connections must be retrieved on every method call, used, and then closed. While the asynchronous bean method can look up connection factories using a global Java Naming and Directory Interface (JNDI) name, this is not recommended for the following reasons:

- The JNDI name is hard coded in the application, for example, as a property or string literal.
- The connection factories are not shared because there is no way to specify a sharing scope.

Evaluate high load scenarios because asynchronous includes might increase the number of threads waiting on the connection.

Performance

Because includes are completed asynchronously, the total performance data for a request must take into consideration the performance of the asynchronous includes. The total time of the request could previously be understood by the time for the top level servlet to complete, but now that servlet is exiting before the includes are completed. The top level servlet still accounts for much of the additional setup time required for each include.

Therefore, a new ARD performance metric was added to the Performance Monitoring Infrastructure to measure the time for a complete request through the ARD channel. The granularity of these metrics is at the request URI level.

Since ARD is an optional feature that has to be enabled, no performance decline is seen when not utilizing ARD. However, non-ARD applications that reside on an ARD-enabled application server would suffer from the extra layer of the ARDChannel. The channel layer does not know to which application it is going so it is either on or off for all applications in a channel chain. These are defined per virtual host.

Security

Security is not invoked on synchronous include dispatches according to the servlet specification. However, security context is passed along through AsynchBeans to support programmatic usage of the isUserInRole and getUserPrincipal methods on the ServletRequest. This security context can also be propagated across to a remote request dispatch utilizing Web services security.

Related concepts

“Asynchronous request dispatcher” on page 2
Asynchronous request dispatcher (ARD) can improve Servlet response time when slow operations can be logically separated and performed concurrently with other operations required to complete the response. ARD enables Java™ servlet programmers to perform standard javax.servlet.RequestDispatcher include calls for the same request concurrently on separate threads. These javax.servlet.RequestDispatcher include calls are completed sequentially on the same thread. ARD is also useful in low CPU, long wait situations like waiting for a database connection.

Related tasks

“Developing servlet applications using asynchronous request dispatcher” on page 1
Web modules can dispatch requests concurrently on separate threads. Requests can be dispatched by the server or client.

Asynchronous request dispatching settings

Use this page to enable the asynchronous request dispatcher (ARD), which enables servlets and JSP pages to make standard include calls concurrently on separate threads.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Web Container Settings** → **Web container** → **Asynchronous request dispatching**.

Additionally, the initiation and the insertion of the include contents can be separated so that the include has more time to execute before it needs to be written to the response. ARD requires aggregation of the include contents with the original response. The application server can aggregate the contents in memory or the client browser can aggregate the contents through AJAX. Aggregation type is configurable at the application level.

Related concepts

“Asynchronous request dispatcher” on page 2
Asynchronous request dispatcher (ARD) can improve Servlet response time when slow operations can be logically separated and performed concurrently with other operations required to complete the response. ARD enables Java™ servlet programmers to perform standard javax.servlet.RequestDispatcher include calls for the same request concurrently on separate threads. These javax.servlet.RequestDispatcher include calls are completed sequentially on the same thread. ARD is also useful in low CPU, long wait situations like waiting for a database connection.

Related tasks

“Developing servlet applications using asynchronous request dispatcher” on page 1
Web modules can dispatch requests concurrently on separate threads. Requests can be dispatched by the server or client.

Allow Asynchronous Request Dispatching

Enables applications installed on this server to use asynchronous request dispatching.

Asynchronous include timeout
Specifies the default timeout in milliseconds to complete asynchronous includes.

Data type	Integer
Units	Milliseconds
Default	60000
Range	

Maximum expired requests per minute
Specifies the maximum percentage of expired response versus total response in one minute before switching to synchronous requests.

Data type	Integer
Units	Percentage
Default	15
Range	

Maximum memory size of results store
Specifies the maximum size of store for client side requests.

Data type	Integer
Units	Megabytes
Default	100
Range	

Task overview: Managing HTTP sessions

IBM® WebSphere Application Server provides a service for managing HTTP sessions, Session Manager. The key activities for session management are summarized in this topic.

About this task

Before you begin these steps, make sure you are familiar with the programming model for accessing HTTP session support in the applications following the Servlet 2.5 API.

1. Plan your approach to session management, which could include session tracking, session recovery, and session clustering.
2. Create or modify your own applications to use session support to maintain sessions on behalf of Web applications.
3. Assemble your application.
4. Deploy your application.
5. Ensure the administrator appropriately configures session management in the administrative domain.
6. Adjust configuration settings and perform other tuning activities for optimal use of sessions in your environment.

Sessions

A session is a series of requests to a servlet, originating from the same user at the same browser.

Sessions allow applications running in a Web container to keep track of individual users.

For example, a servlet might use sessions to provide ″shopping carts″ to online shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she wants to purchase from the Web

site. It is important that the servlet be able to associate incoming requests with particular shoppers. Otherwise, the servlet might mistakenly add Shopper_1's choices to the cart of Shopper_2.

A servlet distinguishes users by their unique session IDs. The session ID arrives with each request. If the user's browser is cookie-enabled, the session ID is stored as a cookie. As an alternative, the session ID can be conveyed to the servlet by URL rewriting, in which the session ID is appended to the URL of the servlet or JavaServer Pages (JSP) file from which the user is making requests. For requests over HTTPS or Secure Sockets Layer (SSL), another alternative is to use SSL information to identify the session. Session tracking using the SSL ID is deprecated in WebSphere Application Server version 7.0. You can configure session tracking to use cookies or modify the application to use URL rewriting.

HTTP session migration

There are no programmatic changes required to migrate from version 5.x to version 6.x. This article describes features that are available after migration.

Migration from Version 5.x

Note: In Version 5 and later, default write frequency mode is TIME_BASED_WRITES, which is different from Version 4.0.x default mode of END_OF_SERVICE.

Session security support

You can integrate HTTP sessions and security in WebSphere Application Server. When security integration is enabled in the session management facility and a session is accessed in a protected resource, you can access that session only in protected resources from then on.

You cannot mix secured and unsecured resources accessing sessions when security integration is turned on. Security integration in the session management facility is not supported in form-based login with SWAM.

Note: SWAM is deprecated in WebSphere Application Server Version 7.0 and will be removed in a future release.

Security integration rules for HTTP sessions

Only authenticated users can access sessions created in secured pages and are created under the identity of the authenticated user. Only this authenticated user can access these sessions in other secured pages. To protect these sessions from unauthorized users, you cannot access them from an unsecured page.

Programmatic details and scenarios

WebSphere Application Server maintains the security of individual sessions.

An identity or user name, readable by the com.ibm.websphere.servlet.session.IBMSession interface, is associated with a session. An unauthenticated identity is denoted by the user name `anonymous`. WebSphere Application Server includes the com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException class, which is used when a session is requested without the necessary credentials.

The session management facility uses the WebSphere Application Server security infrastructure to determine the authenticated identity associated with a client HTTP request that either retrieves or creates a session. WebSphere Application Server security determines identity using certificates, LPTA, and other methods.

After obtaining the identity of the current request, the session management facility determines whether to return the session requested using a getSession call.

The following table lists possible scenarios in which security integration is enabled with outcomes dependent on whether the HTTP request is authenticated and whether a valid session ID and user name was passed to the session management facility.

	Unauthenticated HTTP request is used to retrieve a session	**HTTP request is authenticated, with an identity of ″FRED″ used to retrieve a session**
No session ID was passed in for this request, or the ID is for a session that is no longer valid	A new session is created. The user name is `anonymous`	A new session is created. The user name is `FRED`
A session ID for a valid session is passed in. The current session user name is ″anonymous″	The session is returned.	The session is returned. session management changes the user name to `FRED`
A session ID for a valid session is passed in. The current session user name is `FRED`	The session is not returned. An UnauthorizedSessionRequestException error is created*	The session is returned.
A session ID for a valid session is passed in. The current session user name is `BOB`	The session is not returned. An UnauthorizedSessionRequestException error is created*	The session is not returned. An UnauthorizedSessionRequestException error is created*

* A com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException error is created to the servlet.

Session management support

WebSphere Application Server provides facilities, grouped under the heading *Session Management*, that support the javax.servlet.http.HttpSession interface described in the Servlet API specification.

In accordance with the Servlet 2.3 API specification, the session management facility supports session scoping by Web modules. Only servlets in the same Web module can access the data associated with a particular session. Multiple requests from the same browser, each specifying a unique Web application, result in multiple sessions with a shared session ID. You can invalidate any of the sessions that share a session ID without affecting the other sessions.

You can configure a session timeout for each Web application. A Web application timeout value of 0 (the default value) means that the invalidation timeout value from the session management facility is used.

When an HTTP client interacts with a servlet, the state information associated with a series of client requests is represented as an HTTP session and identified by a session ID. Session management is responsible for managing HTTP sessions, providing storage for session data, allocating session IDs, and tracking the session ID associated with each client request through the use of cookies or URL rewriting techniques. Session management can store session-related information in several ways:

- In application server memory (the default). This information cannot be shared with other application servers.
- In a database. This storage option is known as *database persistent sessions*.

The last two options are referred to as *distributed sessions*. Distributed sessions are essential for using HTTP sessions for the failover facility. When an application server receives a request associated with a session ID that it currently does not have in memory, it can obtain the required session state by accessing the external store (database or memory-to-memory). If distributed session support is not enabled, an application server cannot access session information for HTTP requests that are sent to servers other than

the one where the session was originally created. Session management implements caching optimizations to minimize the overhead of accessing the external store, especially when consecutive requests are routed to the same application server.

Storing session states in an external store also provides a degree of fault tolerance. If an application server goes offline, the state of its current sessions is still available in the external store. This availability enables other application servers to continue processing subsequent client requests associated with that session.

Saving session states to an external location does not completely guarantee their preservation in case of a server failure. For example, if a server fails while it is modifying the state of a session, some information is lost and subsequent processing using that session can be affected. However, this situation represents a very small period of time when there is a risk of losing session information.

The drawback to saving session states in an external store is that accessing the session state in an external location can use valuable system resources. session management can improve system performance by caching the session data at the server level. Multiple consecutive requests that are directed to the same server can find the required state data in the cache, reducing the number of times that the actual session state is accessed in external store and consequently reducing the overhead associated with external location access.

Session tracking options

HTTP session support also involves session tracking. You can use cookies, URL rewriting, or Secure Sockets Layer (SSL) information for session tracking.

the following tracking methods are available:
- Session tracking with cookies
- Session tracking with URL rewriting
- Session tracking with Secure Sockets Layer (SSL) information

Session tracking with cookies

Tracking sessions with cookies is the default. No special programming is required to track sessions with cookies.

Session tracking with URL rewriting

An application that uses URL rewriting to track sessions must adhere to certain programming guidelines. The application developer needs to do the following:
- Program servlets to encode URLs
- Supply a servlet or JavaServer Pages (JSP) file as an entry point to the application

Using URL rewriting also requires that you enable URL rewriting in the session management facility.

Note: In certain cases, clients cannot accept cookies. Therefore, you cannot use cookies as a session tracking mechanism. Applications can use URL rewriting as a substitute.

Program session servlets to encode URLs

Depending on whether the servlet is returning URLs to the browser or redirecting them, include either the encodeURL method or the encodeRedirectURL method in the servlet code. Examples demonstrating what to replace in your current servlet code follow.

Rewrite URLs to return to the browser

Suppose you currently have this statement:

```
out.println("<a href=\"/store/catalog\">catalog<a>");
```

Change the servlet to call the encodeURL method before sending the URL to the output stream:

```
out.println("<a href=\"");
out.println(response.encodeURL ("/store/catalog"));
out.println("\">catalog</a>");
```

Rewrite URLs to redirect

Suppose you currently have the following statement:

```
response.sendRedirect ("http://myhost/store/catalog");
```

Change the servlet to call the encodeRedirectURL method before sending the URL to the output stream:

```
response.sendRedirect (response.encodeRedirectURL ("http://myhost/store/catalog"));
```

The encodeURL method and encodeRedirectURL method are part of the HttpServletResponse object. These calls check to see if URL rewriting is configured before encoding the URL. If it is not configured, the calls return the original URL.

If both cookies and URL rewriting are enabled and the response.encodeURL method or encodeRedirectURL method is called, the URL is encoded, even if the browser making the HTTP request processed the session cookie.

You can also configure session support to enable protocol switch rewriting. When this option is enabled, the product encodes the URL with the session ID for switching between HTTP and HTTPS protocols.

Supply a servlet or JSP file as an entry point

The entry point to an application, such as the initial screen presented, may not require the use of sessions. However, if the application in general requires session support (meaning some part of it, such as a servlet, requires session support), then after a session is created, all URLs are encoded to perpetuate the session ID for the servlet (or other application component) requiring the session support.

The following example shows how you can embed Java code within a JSP file:

```
<%
response.encodeURL ("/store/catalog");
%>
```

Session tracking with SSL information (Deprecated)

Note: Session tracking using the SSL ID is deprecated in WebSphere Application Server version 7.0. You can configure session tracking to use cookies or URL rewriting.

No special programming is required to track sessions with Secure Sockets Layer (SSL) information.

To use SSL information, turn on **Enable SSL ID tracking** in the session management property page. Because the SSL session ID is negotiated between the Web browser and HTTP server, this ID cannot survive an HTTP server failure. However, the failure of an application server does not affect the SSL session ID if an external HTTP server is present between WebSphere Application Server and the browser.

SSL tracking is supported for the IBM HTTP Server and iPlanet Web servers only. You can control the lifetime of an SSL session ID by configuring options in the Web server. For example, in the IBM HTTP Server, set the configuration variable SSLV3TIMEOUT to provide an adequate lifetime for the SSL session ID. An interval that is too short can cause a premature termination of a session. Also, some Web browsers might have their own timers that affect the lifetime of the SSL session ID. These Web browsers may not leave the SSL session ID active long enough to serve as a useful mechanism for session tracking. The internal HTTP Server of WebSphere Application Server also supports SSL tracking.

When using the SSL session ID as the session tracking mechanism in a cloned environment, use either cookies or URL rewriting to maintain session affinity. The cookie or rewritten URL contains session affinity information that enables the Web server to properly route a session back to the same server for each request.

Distributed sessions

In a distributed environment, you can save sessions in a database using database session persistence or you can store sessions in multiple WebSphere Application Server instances using memory-to-memory session replication.

WebSphere Application Server provides the following session mechanisms in a distributed environment:

- **Database session persistence**, where sessions are stored in the database specified.
- **Memory-to-memory session replication**, where sessions are stored in one or more specified WebSphere Application Server instances or profiles.

When a session contains attributes that implement HttpSessionActivationListener, notification occurs anytime the session is activated (that is, session is read to the memory cache) or passivated (that is, session leaves the memory cache). Passivation can occur because of a server shutdown or when the session memory cache is full and an older session is removed from the memory cache to make room for a newer session. It is not guaranteed that a session is passivated in one application server prior to activation in another application.

Session recovery support

For session recovery support, WebSphere Application Server provides distributed session support in the form of database sessions and memory-to-memory replication. You can use session recovery support when the user's session data must be maintained across a server restart or when the user's session data is too valuable to lose through an unexpected server failure.

All the attributes set in a session must implement java.io.Serializable if the session requires external storage. In general, consider making all objects held by a session serialized, even if immediate plans do not call for session recovery support. If the Web site grows, and session recovery support becomes necessary, the transition occurs transparently to the application if the sessions only hold serialized objects. If not, a switch to session recovery support requires coding changes to make the session contents serialized.

Distributed environment settings

Use this page to specify a type for saving a session in a distributed environment.

To view this administrative console page at the Web container level, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Session management** → **Distributed environment settings**.

Note that the distributed environment settings can be overridden at the application level.

Distributed sessions:

Specifies the type of distributed environment to be used for saving sessions.

None	Specifies that the session management facility discards the session data when the server shuts down.
Database	Specifies that the session management facility stores session information in the data source specified by the data source connection settings. Click **Database** to change these data source settings.

Memory-to-memory replication	Specifies that the session management facility stores the session information in a data source in memory. The session information is copied to other session management facilities for failure recovery. Click **Memory-to-memory replication** to change these data source settings.

Memory-to-memory replication

Memory-to-memory session replication is the session replication to another WebSphere Application Server. In this mode, sessions can replicate to one or more Application Servers to address HTTP Session single point of failure (SPOF).

The WebSphere Application Server instance in which the session is currently processed is referred to as the *owner of the session*. In a clustered environment, session affinity in the WebSphere Application Server plug-in routes the requests for a given session to the same server. If the current owner server instance of the session fails, then the WebSphere Application Server plug-in routes the requests to another appropriate server in the cluster. In a peer-to-peer cluster, the hot failover feature causes the plug-in to failover to a server that already contains the backup copy of the session, avoiding the overhead of session retrieval from another server containing the backup. In a client/server cluster, the server retrieves the session from a server that has the backup copy of the session. The server now becomes the owner of the session and affinity is now maintained to this server.

There are three possible modes to run in:
- **Server mode:** Only store backup copies of other WebSphere Application Server sessions and not to send out copies of any session created in that particular server
- **Client mode:** Only broadcast or send out copies of the sessions it owns and not to receive backup copies of sessions from other servers
- **Both mode:** Simultaneously broadcast or send out copies of the sessions it owns and act as a backup table for sessions owned by other WebSphere Application Server instances.

You can select the replication mode of server, client, or both when configuring the session management facility for memory-to-memory replication. The default is both. This storage option is controlled by the mode parameter.

The memory-to-memory replication function is accomplished by the creation of a data replication service instance in an application server that talks to other data replication service instances in remote application servers. You must configure this data replication service instance as a part of a replication domain. Data replication service instances on disparate application servers that replicate to one another must be configured as a part of the same domain. You must configure all session managers connected to a replication domain to have the same topology. If one session manager instance in a domain is configured to use the client/server topology, then the rest of the session manager instances in that domain must be a combination of servers configured as Client only and Server only. If one session manager instance is configured to use the peer-to-peer topology, then all session manager instances must be configured as Both client and server. For example, a server only data replication service instance and a both client and server data replication service instance cannot exist in the same replication domain. Multiple data replication service instances that exist on the same application server due to session manager memory-to-memory configuration at various levels that are configured to be part of the same domain must have the same mode.

With respect to mode, the following are the primary examples of memory-to-memory replication configuration:
- Peer-to-peer replication
- Client/server replication

Although the administrative console allows flexibility and additional possibilities for memory-to-memory replication configuration, only the configurations provided above are officially supported.

There is a single replica in a cluster by default. You can modify the number of replicas through the replication domain.

HTTP session replication in the controller

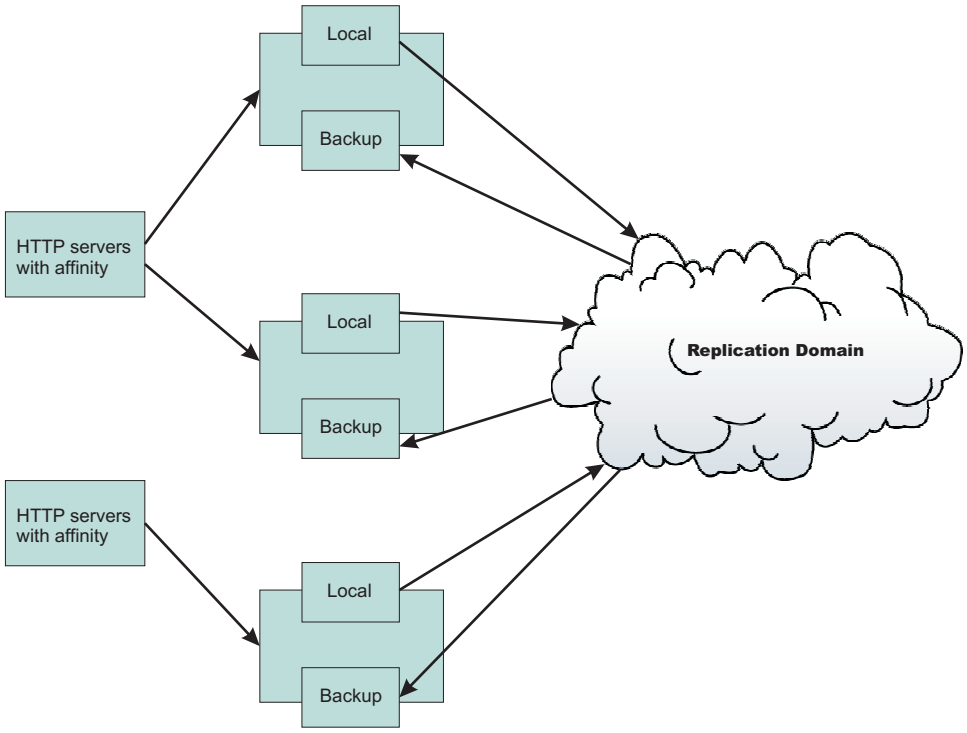
WebSphere Application Servers on z/OS that are enabled for HTTP session memory-to-memory replication can store replicated HTTP session data in the controller and replicate data to other WebSphere Application Servers. HTTP session data that is stored in a controller is retrievable by any of the servants of that controller. HTTP session affinity is still associated to a particular servant; however, if that servant should fail, any of the other servants can retrieve the HTTP session data stored in the controller and establish a new affinity.

The capability of storing HTTP sessions in the controller can also be enabled in unmanaged application servers on z/OS. When this capability is enabled, servants store the HTTP session data in the controller for retrieval when a servant fails which is similar to managed servers. HTTP session data stored in the controller of an unmanaged application server is not retrievable by other application servers and is not replicated to other application servers.

The capability to store HTTP session data in the controller in an unmanaged application server is enabled by setting the JVM custom property HttpSessionEnableUnmanagedServerReplication to true. You can set this property at **Servers > Application servers >** *server_name*. Then, under Server Infrastructure, click **Java and Process Management > Process Definition > Servant > Java Virtual Machine > Custom Properties**.

Memory-to-memory topology: Peer-to-peer function

The basic peer-to-peer (both client and server function, or both mode) topology is the default configuration and has a single replica. However, you can also add additional replicas by configuring the replication domain.



In this basic peer-to-peer topology, each server Java Virtual Machine (JVM) can:

- Host the Web application leveraging the HTTP session
- Send out changes to the HTTP session that it owns
- Receive backup copies of the HTTP session from all of the other servers in the cluster

This configuration represents the most consolidated topology, where the various system parts are collocated and requires the fewest server processes. When using this configuration, the most stable implementation is achieved when each node has equal capabilities (CPU, memory, and so on), and each handles the same amount of work.

It is also important to note that when using the peer-to-peer topology, that one session replication backup must run at all times for invalidation to occur. For example, if you have a cluster of 2 application servers, server1 and server2, that are both configured in the peer-to-peer mode and server2 goes down. All of backup information for server1 is lost and those sessions are no longer invalidated properly.

Session hot failover

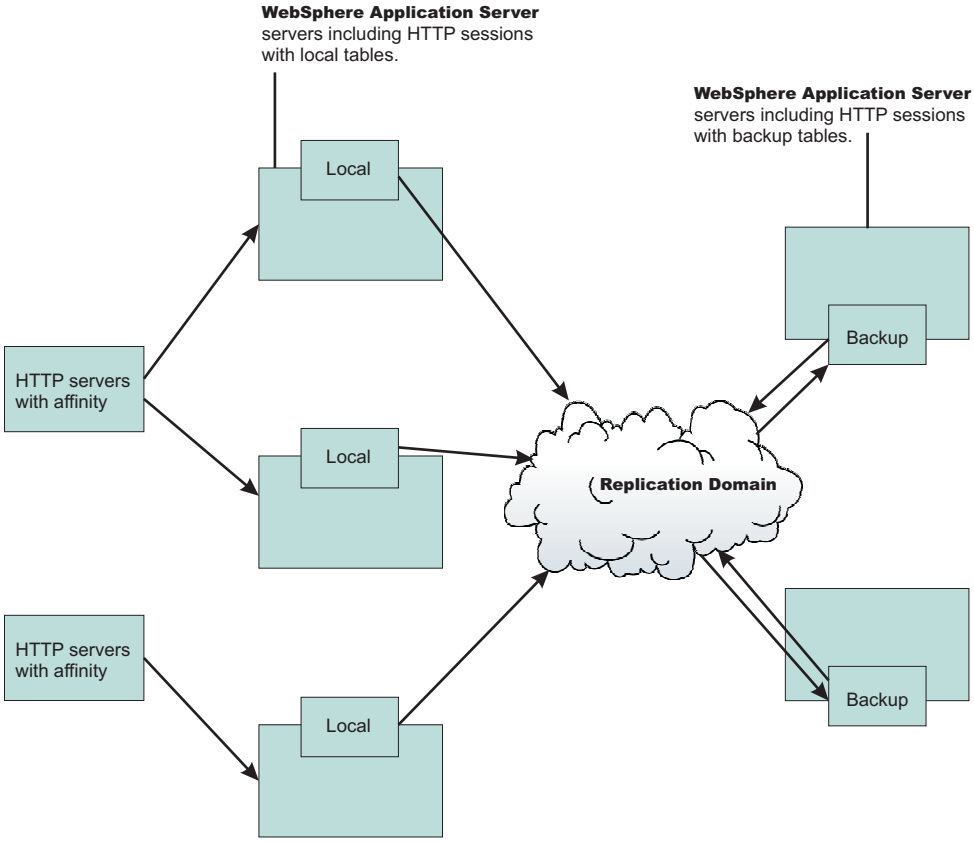
A new feature called session hot failover has been added to this release. This feature is only applicable to the peer-to-peer mode. In a clustered environment, session affinity in the WebSphere Application Server plug-in routes the requests for a given session to the same server. If the current owner server instance of the session fails, then the WebSphere Application Server plug-in routes the requests to another appropriate server in the cluster. For a cluster configured to run in the peer-to-peer mode this feature causes the plug-in to failover to a server that already contains the backup copy of the session, therefore avoiding the overhead of session retrieval from another server containing the backup. However, hot failover is specifically for servant region failures. When an entire server, meaning both controller and server fail, sessions may have to be retrieved over the network.

You must upgrade all WebSphere Application Server plug-in instances that front the Application Server cluster to version 6.0 to ensure session affinity when using the peer-to-peer mode.

Memory-to-memory topology: Client/server function

The client/server configuration, used to attain session affinity, consists of a cluster of servers that are configured as client only and server only. Using the client/server configuration has benefits such as isolating the handling of backup data from local data, recycling backup servers without affecting the servers running the application, and removing the need for a one-to-one correspondence between servers to attain session affinity.

The following figure depicts the client/server mode. There is a tier of applications servers that host Web applications using HTTP sessions, and these sessions are replicated out as they are created and updated. There is a second tier of servers without a Web application installed, where the session manager receives updates from the replication clients.



Benefits of the client/server configuration include:

Isolation for failure recovery

In this case we are isolating the handling of backup data from local data; aside from isolating the moving parts in case of a catastrophic failure in one of them, you again free up memory and processing in the servers processing the Web application.

Isolation for stopping and starting

You can recycle a backup server without affecting the servers running the application (when there are two or more backups, failure recovery is possible), and conversely recycle an application JVM without potentially losing that backup data for someone.

Consolidation

There is most likely no need to have a one-to-one correspondence between servers handling backups and those processing the applications; hence, you are again reducing the number of places to which you transfer the data.

Disparate hardware:

While you run your Web applications on cheaper hardware, you may have one or two more powerful computers in the back end of your enterprise that have the capacity to run a couple of session managers in replication server mode; allowing you to free up your cheaper Web application hardware to process the Web application.

Timing consideration: Start the backup application servers first to avoid unexpected timing windows. The clients attempt to replicate information and HTTP sessions to the backup servers as soon as they come up. As a result, HTTP sessions that are created prior to the time at which the servers come up might not replicate successfully.

Memory-to-memory session partitioning

Session partitioning gives the administrator the ability to filter or reduce the number of destinations that the session object gets sent to by the replication service. You can also configure session partitioning by specifying the number of replicas on the replication domain. The Single replica option is chosen by default. Since the number of replicas is global for the entire replication domain, all the session managers connected to the replication domain use the same setting.

Single replica

You can replicate a session to only one other server, creating a single replica. When this option is chosen, a session manager picks another session manager that is connected to the same replication domain to replicate the HTTP session to during session creation. All updates to the session are only replicated to that single server. This option is set at the replication domain level. When this option is set, every session manager connected to this replication domain creates a single backup copy of HTTP session state information on a backup server.

Full group replica

Each object is replicated to every application server that is configured as a consumer of the replication domain. However, in the peer-to-peer mode, this topology is the most redundant because everyone replicates to everyone and as you add servers, more overhead (both CPU and memory) is needed to deal with replication. This mode is most useful for dynamic caching replication. Redundancy does not affect the client/server mode because clients only replicate to servers that are set to server mode.

Specific number of replicas

You can specify a specific number of replicas for any entry that is created in the replication domain. The number of replicas is the number of application servers that the user wants to use to replicate in the domain. This option eliminates redundancy that occurs in a full group replica and also provides additional backup than a single replica. In peer-to-peer mode, the number of replicas cannot exceed the total number of application servers in the cluster. In the client/server mode, the number of replicas cannot exceed the total number of application servers in the cluster that are set to server mode.

Clustered session support

A clustered environment supports load balancing, where the workload is distributed among the application servers that compose the cluster.

In a cluster environment, the same Web application must exist on each of the servers that can access the session. You can accomplish this setup by installing an application onto a cluster definition. Each of the servers in the group can then access the Web application

In a clustered environment, the session management facility requires an affinity mechanism so that all requests for a particular session are directed to the same application server instance in the cluster. This requirement conforms to the Servlet 2.3 specification in that multiple requests for a session cannot coexist in multiple application servers. One such solution provided by IBM WebSphere Application Server is *session affinity* in a cluster; this solution is available as part of the WebSphere Application Server plug-ins

for Web servers. It also provides for better performance because the sessions are cached in memory. In clustered environments other than WebSphere Application Server clusters, you must use an affinity mechanism (for example, IBM WebSphere Edge Server affinity).

If one of the servers in the cluster fails, it is possible for the request to reroute to another server in the cluster. If distributed sessions support is enabled, the new server can access session data from the database or another WebSphere Application Server instance. You can retrieve the session data only if a new server has access to an external location from which it can retrieve the session.

Session management tuning

WebSphere Application Server session support has features for tuning session performance and operating characteristics, particularly when sessions are configured in a distributed environment. These options support the administrator flexibility in determining the performance and failover characteristics for their environment.

The following table summarizes the features, including whether they apply to sessions tracked in memory, in a database, with memory-to-memory replication, or all. Some features are easily manipulated using administrative settings; others require code or database changes.

Feature or option	Goal	Applies to sessions in memory, database, or memory-to-memory
Write frequency	Minimize database write operations.	Database and Memory-to-Memory
Session affinity	Access the session in the same application server instance.	All
Multirow schema	Fully utilize database capacities.	Database
Base in-memory session pool size	Fully utilize system capacity without overburdening system.	All
Write contents	Allow flexibility in determining what session data to write	Database and Memory-to-Memory
Scheduled invalidation	Minimize contention between session requests and invalidation of sessions by the Session Management facility. Minimize write operations to database for updates to last access time only.	Database and Memory-to-Memory
Tablespace and row size	Increase efficiency of write operations to database.	Database (DB2® only)

HTTP sessions: Resources for learning

Use the following links to find relevant supplemental information about HTTP sessions. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks® that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:

Programming model and decisions

- Improving session persistence performance with DB2

Programming instructions and examples
- Java Servlet documentation, tutorials, and examples site

Programming specifications
- Java Servlet 2.4 API specification download site
- J2EE 1.4 specification download site

Scheduled invalidation
Instead of relying on the periodic invalidation timer that runs on an interval based on the session timeout parameter, you can set specific times for the session management facility to scan for invalidated sessions in a distributed environment.

When used with distributed sessions, this feature has the following benefits:
- You can schedule the scan for invalidated sessions for times of low application server activity, avoiding contention between invalidation scans of database or another WebSphere Application Server instance and read and write operations to service HTTP session requests.
- Significantly fewer external write operations can occur when running with the End of Service Method Write mode because the last access time of the session does not need to be written out on each HTTP request. (Manual Update options and Time Based Write options already minimize the writing of the last access time.)

Usage considerations
- The session manager invalidates sessions only at the scheduled time, therefore sessions are available to an application if they are requested before the session is invalidated.
- With scheduled invalidation configured, HttpSession timeouts are not strictly enforced. Instead, all invalidation processing is handled at the configured invalidation times.
- HttpSessionBindingListener processing is handled at the configured invalidation times unless the HttpSession.invalidate method is explicitly called.
- The HttpSession.invalidate method immediately invalidates the session from both the session cache and the external store.
- The periodic invalidation thread still runs with scheduled invalidation. If the current hour of the day does not match one of the configured hours, sessions that have exceeded the invalidation interval are removed from cache, but not from the external store. Another request for that session results in returning that session back into the cache.
- When the periodic invalidation thread runs during one of the configured hours, all sessions that have exceeded the invalidation interval are invalidated by removal from both the cache and the external store.
- The periodic invalidation thread can run more than once during an hour and does not necessarily run exactly at the top of the hour.
- If you specify the interval for the periodic invalidation thread using the HttpSessionReaperPollInterval custom property, do not specify a value of more than 3600 seconds (1 hour) to ensure that invalidation processing happens at least once during each hour.

Base in-memory session pool size
The base in-memory session pool size number depends on the session support configuration.
- With in-memory sessions, session access is optimized for up to this number of sessions.
- With distributed sessions, when sessions are stored in a database or in another WebSphere Application Server instance,; the pool size also specifies the cache size and the number of last access time updates saved in manual update mode.

For distributed sessions, when the session cache has reached its maximum size and a new session is requested, the Session Management facility removes the least recently used session from the cache to make room for the new one.

General memory requirements for the hardware system, and the usage characteristics of the e-business site, determines the optimum value.

Note that increasing the base in-memory session pool size can necessitate increasing the heap sizes of the Java processes for the corresponding WebSphere Application Servers.

Overflow in non-distributed sessions

By default, the number of sessions maintained in memory is specified by base in-memory session pool size. If you do not wish to place a limit on the number of sessions maintained in memory and allow overflow, set overflow to *true*.

Allowing an unlimited amount of sessions can potentially exhaust system memory and even allow for system sabotage. Someone could write a malicious program that continually hits your site and creates sessions, but ignores any cookies or encoded URLs and never utilizes the same session from one HTTP request to the next.

When overflow is disallowed, the Session Management facility still returns a session with the HttpServletRequest getSession(true) method when the memory limit is reached, and this is an invalid session that is not saved.

With the WebSphere Application Server extension to HttpSession, com.ibm.websphere.servlet.session.IBMSession, an isOverflow method returns *true* if the session is such an invalid session. An application can check this status and react accordingly.

HTTP session invalidation

HTTP sessions are invalidated by calling the invalidate method on the session object or by specifying a specific time interval using the MaxInactiveInterval property.

Sessions that are invalidated explicitly by application code are invalidated immediately. Sessions that are not invalidated by application code are invalidated by the session manager. Session invalidation occurs regardless of session persistence configuration.

A session is a candidate for invalidation if it has not been accessed for a period that is longer than the specified session timeout, specified by the MaxInactiveInterval value. The session manager has an invalidation process thread that runs every X seconds to invalidate sessions that are eligible for invalidation.

The session manager uses a formula to determine the value of X, specified by the ReaperInterval property. The value of X is calculated based on the MaxInactiveInterval value that is specified in the session manager.

For example, for a maximum inactive interval less than 15 minutes, the ReaperInterval value is approximately 60 to 90 seconds. For a maximum inactive interval greater than 15 minutes, the ReaperInterval value is approximately 300 to 360 seconds.

A session is invalidated when the MaxInactiveInterval is exceeded and the ReaperInterval passes. After a session is eligible for invalidation, the invalidation thread must run for the session to be invalidated. Therefore, a session might not be invalidated for the sum of the MaxInactiveInterval and ReaperInterval value in seconds.

A session that has exceeded the MaxInactiveInterval but is not yet removed by the invalidation thread is still available for use. If that session is requested then it is returned to the client.

You can specify whether the session is invalidated immediately or after a specified time interval. For immediate invalidation the application should call the invalidate method. To invalidate a session at a specific time, you can set the ReaperInterval Web container custom property in seconds to specify the frequency of the invalidation thread.

Write operations

You can manually control when modified session data is written out to the database or to another WebSphere Application Server instance by using the sync method in the com.ibm.websphere.servlet.session.IBMSession interface. The manual update, end of service servlet and the time based write frequency modes are available to tune write frequency of session data.

This interface extends the javax.servlet.http.HttpSession interface. By calling the sync method from the service method of a servlet, you send any changes in the session to the external location. When manual update is selected as the write frequency mode, session data changes are written to an external location only if the application calls the sync method. If the sync method is not called, session data changes are lost when a session object leaves the server cache. When end of service servlet or time based is the write frequency mode, the session data changes are written out whenever the sync method is called. If the sync method is not called, changes are written out at the end of service method or on a time interval basis based on the write frequency mode that is selected.

```
IBMSession iSession = (IBMSession) request.getSession();
iSession.setAttribute("name", "Bob");

//force write to external store
iSession.sync( )
```

If the database is down or is having difficulty connecting during an update to session values, the sync method always makes three attempts before it finally creates a BackedHashtable.getConnectionError error. For each connection attempt that fails, the BackedHashtable.StaleConnectionException is created and can be found in the sync method. If the database opens during any of these three attempts, the session data in the memory is then persisted and committed to the database.

However, if the database is still not up after the three attempts, then the session data in the memory is persisted only after the next check for session invalidation. Session invalidation is checked by a separate thread that is triggered every five minutes. The data in memory is consistent unless a request for session data is issued to the server between these events. For example, if the request for session data is issued within five minutes, then the previous persisted session data is sent.

Sessions are not transactional resources. Because the sync method is associated with a separate thread than the client, the exception that is created does not propagate to the client, which is running on the primary thread. Transactional integrity of data can be maintained through resources such as enterprise beans.

Related reference

“Session management tuning” on page 20
WebSphere Application Server session support has features for tuning session performance and operating characteristics, particularly when sessions are configured in a distributed environment. These options support the administrator flexibility in determining the performance and failover characteristics for their environment.

Tuning parameter settings

Use this page to set tuning parameters for distributed sessions.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Session management** → **Distributed environment settings** → **Custom tuning parameters**.

Tuning level
Specifies that the session management facility provides certain predefined settings that affect performance.

Select one of these predefined settings or customize a setting. To customize a setting, select one of the predefined settings that comes closest to the setting desired, click **Custom settings**, make your changes, and then click **OK**.

Very high (optimize for performance)

Write frequency	Time based
Write interval	300 seconds
Write contents	Only updated attributes
Schedule sessions cleanup	true
First time of day default	0
Second time of day default	2

High

Write frequency	Time based
Write interval	300 seconds
Write contents	All session attributes
Schedule sessions cleanup	false

Medium

Write frequency	End of servlet service
Write contents	Only updated attributes
Schedule sessions cleanup	false

Low (optimize for failover)

Write frequency	End of servlet service
Write contents	All session attributes
Schedule sessions cleanup	false

Custom settings

Write frequency default	Time based
Write interval default	10 seconds
Write contents default	All session attributes
Schedule sessions cleanup default	false

Tuning parameter custom settings
Use this page to customize tuning parameters for distributed sessions.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Session management** → **Distributed environment settings** → **Custom tuning parameters** → **Custom settings**.

Write frequency
Specifies when the session is written to the persistent store.

End of servlet service
A session writes to a database or another WebSphere Application Server instance after the servlet completes execution.

Manual update
A programmatic sync on the IBMSession object is required to write the session data to the database or another WebSphere Application Server instance.

Time based
Session data writes to the database or another WebSphere Application Server instance based on the specified Write interval value. Default: 10 seconds

Write contents
Specifies whether updated attributes are only written to the external location or all of the session attributes are written to the external location, regardless of whether or not they changed. The external location can be either a database or another application server instance.

Only updated attributes
Only updated attributes are written to the persistent store.

All session attribute
All attributes are written to the persistent store.

Schedule sessions cleanup
Specifies when to clean the invalid sessions from a database or another application server instance.

Specify distributed sessions cleanup schedule
Enables the scheduled invalidation process for cleaning up the invalidated HTTP sessions from the external location. Enable this option to reduce the number of updates to a database or another application server instance required to keep the HTTP sessions alive. When this option is not enabled, the invalidator process runs every few minutes to remove invalidated HTTP sessions.

When this option is enabled, specify the two hours of a day for the process to clean up the invalidated sessions in the external location. Specify the times when there is the least activity in the application servers. An external location can be either a database or another application server instance.

First Time of Day (0 - 23)
Indicates the first hour during which the invalidated sessions are cleared from the external location. Specify this value as a positive integer between 0 and 23. This value is valid only when schedule invalidation is enabled.

Second Time of Day (0 - 23)
Indicates the second hour during which the invalidated sessions are cleared from the external location. Specify this value as a positive integer between 0 and 23. This value is valid only when schedule invalidation is enabled.

Best practices for using HTTP sessions
This topic presents best practices for the implementation of HTTP sessions.

Note: Browse the following recommendations for implementing HTTP sessions.

- **Enable Security integration for securing HTTP sessions**

  HTTP sessions are identified by session IDs. A session ID is a pseudo-random number generated at the runtime. Session hijacking is a known attack HTTP sessions and can be prevented if all the requests going over the network are enforced to be over a secure connection (meaning, HTTPS). But not every configuration in a customer environment enforces this constraint because of the performance impact of SSL connections. Due to this relaxed mode, HTTP session is vulnerable to hijacking and because of this vulnerability, WebSphere Application Server has the option to tightly integrate HTTP sessions and

WebSphere Application Server security. Enable security in WebSphere Application Server so that the sessions are protected in a manner that only users who created the sessions are allowed to access them.

- **Release HttpSession objects using `javax.servlet.http.HttpSession.invalidate()` when finished.**

  HttpSession objects live inside the Web container until:
  - The application explicitly and programmatically releases it using the javax.servlet.http.HttpSession.invalidate method; quite often, programmatic invalidation is part of an application logout function.
  - WebSphere Application Server destroys the allocated HttpSession when it expires (default = 1800 seconds or 30 minutes). The WebSphere Application Server can only maintain a certain number of HTTP sessions in memory based on session management settings. In case of distributed sessions, when maximum cache limit is reached in memory, the session management facility removes the least recently used (LRU) one from cache to make room for a session.

  .
- **Avoid trying to save and reuse the HttpSession object outside of each servlet or JSP file.**

  The HttpSession object is a function of the HttpRequest (you can get it only through the req.getSession method), and a copy of it is valid only for the life of the service method of the servlet or JSP file. You *cannot* cache the HttpSession object and refer to it outside the scope of a servlet or JSP file.
- **Implement the java.io.Serializable interface when developing new objects to be stored in the HTTP session.**

  Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Implementing the java.io.Serializable interface allows the object to properly serialize when using distributed sessions. Classes that do not implement this interface will not have their states serialized or deserialized. Therefore, if a class does not implement the Serializable interface, the JVM cannot persist its state into a database or into another JVM. All subtypes of a serializable class are serializable. An example of this follows:

  ```
  public class MyObject implements java.io.Serializable {...}
  ```

  Make sure all instance variable objects that are not marked transient are serializable. You cannot cache a non-serializable object.

  In compliance with the Java Servlet specification, the distributed servlet container must create an IllegalArgumentException for objects when the container cannot support the mechanism necessary for migration of the session storing them. An exception is created only when you have selected distributable.
- **The HTTPSession API does not dictate transactional behavior for sessions.**

  Distributed HTTPSession support does not guarantee transactional integrity of an attribute in a failover scenario or when session affinity is broken. Use transactional aware resources like enterprise Java beans to guarantee the transaction integrity required by your application.
- **Ensure the Java objects you add to a session are in the correct class path.**

  If you add Java objects to a session, place the class files for those objects in the correct class path (the application class path if utilizing sharing across Web modules in an enterprise application, or the Web module class path if using the Servlet 2.2-complaint session sharing) or in the directory containing other servlets used in WebSphere Application Server. In the case of session clustering, this action applies to every node in the cluster.

  Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.
- **Avoid storing large object graphs in the HttpSession object.**

  In most applications each servlet only requires a fraction of the total session data. However, by storing the data in the HttpSession object as one large object, an application forces WebSphere Application Server to process all of it each time.
- **Utilize Session Affinity to help achieve higher cache hits in the WebSphere Application Server.**

  WebSphere Application Server has functionality in the HTTP Server plug-in to help with session affinity. The plug-in reads the cookie data (or encoded URL) from the browser and helps direct the request to

the appropriate application or clone based on the assigned session key. This functionality increases use of the in-memory cache and reduces hits to the database or another WebSphere Application Server instance

- **Maximize use of session affinity and avoid breaking affinity.**

  Using session affinity properly can enhance the performance of the WebSphere Application Server. Session affinity in the WebSphere Application Server environment is a way to maximize the in-memory cache of session objects and reduce the amount of reads to the database or another WebSphere Application Server instance. Session affinity works by caching the session objects in the server instance of the application with which a user is interacting. If the application is deployed in multiple servers of a server group, the application can direct the user to any one of the servers. If the users starts on server1 and then comes in on server2 a little later, the server must write all of the session information to the external location so that the server instance in which server2 is running can read the database. You can avoid this database read using session affinity. With session affinity, the user starts on server1 for the first request; then for every successive request, the user is directed back to server1. Server1 has to look only at the cache to get the session information; server1 never has to make a call to the session database to get the information.

  You can improve performance by not breaking session affinity. Some suggestions to help avoid breaking session affinity are:
  - Combine all Web applications into a single application server instance, if possible, and use modeling or cloning to provide failover support.
  - Create the session for the frame page, but do not create sessions for the pages within the frame when using multi-frame JSP files. (See discussion later in this topic.)
- **When using multi-framed pages, follow these guidelines:**
  - Create a session in only one frame or before accessing any frame sets. For example, assuming there is no session already associated with the browser and a user accesses a multi-framed JSP file, the browser issues concurrent requests for the JSP files. Because the requests are not part of any session, the JSP files end up creating multiple sessions and all of the cookies are sent back to the browser. The browser honors only the last cookie that arrives. Therefore, only the client can retrieve the session associated with the last cookie. Creating a session before accessing multi-framed pages that utilize JSP files is recommended.
  - By default, JSP files get a HTTPSession using `request.getSession(true)` method. So by default JSP files create a new session if none exists for the client. Each JSP page in the browser is requesting a new session, but only one session is used per browser instance. A developer can use `<% @ page session="false" %>` to turn off the automatic session creation from the JSP files that do not access the session. Then if the page needs access to the session information, the developer can use `<%HttpSession session = javax.servlet.http.HttpServletRequest.getSession(false); %>` to get the already existing session that was created by the original session creating JSP file. This action helps prevent breaking session affinity on the initial loading of the frame pages.
  - Update session data using only one frame. When using framesets, requests come into the HTTP server concurrently. Modifying session data within only one frame so that session changes are not overwritten by session changes in concurrent frameset is recommended.
  - Avoid using multi-framed JSP files where the frames point to different Web applications. This action results in losing the session created by another Web application because the JSESSIONID cookie from the first Web application gets overwritten by the JSESSIONID created by the second Web application.
- **Secure all of the pages (not just some) when applying security to servlets or JSP files that use sessions with security integration enabled, .**

  When it comes to security and sessions, it is all or nothing. It does not make sense to protect access to session state only part of the time. When security integration is enabled in the session management facility, all resources from which a session is created or accessed must be either secured or unsecured. You cannot mix secured and unsecured resources.

  The problem with securing only a couple of pages is that sessions created in secured pages are created under the identity of the authenticated user. Only the same user can access sessions in other secured pages. To protect these sessions from use by unauthorized users, you cannot access these

sessions from an unsecured page. When a request from an unsecured page occurs, access is denied and an `UnauthorizedSessionRequestException` error is created. (`UnauthorizedSessionRequestException` is a runtime exception; it is logged for you.)

- **Use manual update and either the sync() method or time-based write in applications that read session data, and update infrequently.**

  With END_OF_SERVICE as write frequency, when an application uses sessions and anytime data is read from or written to that session, the LastAccess time field updates. If database sessions are used, a new write to the database is produced. This activity is a performance hit that you can avoid using the Manual Update option and having the record written back to the database only when data values update, not on every read or write of the record.

  To use manual update, turn it on in the session management service. (See the tables above for location information.) Additionally, the application code must use the `com.ibm.websphere.servlet.session.IBMSession` class instead of the generic HttpSession. Within the IBMSession object there is a sync method. This method tells the WebSphere Application Server to write the data in the session object to the database. This activity helps the developer to improve overall performance by having the session information persist only when necessary.

  Note: An alternative to using the manual updates is to utilize the timed updates to persist data at different time intervals. This action provides similar results as the manual update scheme.
- Implement the following suggestions to achieve high performance:
  - If your applications do not change the session data frequently, use Manual Update and the sync function (or timed interval update) to efficiently persist session information.
  - Keep the amount of data stored in the session as small as possible. With the ease of using sessions to hold data, sometimes too much data is stored in the session objects. Determine a proper balance of data storage and performance to effectively use sessions.
  - If using database sessions, use a dedicated database for the session database. Avoid using the application database. This helps to avoid contention for JDBC connections and allows for better database performance.
  - If using memory-to-memory sessions, employ partitioning (either group or single replica) as your clusters grow in size and scaling decreases.
  - Verify that you have the latest fix packs for the WebSphere Application Server.
- Utilize the following tools to help monitor session performance.
  - Run the com.ibm.servlet.personalization.sessiontracking.IBMTrackerDebug servlet. - To run this servlet, you must have the servlet invoker running in the Web application you want to run this from. Or, you can explicitly configure this servlet in the application you want to run.
  - Use the WebSphere Application Server Resource Analyzer which comes with WebSphere Application Server to monitor active sessions and statistics for the WebSphere Application Server environment.
  - Use database tracking tools such as ″Monitoring″ in DB2. (See the respective documentation for the database system used.)

HTTP session manager troubleshooting tips

This article provides troubleshooting tips for problems creating or using HTTP sessions with your Web application hosted by WebSphere Application Server.

Here are some steps to take:

- View the logs for the application server which hosts the problem application:
  - first, look at messages written while each application is starting. They will be written between the following two messages:

    ```
    Starting application: application
    ......................
    Application started: application
    ```
  - Within this block, look for any errors or exceptions containing a package name of com.ibm.ws.webcontainer.httpsession. If none are found, this is an indication that the session manager started successfully.

  - Error ″**SRVE0054E: An error occurred while loading session context and Web application**″ indicates that SessionManager didn't start properly for a given application.
  - Look within the logs for any Session Manager related messages. These messages will be in the format SESNxxxxE and SESNxxxxW for errors and warnings, respectively, where xxxx is a number identifying the precise error. Look up the extended error definitions in the Session Manager message table.
- See the Best practices for using HTTP Sessions section in the *Developing and deploying applications* PDF books for more details.
- Alternatively, a special servlet can be invoked that displays the current configuration and statistics related to session tracking.
  - Servlet name: **com.ibm.ws.webcontainer.httpsession.IBMTrackerDebug**.
  - It can be invoked from any Web module which is enabled to serve by class name. For example, using default_app, **http://localhost:9080/servlet/com.ibm.ws.webcontainer.httpsession.IBMTrackerDebug**.
  - If you are viewing the module via the serve-by-class-name feature, be aware that it may be viewable by anyone who can view the application. You may wish to map a specific, secured URL to the servlet instead and disable the serve-servlets-by-classname feature.
- If you are using **database-based persistent sessions**, look for problems related to the **data source** the Session Manager relies on to keep session state information. For details on diagnosing database related problems see the Errors accessing a datasource or connection pool in the *Administering applications and their environment* PDF book

Error message `SRVE0079E Servlet host not found` after you define a port

Error message SRVE0079E can occur after you define the port in WebContainer > HTTP Transports for a server, indicating that you do not have the port defined in your virtual host definitions. To define the port,

1. On the administrative console, go to Environment > Virtual Hosts > default_host> Host Aliases> New
2. Define the new port on host ″*″

The application server gets EC3 - 04130007 ABENDs

To prevent an EC3 - 04130007 abend from occuring on the application server, change the HTTP Output timeout value. The custom property *ConnectionResponseTimeout* specifies the maximum number of seconds the HTTP port for an individual server can wait when trying to read or write data. For instructions on how to set *ConnectionResponseTimeout*, see HTTP transport custom properties section of the *Administering applications and their environment* PDF book.

If none of these steps fixes your problem, check to see if the problem has been identified and documented by looking at the available online support (hints and tips, technotes, and fixes). If you don't find your problem listed there contact IBM support.

For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

HTTP session problems

This article provides troubleshooting information related to creating or using Hypertext Transfer Protocol (HTTP) sessions.

To view and update the session manager settings discussed here, use the administrative console. Select the application server that hosts the problem application, then under **Additional properties**, select **Web Container**, then **Session manager**.

What kind of problem are you having?
- HTTP Sessions are not getting created, or are lost between requests.
- HTTP Sessions are not persistent (session data lost when application server restarts, or not shared across cluster).
- Session is shared across multiple browsers on same client machine.
- Session is not getting invalidated immediately after specified session timeout interval.
- Unwanted sessions are being created by JavaServer Pages.
- Session data intended for one client is seen by another client.
- A ClassCastException error occurs during failover of a session that contains an Enterprise JavaBeans™ (EJB) reference.
- Users are not logged out after HTTP session timer expires

HTTP sessions are not getting created, or are lost between requests

By default, the session manager uses cookies to store the session ID on the client between requests. Unless you intend to avoid cookie-based session tracking, ensure that cookies are flowing between WebSphere Application Server and the browser:
- Make sure the **Enable cookies** check box is checked under the **Session tracking Mechanism** property.
- Make sure cookies are enabled on the browser you are testing from or from which your users are accessing the application.
- Check the Cookie domain specified on the SessionManager (to view the or update the cookie settings, in the **Session tracking mechanism->enable cookies** property, click **Modify**).
  - For example, if the cookie domain is set as ″.myCom.com″, resources should be accessed using that domain name. Example: http://www.myCom.com/myapp/servlet/sessionservlet.
  - If the domain property is set, make sure it begins with a dot (.). Certain versions of Netscape do not accept cookies if domain name doesn't start with a dot. Internet Explorer honors the domain with or without a dot. For example, if the domain name is set to *mycom.com*, change it to *.mycom.com* so that both Netscape and Internet Explorer honor the cookie.

    Note: When the servers are on different hosts, ensure that session cookies flow to all the servers by configuring a front-end router such as a Web server with the plug-in or setting the Cookie domain.
- Check the **Cookie path** specified on the SessionManager. Check whether the problem URL is hierarchically below the Cookie path specified. If not correct the Cookie path.
- If the `Cookie maximum age` property is set, ensure that the client (browser) machine's date and time is the same as the server's, including the time zone. If the client and the server time difference is over the ″Cookie maximum age″ then every access would be a new session, since the cookie expires after the access.
- If you have multiple Web modules within an enterprise application that track sessions:
  - If you want to have different session settings among Web modules in an enterprise application, ensure that each Web module specifies a different cookie name or path, or
  - If Web modules within an enterprise application use a common cookie name and path, ensure that the HTTP session settings, such as Cookie maximum age, are the same for all Web modules. Otherwise cookie behavior is unpredictable, and depends upon which application creates the session. Note that this does not affect session data, which is maintained separately by Web module.
- Check the cookie flow between browser and server:
  1. On the browser, enable ″cookie prompt″. Hit the servlet and make sure cookie is being prompted.
  2. Access the session servlet from the browser.
  3. The browser prompts for the cookie; note the jsessionid.
  4. Reload the servlet, note down the cookie if a new cookie is sent.
  5. Check the session trace and look for the session ID and trace the request by the thread. Verify that the session is stable across Web requests:
     - Look for **getIHttpsession(...)** which is start of session request.
     - Look for **releaseSession(..)** which is end of servlet request.
- If you are using URL rewriting instead of cookies:

  – Ensure there are no static HTML pages on your application's navigation path.
•

  Note: Session tracking using the SSL ID is deprecated in WebSphere Application Server version 7.0. You can configure session tracking to use cookies or modify the application to use URL rewriting.

  If you are using SSL as your session tracking mechanism:
  – Ensure that you have SSL enabled on your IBM HTTP Server or iPlanet HTTP server.
- If you are in a clustered (multiple node) environment, ensure that you have session persistence enabled.

HTTP Sessions are not persistent

If your HTTP sessions are not persistent, that is session data is lost when the application server restarts or is not shared across the cluster:
- Check the data source.
- Check the session manager's persistence settings properties:
  – If you intend to take advantage of session persistence, verify that Persistence is set to **Database**.
  – If you are using **Database-based persistence**:
    - Check the JNDI name of the data source specified correctly on SessionManager.
    - Specify correct userid and password for accessing the database.

      Note that these settings have to be checked against the properties of an existing data source in the administrative console. The session manager does not automatically create a session database for you.
    - The data source should be non-JTA, for example, non XA enabled.
    - Check the logs for appropriate database error messages.
    - With DB2, for row sizes other than 4k make sure specified row size matches the DB2 page size. Make sure tablespace name is specified correctly.
  – If you are using **memory-based persistence** (available only in a network deployment environment):
    - Review the Memory-to-memory replication section of the *Developing and deploying applications* PDF book.
    - Review the **Internal Replication Domains properties** of your session manager.

Session is shared across multiple browsers on same client machine

This behavior is browser-dependent. It varies between browser vendors, and also may change according to whether a browser is launched as a new process or as a subprocess of an existing browser session (for example by hitting Ctl-N on Windows®).

The Cookie maximum age property of the session manager also affects this behavior, if cookies are used as the session-tracking mechanism. If the maximum age is set to some positive value, all browser instances share the cookies, which are persisted to file on the client for the specified maximum age time.

Session is not getting invalidated immediately after specified session timeout interval

The SessionManager invalidation process thread runs every *x* seconds to invalidate any invalid sessions, where x is determined based on the session timeout interval specified in the session manager properties. For the default value of 30 minutes , *x* is around 300 seconds. In this case, it could take up to 5 minutes (300 seconds) beyond the timeout threshold of 30 minutes for a particular session to become invalidated.

Unwanted sessions are being created by JavaServer Pages

As required by the JavaServer Pages (JSP) specification, JSP pages by default perform a request.getSession(true), so that a session is created if none exists for the client. To prevent JSP pages from creating a new session, set the session scope to **false** in the .jsp file using the page directive as follows:

```
<% @page session="false" %>
```

Users are not logged out after the HTTP session timer expires

If users of WebSphere Application Server log onto an application and sit idle longer than the specified HTTP session timeout value, the user information is not invalidated and user credentials stay active until LTPA token timeout occurs.

After you apply PK25740, complete the following steps to log out users from the application after the HTTP session has expired.

1. In the administrative console, click **Security > Global security**.
2. Under Custom properties, click **New**.
3. In the Name field, enter com.ibm.ws.security.web.logoutOnHTTPSessionExpire.
4. In the Values field, enter true.
5. Click Apply and Save to save the changes to your configuration.
6. Resynchronize and restart the server.

IBM Support has documents and tools that can save you time gathering information needed to resolve problems as described in Troubleshooting help from IBM. Before opening a problem report, see the Support page:

- http://www.ibm.com/software/webservers/appserv/zos_os390/support/

Developing session management in servlets

About this task

This information, combined with the coding example SessionSample.java, provides a programming model for implementing sessions in your own servlets.

1. Get the HttpSession object.

   To obtain a session, use the getSession method of the javax.servlet.http.HttpServletRequest object in the Java Servlet 2.3 API.

   When you first obtain the HttpSession object, the Session Management facility uses one of three ways to establish tracking of the session: cookies, URL rewriting, or Secure Sockets Layer (SSL) information.

   Note: Session tracking using the SSL ID is deprecated in WebSphere Application Server version 7.0. You can configure session tracking to use cookies or modify the application to use URL rewriting

   Assume the Session Management facility uses cookies. In such a case, the Session Management facility creates a unique session ID and typically sends it back to the browser as a *cookie*. Each subsequent request from this user (at the same browser) passes the cookie containing the session ID, and the Session Management facility uses this ID to find the user's existing HttpSession object.

   In Step 1 of the code sample, the Boolean(create) is set to `true` so that the HttpSession object is created if it does not already exist. (With the Servlet 2.3 API, the javax.servlet.http.HttpServletRequest.getSession() method with no boolean defaults to `true` and creates a session if one does not already exist for this user.)
2. Store and retrieve user-defined data in the session.

   After a session is established, you can add and retrieve user-defined data to the session. The HttpSession object has methods similar to those in java.util.Dictionary for adding, retrieving, and removing arbitrary Java objects.

In Step 2 of the code sample, the servlet reads an integer object from the HttpSession, increments it, and writes it back. You can use any name to identify values in the HttpSession object. The code sample uses the name sessiontest.counter.

Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

3. (Optional) Output an HTML response page containing data from the HttpSession object.
4. Provide feedback to the user that an action has taken place during the session. You may want to pass HTML code to the client browser indicating that an action has occurred. For example, in step 3 of the code sample, the servlet generates a Web page that is returned to the user and displays the value of the sessiontest.counter each time the user visits that Web page during the session.
5. (Optional) Notify Listeners. Objects stored in a session that implement the javax.servlet.http.HttpSessionBindingListener interface are notified when the session is preparing to end and become invalidated. This notice enables you to perform post-session processing, including permanently saving the data changes made during the session to a database.
6. End the session. You can end a session:
   - Automatically with the Session Management facility if a session is inactive for a specified time. The administrators provide a way to specify the amount of time after which to invalidate a session.
   - By coding the servlet to call the invalidate() method on the session object.

Example

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class SessionSample  extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {


   // Step 1: Get the Session object

      boolean create = true;
      HttpSession session = request.getSession(create);

   // Step 2: Get the session data value


      Integer ival = (Integer)
      session.getAttribute ("sessiontest.counter");
      if (ival == null) ival = new Integer (1);
      else ival = new Integer (ival.intValue () + 1);
      session.setAttribute ("sessiontest.counter", ival);

   // Step 3: Output the page

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head><title>Session Tracking Test</title></head>");
      out.println("<body>");
      out.println("<h1>Session Tracking Test</h1>");
      out.println ("You have hit this page " + ival + " times" + "<br>");
      out.println ("Your " + request.getHeader("Cookie"));
      out.println("</body></html>");
   }
}
```

Assembling so that session data can be shared

By default, the Session Management facility supports session scoping by Web module in accordance with the Servlet 2.3 API specification. Only servlets in the same Web module can access the data associated with a particular session. However, you can use the IBMApplicationSession object or the IBM extension, shared session context, to share data outside of the Web module scope.

About this task

Note: The IBMApplicationSession object is the recommended method for sharing session attributes. The IBM extension, shared session context, is deprecated.

The IBMApplicationSession object is a parent session object that can be retrieved by a Web module's session and can share session attributes across all of the Web modules in a business-level application. The default scope of the business-level application is the enterprise application. The shared session context option extends the scope of the session attributes as well. Using the shared session context extension, there is only one session object for the entire business-level application or for the default enterprise application.

If you are using a shared session for a business-level application, then the class files for all objects placed in the session must exist in an isolated shared library and be common among all applications.

The benefit to using the IBMApplicationSession method is that each Web module can maintain its own session as well as have a reference to the shared session.

If you're migrating an application from a previous version of the product, the IBMApplicationSession method requires a change to the application logic of the application.

Restriction: To use a shared session, you must install all applications within a business-level application on a given server. You cannot split up the enterprise application by servers. For example, you cannot use this option when one enterprise application in "BLA1" is installed on one server and a second enterprise application also in "BLA1" is installed on a different server. In such split installations, applications might share session attributes across Web modules using distributed sessions, but session data integrity is lost when concurrent access to a session is made in different Web modules. It also severely restricts use of some Session Management features, like TIME_BASED_WRITES.

For enterprise applications on which this shared session context extension is enabled, the Session Management configuration on the Web module inside the enterprise application is ignored. Then Session Management configuration defined on enterprise application is used if Session Management is overwritten at the enterprise application level. Otherwise, the Session Management configuration on the Web container is used. If using multiple enterprise applications within a business-level application, the session management configuration must be common among all applications and Web modules within this business-level application.

Note: HttpSession listeners defined in all the Web modules inside the business-level application or enterprise application are invoked for session events. The order of listener invocation is not guaranteed.

Do the following to share session data across the business-level application.

1. Do the following to share session data using the IBMApplicationSession object within the application code.
   a. Retrieve the session object
      ```
      HttpSession session = request.getSession();
      ```
   b. Cast this object to an IBMSession object and call the getIBMApplicationSession method.
      ```
      IBMApplicationSession appSession = ((IBMSession)session).getIBMApplicationSession();
      ```

c. Use the appSession like a normal session object.

2. Do the following to share session data using the Shared session context extension.
   a. Start an assembly tool.
   b. In the assembly tool, right-click the application (EAR file) that you want to share and click **Open With > Deployment Descriptor Editor**.
   c. In the application deployment descriptor editor of the assembly tool, select **Shared session context** under **WebSphere Extensions**. Make sure the class definition of attributes put into session are available to all Web modules in the enterprise application. The shared session context does not fully meet the requirements of the specifications.
   d. Save the application (EAR) file. In the assembly tool, after you close the application deployment descriptor editor, confirm that you want to save changes made to the application.

Developing servlets with WebSphere Application Server extensions

About this task

Several WebSphere Application Server extensions are provided for enhancing your servlets. This task provides a summary of the extensions that you can utilize.

1. Review the supported specifications.

   Create Java components, referring to the Servlet specifications from Sun Microsystems.

   The application server includes its own packages that extend and add to the Java Servlet Application Programming Interface (API). These extensions and additions make it easier to manage session states, create personalized Web pages, generate better servlet error reports, and access databases. Locate the API documentation for the application server APIs in the `install_root\web\apidocs` directory for the default installation. All of the public Application Server APIs are located in the `com.ibm.websphere` packages, however, com.ibm.websphere.servlet package is specific to the product servlet APIs.
2. Use your favorite integrated development environment (IDE), or a text editor, to develop or migrate code artifacts that meet the specifications.
3. Test the code artifacts.

What to do next

Assemble your code artifacts into a Web module using assembly tools as a prerequisite to deploying the code to the application server.

Configuring page list servlet client configurations

You can define PageListServlet configuration information in the IBM Web Extensions file. The IBM Web Extensions file is created and stored in the Web applications archive (WAR) file by an assembly tool.

About this task

Note: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 7.0 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes. Starting from the Servlet 2.3 specification, javax.servlet.filter classes you can intercept requests and examine responses. You can also use javax.servlet.filter classes to achieve chaining functionality, as well as embellishing or truncating responses.

To configure and implement page lists:

1. To configure page list information, use the Add Markup Language entry dialog of an assembly tool. On the **Servlets** tab of a Web deployment descriptor editor, select a servlet and click **Add** under **WebSphere Extensions**.

2. Add the `callPage()` method to your servlet to invoke a JavaServer Page (JSP) file in response to a client request.

   The PageListServlet has a `callPage()` method that invokes a JSP file in response to the HTTP request for a page in a page list. The `callPage()` method can be invoked in one of the following ways:

   - `callPage(String pageName, HttpServletRequest request, HttpServletResponse response)`

     where the method arguments are:

     pageName
     A page name defined in the PageListServlet configuration

     request
     The HttpServletRequest object

     response
     The HttpServletResponse object

   - `callPage(String mlName, String pageName, HttpServletRequest request, HttpServletResponse response)`

     where the method arguments are:

     mlName A markup language type

     pageName
     A page name defined in the PageListServlet configuration

     request
     The HttpServletRequest object

     response
     The HttpServletResponse object

3. Use the PageList Servlet client type detection support to determine the markup language type a calling client requires for the response.

Extending PageListServlet

The following example shows how a servlet extends the PageListServlet class and determines the markup-language type required by the client. The servlet then uses the callPage method to call an appropriate JavaServer Pages (JSP) file. In this example, the JSP file that provides the correct markup-language for the response is *Hello.page*.

```
public class HelloPervasiveServlet extends PageListServlet implements Serializable
{
     /*
     * doGet -- Process incoming HTTP GET requests
     */
    public  void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
       // This is the name of the page to be called:
       String pageName = "Hello.page";

       // First check if the servlet was invoked with a queryString that contains
    // a markup-language value.
       // For example, if this is how the  servlet is invoked:
       //  http://localhost/servlets/HeloPervasive?mlname=VXML
       //  then use the following method:
       String mlname= getMLNameFromRequest(request);

        // If no markup language type is provided in the queryString,
        // then try to determine the client
        // Type from the request, and use the markup-language name configured in
        // the client_types.xml file.
        if (mlName == null)
        {
          mlName = getMLTypeFromRequest(request);
         }
         try
         {
```

```
            // Serve the request page.
            callPage(mlName, pageName, request, response);
           }
           catch (Exception e)
           {
             handleError(mlName, request, response, e);
            }
       }
}
```

Page lists

Page lists allow you to avoid hard-coding Uniform Resource Locators (URLs) in servlets and JSP files. A page list specifies the location where a request is to be forwarded, but automatically customizes that location depending on the MIME type of the servlet. Use these properties to specify a markup language and an associated MIME type. For the given MIME type, you also specify a set of pages to invoke.

Note: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 7.0 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes. Starting from the Servlet 2.3 specification, javax.servlet.filter classes you can intercept requests and examine responses. You can also use javax.servlet.filter classes to achieve chaining functionality, as well as embellishing or truncating responses.

The following list of classes are deprecated:

- com.ibm.servlet.ClientList
- com.ibm.servlet.ClientListElement
- com.ibm.servlet.MLNotFoundException
- com.ibm.servlet.PageListServlet
- com.ibm.servlet.PageNotFoundException

WebSphere Application Server supplies the PageListServlet servlet, which you can use to call a JavaServer Pages (JSP) file by name based on the configuration data in the `client_types.xml` file. This file maps a JSP file to a Uniform Resource Identifier (URI). When the URI is invoked, it specifies another JSP file in a Web module. This support allows you to access multiple URLs without hard-coding them in your servlets.

You can also logically group page lists according to the markup language type, such as, Hypertext Markup Language (HTML) or Wireless Markup Language (WML). This allows applications that use servlets to extend the PageListServlet servlet, to call JSP files which return the proper markup-language type for the client request. For example, a request that originates from a PDA device requires WML data. The application server sends the request to a servlet that extends the PageListServlet servlet, and the servlet calls a JSP file that returns a WML response.

Client type detection support

In addition to providing the page list mapping capability, the PageListServlet also provides *Client Type Detection* support. A servlet determines the markup language type that a calling client needs in the response, using the configuration information in the `client_types.xml` file.

Client type detection support allows a servlet, extending the PageListServlet, to call an appropriate JavaServer Pages (JSP) file. The servlet invokes the `callPage` method, which calls a JSP file based on the markup-language type of the request.

The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 7.0 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes.

client_types.xml

The client_types.xml file provides client type detection support for servlets extending PageListServlet. Using the configuration data in the client_types.xml file, servlets can determine the language type that calling clients require for the response.

Note: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 7.0 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes.

The client type detection support allows servlets to call appropriate JavaServer Pages (JSP) files with the callPage method. Servlets select JSP files based on the markup-language type of the request.

Servlets must use the following version of the callPage method to determine the markup language type required by the client:

```
callPage(String mlName, String pageName, HttpServletRequest request,
                  HttpServletResponse response)
```

where the arguments are:
- mlName - a markup language type
- pageName - a page name defined in the PageListServlet configuration
- request - the HttpServletRequest object
- response - the HttpServletResponse object

Review the Extending the PageListServlet code example in Extending the PageListServlet to see how the callPage method is invoked by a servlet.

In the example, the client type detection method, getMLTypeFromRequest(HttpServletRequestrequest), provided by the PageListServlet, inspects the HttpServletRequest object request headers, and searches for a match in the client_types.xml file.

The client type detection method does the following:
- Uses the input HttpServletRequest and the client_types.xml file, to check for a matching HTTP request name and value.
- Returns the markup-language value configured for the <client-type> element, if a match is found.
- If multiple matches are found, this method returns the markup-language for the first <client-type> element for which a match is found.
- If no match is found, returns the value of the markup-language for the default page defined in the PageListServlet configuration.

Location

The client_types.xml file is located in the *install_root*/properties directory.

Usage notes

- Is this file read-only?

  No
- Is this file updated by a product component?

  No
- If so, what triggers its update?

  This file is created and updated manually by users.
- How and when are the contents of this file used?

  Servlets that extending the PageListServlet servlet use this file to determine the language type that calling clients require for the response.

Sample file entry

```
<?xml version="1.0" >
<!DOCTYPE clients [
<!ELEMENT client-type (description, markup-language,request-header+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT markup-language (#PCDATA)>
<!ELEMENT request-header (name, value)>
<!ELEMENT clients (client-type+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>]>
<clients>
      <client-type>
      <description>IBM Speech Client</description>
      <markup-language>VXML</markup-language>
      <request-header>
           <name>user-agent</name>
           <value>IBM VoiceXML pre-release version 000303</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vxml</value>
      </request-header>
      </client-type>
      <client-type>
           <description>WML Browser</description>
           <markup-language>WML</markup-language>
      <request-header>
           <name>accept</name>
           <value>text/x-wap.wml</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vnd.wap.xml</value>
      </request-header>
    </client-type>
</clients>
```

autoRequestEncoding and autoResponseEncoding

Starting with WebSphere Application Server Version 5, the Web container no longer automatically sets request and response encodings, and response content types. Programmers are expected to set these values using available methods in the Servlet 2.3 Specification or later. If programmers choose not to use the character encoding methods, they can specify the `autoRequestEncoding` and `autoResponseEncoding` extensions, which enable the application server to set the encoding values and content type.

The values of the `autoRequestEncoding` and `autoResponseEncoding` extensions are either `true` or `false`. The default value for both extensions is `false`. If the value is `false` for both `autoRequestEncoding` and `autoResponseEncoding`, then the request and response character encoding is set to the Servlet 2.3 Specification default, which is ISO-8859-1. Also, if the value is set to `false` for a response, the Web container cannot set a response content type. Different character encodings are possible if the client defines character encoding in the request header, or if the code includes the `setCharacterEncoding(String encoding)` method.

If the `autoRequestEncoding` value is set to `true`, and the client did not specify character encoding in the request header, and the code does not include the `setCharacterEncoding(String encoding)` method, the Web container tries to determine the correct character encoding for the request parameters and data.

Use an assembly tool to change the default values for the `autoRequestEncoding` and `autoResponseEncoding` extensions.

The Web container performs each step in the following list until a match is found:

- Looks at the character set (charset) in the *Content-Type* header.

- Attempts to map the servers locale to a character set using defined properties.
- Attempts to use the DEFAULT_CLIENT_ENCODING system property, if one is set.
- Uses the ISO-8859-1 character encoding as the default.

If the autoResponsetEncoding value is set to true, and the client did not specify character encoding in the request header, and the code does not include the setCharacterEncoding(String encoding) method, the Web container does the following:
- Attempts to determine the response content type and character encoding from information in the request header.
- Uses the ISO-8859-1 character encoding as the default.

Initial parameters for servlets settings

Use this page to specify initial parameters that are passed to the init method of Web module servlet filters. You can specify initial parameter values for servlets in Web modules during or after installation of an application onto a WebSphere Application Server deployment target. The <param-value> values specified in <init-param> statements in the web.xml file of Web modules are used by default.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Init parameters for servlets**. This page is the same as the Init parameters for servlets in each Web module panel on the application installation and update wizards.

Module

Specifies the name of a module in the application that you are installing or that you are viewing after installation.

URI

Specifies the location of the module relative to the root of the application (EAR file).

Servlet

Specifies a unique name for the servlet within the application.

A *servlet* is a Java program that uses the Java Servlet Application Programming Interface (API). You must package servlets in a Web archive (WAR) file or Web module for deployment to an application server. Servlets run on a Java-enabled Web server and extend the capabilities of a Web server, similar to the way applets run on a browser and extend the capabilities of a browser.

Name

Specifies the name of the initial parameter passed to the init method of the Web module servlet filter.

The following example servlet filter statement in a web.xml file specifies an initial parameter name of attribute:

```
<init-param>
   <param-name>attribute</param-name>
   <param-value>tests.Filter.DoFilter_Filter.SERVLET_MAPPED</param-value>
</init-param>
```

Value

Specifies the value assigned to an initial parameter passed to the init method of the Web module servlet filter.

The following example servlet filter statement in a web.xml file specifies an initial parameter value of tests.Filter.DoFilter_Filter.SERVLET_MAPPED for the init parameter attribute:

```
<init-param>
   <param-name>attribute</param-name>
   <param-value>tests.Filter.DoFilter_Filter.SERVLET_MAPPED</param-value>
</init-param>
```

Description
Specifies information on the initial parameter.

Servlet filtering
Servlet filtering provides a new type of object called a *filter* that can transform a request or modify a response.

You can chain filters together so that a group of filters can act on the input and output of a specified resource or group of resources.

Filters typically include logging filters, image conversion filters, encryption filters, and Multipurpose Internet Mail Extensions (MIME) type filters (functionally equivalent to the servlet chaining). Although filters are not servlets, their life cycle is very similar.

Filters are handled in the following manner:

1. The Web container determines whether it needs to construct a `FilterChain` containing the `LoggingFilter` for the requested resource.

   The `FilterChain` begins with the invocation of the `LoggingFilter` and ends with the invocation of the requested resource.
2. If other filters need to go in the chain, the Web container places them after the`LoggingFilter` and before the requested resource.
3. The Web container then instantiates and initializes the `LoggingFilter` (if it was not done previously) and invokes its `doFilter(FilterConfig)` method to start the chain.
4. The `LoggingFilter` preprocesses the request and response objects and then invokes the filter chain `doFilter(ServletRequest, ServletResponse)` method.

   This method passes the processing to the next resource in the chain, the requested resource.
5. Upon return from the filter chain `doFilter(ServletRequest, ServletResponse)` method, the `LoggingFilter` performs post-processing on the request and response object before sending the response back to the client.

Note: Java Specification 2.4 allows you to define a new `<dispatcher>` element in the deployment descriptor with possible values such as REQUEST, FORWARD, INCLUDE, ERROR, instead of invoking filters with RequestDispatcher.

For example:

```
<filter-mapping>
<filter-name>Logging Filter</filter-name>
<url-pattern>/products/*</url-pattern>
<dispatcher>FORWARD</dispatcher>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

This indicates that the filter should be applied to requests directly from the client as well as forward requests. Adding the INCLUDE and ERROR values also indicates that the filter should additionally be applied for included requests and `<error-page>` requests. If you do not specify any <dispatcher> elements, then the default is REQUEST.

Filter, FilterChain, FilterConfig classes for servlet filtering

The following interfaces are defined as part of the javax.servlet package:
- Filter interface - methods: doFilter, getFilterConfig, setFilterConfig
- FilterChain interface - methods: doFilter
- FilterConfig interface - methods: getFilterName, getInitParameter, getInitParameterNames, getServletContext

The following classes are defined as part of the javax.servlet.http package:
- HttpServletRequestWrapper - methods: See the Servlet 2.4 Specification
- HttpServletResponseWrapper - methods: See the Servlet 2.4 Specification

Application life cycle listeners and events

With application life cycle listeners and events, which are now part of the Servlet API, you can notify interested listeners when servlet contexts and sessions change. For example, you can notify users when attributes change and if sessions or servlet contexts are created or destroyed.

The life cycle listeners give the application developer greater control over interactions with ServletContext and HttpSession objects. Servlet context listeners manage resources at an application level. Session listeners manage resources that are associated with a series of requests from a single client. Listeners are available for life cycle events and for attribute modification events. The listener developer creates a class that implements the javax listener interface, corresponding to the listener functionality that you want.

At application startup time, the container uses *introspection* to create an instance of your listener class and registers it with the appropriate event generator.

When a servlet context is created, the contextInitialized method of your listener class is invoked, which creates the database connection for the servlets in your application to use if this context is for your application. All servlet context listeners are notified of context initialization before any servlet in the Web application is initialized.

When the servlet context is destroyed, your contextDestroyed method is invoked, which releases the database connection, if this context is for your application. You must destroy all servlets before any servlet context listeners are notified of context destruction.

Notifications to session listeners precede notifications to context listeners.

Listener classes for servlet context and session changes

The following methods are defined as part of the javax.servlet.ServletContextListener interface:
- void contextInitialized(ServletContextEvent)

  Notification that the Web application is ready to process requests. Place code in this method to see if the created context is for your Web application and if it is, allocate a database connection and store the connection in the servlet context.
- void contextDestroyed(ServletContextEvent)

  Notification that the servlet context is about to shut down. Place code in this method to see if the created context is for your Web application and if it is, close the database connection stored in the servlet context.

The following methods are defined as part of the `javax.servlet.ServletRequestListener` interface:
- `public void requestInitialized(ServletRequestEvent re)`
  - Notification that the request is about to come into scope

    A request is defined as coming into scope when it is about to enter the first filter in the filter chain that processes the request.
- `public void requestDestroyed(ServletRequestEvent re)`
  - Notification that the request is about to go out of scope

    A request is defined as going out of scope when it exits the last filter in its filter chain.

The following listener interfaces are defined as part of the javax.servlet package:
- ServletContextListener
- ServletContextAttributeListener

The following filter interface is defined as part of the javax.servlet package:
- FilterChain interface - methods: doFilter()

The following event classes are defined as part of the javax.servlet package:
- ServletContextEvent
- ServletContextAttributeEvent

The following interfaces are defined as part of the javax.servlet.http package:
- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionActivationListener

The following event class is defined as part of the javax.servlet.http package:
- HttpSessionEvent

Example: Creating a servlet context listener with com.ibm.websphere.DBConnectionListener.java

The following example shows how to create a servlet context listener:

```
package com.ibm.websphere;

import java.io.*;
import javax.servlet.*;

public class DBConnectionListener implements ServletContextListener
{
     // implement the required context init method
    void contextInitialized(ServletContextEvent sce)
    {
    }

     // implement the required context destroy method
    void contextDestroyed(ServletContextEvent sce)
    {
    }
}
```

Configuring JSP engine parameters

About this task

WebSphere Application Server does not support the modification of deployment descriptor extension parameters through the Administrative Console or through administrative scripting.

To add, change or delete JSP engine configuration parameters, complete the following steps:

1. Open the `WEB-INF/ibm-web-ext.xmi` file.

   JSP engine configuration parameters are stored in a web module's configuration directory or in a web modules's binaries directory in the `WEB-INF/ibm-web-ext.xmi` file. Open the `WEB-INF/ibm-web-ext.xmi` file from:

   - The configuration directory, as in the following example: `{WAS_ROOT}/profiles/`*`profilename`*`/config/cells/`*`cellname`*`/applications/`*`enterpriseappname`*`/ deployments/`*`deployedname`*`/`*`webmodulename`*
   - The binaries directory if an application was deployed into WebSphere Application Server with the flag "Use Binary Configuration" set to true. An example of a binaries directory is: `{WAS_ROOT}/profiles/`*`profilename`*`/installedApps/`*`nodename`*`/`*`EnterpriseAppName`*`/`*`WebModuleName`*`/`
2. Edit the `WEB-INF/ibm-web-ext.xmi` file.
   - To add configuration parameters, use the following format: `xmi:id="JSPAttribute_6" name="parametername" value="parametervalue"/>`

- To delete configuration parameters, either delete the line from the file, or enclose the statement with <!-- --> tags.

3. Save the file.
4. Restart the Enterprise Application. It is not necessary to restart the server for parameter changes to take effect. However, some JSP engine configuration parameters affect the Java source code that is generated for a JSP. If such a parameter is changed, then you must retranslate the JSP files in the Web module to regenerate Java source. You can use the batch compiler to retranslate all JSP files in a Web module. The batch compiler uses the JSP engine configuration parameters that you have set in the `ibm-web-ext.xmi` file, unless you specifically override them. The topic JSP engine configuration parameters″ identifies the parameters that affect the generated Java source.

Example

The following is a sample of the `WEB-INF/ibm-web-ext.xmi` file. The lines in bold text are JSP engine configuration parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<webappext:WebAppExtension xmi:version="2.0" xmlns:xmi=http://www.omg.org/XMI
   xmlns:webappext="webappext.xmi" xmlns:webapplication="webapplication.xmi" xmi:id="WebAppExtension_1"
   reloadInterval="9" reloadingEnabled="true" defaultErrorPage="error.jsp" additionalClassPath=""
   fileServingEnabled="true" directoryBrowsingEnabled="false" serveServletsByClassnameEnabled="true"
   autoRequestEncoding="true" autoResponseEncoding="false"
   <webApp href="WEB-INF/web.xml#WebApp_1"/>
   <jspAttributes xmi:id="JSPAttribute_1" name="useThreadTagPool" value="true"/>
   <jspAttributes xmi:id="JSPAttribute_2" name="verbose" value="false"/>
   <jspAttributes xmi:id="JSPAttribute_3" name="deprecation" value="false"/>
   <jspAttributes xmi:id="JSPAttribute_4" name="reloadEnabled" value="true"/>
   <jspAttributes xmi:id="JSPAttribute_5" name="reloadInterval" value="5"/>
   <jspAttributes xmi:id="JSPAttribute_6" name="keepgenerated" value="true"/>
   <!--<jspAttributes xmi:id="JSPAttribute_7" name="trackDependencies" value="true"/> -->

</webappext:WebAppExtension>
```

Note: The integer **n** in **JSPattribute_n** has to be unique within the file.

JSP engine

The WebSphere Application Server JavaServer Pages (JSP) engine is the implementation of the JavaServer Pages Specification.

WebSphere Application Server Version 7.0 supports the JSP 2.1 specification.

The JSP engine

- Validates JSP source, both classic and XML styles
- Translates JSP source to Java classes
- Compiles Java classes, reporting any errors
- Generates Java classes for any tag files that are used by the JSP
- Interfaces with the Web container to load JSP class files
- Supports JSP batch compilation, JSP compilation during application installation, and JSP compilation during the build process of customer applications, through an Ant task.
- Loads class files, and manage life-cycle (reloading, unloading as necessary)
- Supports debugging of JavaServer Pages files through support for JSR 45 (Debugging Support for Other Languages)

JSP engine configuration parameters

In WebSphere Application Server, you can configure the JavaServer Pages (JSP) engine configuration parameters for optimal performance in a production server environment and for the needs of developers in a development environment.

The JSP engine parameters are case sensitive. If the value specified for a parameter is comprised of two or more words separated by spaces, you must add quotation marks around the value. Some parameters affect the Java source that is generated for a JSP or tag file. These parameters are identified by the statement ″This parameter requires regeneration of Java source.″ This statement indicates that if the configuration parameter is modified, the new value for the parameter does not have any effect until the JSP files are retranslated and the Java sources are recompiled.

compileWithAssert

Specifies whether the generated Java classes should contain support for the Developer Kit, Java Technology Edition 1.4 Assertion facility. The effect of setting this parameter to true is that the –source 1.4 option is passed to the Java compiler. The default for this parameter is `false`. This parameter requires regeneration of Java source.

classdebuginfo

Indicates whether the compiler includes debugging information in the generated class file. When you set this parameter to true, the –g option is passed to the Java compiler. The default for this parameter is `false`. This parameter requires regeneration of Java source.

convertAttrValueToString

Specifies whether to convert start and end attributes of the repeat tag to strings before they are used. The default for this parameter is `false`. This parameter requires regeneration of Java source.

deprecation

Specifies whether the compiler generates deprecation warnings when compiling the generated Java source. When you set this parameter to true, the -deprecation option is passed to the Java compiler. The default for this parameter is `false`. This parameter requires regeneration of Java source.

disableElCache

Set the com.ibm.wsspi.jsp.disableElCache Web container custom property to `true` to disable the commons-el expression cache if you are experiencing out of memory conditions because the hash maps are held by the expression evaluator. The default for this parameter is `false`.

disableJspRuntimeCompilation

If this option is set to true, the JSP engine at runtime does not translate and compile JSP files; the JSP engine loads only precompiled class files. JSP source files do not need to be present in order to load class files. When this option is set to true, you can install an application without JSP source, but the application must have precompiled class files. There is a Web container custom property with the same name that is used to determine the behavior of all Web modules installed in a server. If both the Web container custom property and the JSP engine option are set, the JSP engine option takes precedence. The default for this parameter is `false`.

evalQuotedAndEscapedExpression

Set this option to true to handle escape characters and quotations properly when determining whether to evaluate an expression.

During the translation phase of a JSP compile, expressions are evaluated by the JSP engine. Characters, such as the escape character (\) or nested quotations, single or double, causes the JSP file translation to fail. For example, when you use functions that contain an expression such as

```
<input type="text" value="${fn:substring('1234567', 0,4)}"/>
```

Because of the double quote directly before the fn:substring statement, the JSP file fails to compile because the translator did not add the function mapper to the generated Java class. Also, if a dollar sign ($) was escaped using the backslash (\$), the translator still attempts to evaluate the expression instead of treating it as a literal string. To handle escape characters and quotations properly, you must set evalQuotedAndEscapedExpression to `true` in the ibm-web-ext.xmi file of the failing application.. An example entry follows:

```
<jspAttributes xmi:id="JSPAttribute_1"
name="evalQuotedAndEscapedExpression" value="true"/>
```

To apply this behavior globally across all Web applications, you can set the com.ibm.wsspi.jsp.evalQuotedAndEscapedExpression Web container custom property to true.

extendedDocumentRoot

To share a JSP file resource across Web application archives, specify a comma delimited list of directories or Java Archive (JAR) files or both as search paths to be used if the requested resource is not located in the public document tree of the Web application archive. If the request is a valid partial request for a welcome file, a 404 error is returned. If the JSP file is located inside a JAR file and reloadEnabled is true, the timestamp of the JAR file is used for isOutDated checks for recompile purposes. The default for this parameter is null.

ieClassID

Indicates the Java plug-in COM class ID for Internet Explorer.

The `<jsp:plugin>` tags use this value. The default classid is `clsid:8AD9C840-044E-11D1-B3E9-00805F499D93`

javaEncoding

Specifies the encoding that is used when the `.java` file is generated, and when it is compiled by the Java compiler. Set this parameter when the page encoding of your JSP pages is not UTF-8 compatible. When javaEncoding is set, the encoding is passed to the Java compiler through the -encoding argument. Note that encoding is not supported by Jikes. The default is UTF-8. This parameter requires regeneration of Java source.

jdkSourceLevel

This is a new JSP engine parameter which was introduced in WebSphere Application Server version 6.1 to support JDK 5. This parameter should be used instead of the compileWithAssert parameter, although compile WithAssert still works in version 6.1.

The default value for this parameter is 13. This parameter requires regeneration of Java source. The following are jdkSourceLevel paramater values:

- **13 (default)** - This value will disable all new language features of JDK 1.4 and JDK 5.0.
- **14** - This value will enable the use of the assertion facility and will disable all new language features of JDK 5.0.
- **15** - This value will enable the use of the assertion facility and all new language features of JDK 5.0.

jsp.file.extensions

For JSP files with extensions other than the four standard extensions, *.jsp, *.jspx, *.jsw, and *.jsv, you can configure this the extensions using this parameter. These extensions are added to the standard extensions.

The preferred method for doing this is to create a `<jsp-property-group>`in web.xml, and add a `<url-pattern>` tag for each extension.

The JSP engine can handle a list of file extensions that is separated by a colon or semi-colon. For example, *.ext1;*.ext2:*.extn

keepgenerated
Indicates that the Java files generated by the JSP compiler during the translation phase of the processing are retained. The default for this parameter is `false`. This parameter requires regeneration of Java source.

keepGeneratedclassfiles
Indicates that the class files generated by the JSP compiler during the translation phase of the processing are retained. The default for this parameter is `true`. This parameter requires regeneration of Java source.

modifyPageContextVariable
During the translation phase of a tag file that is compiled, the JSP container implicitly uses the pageContext variable for the PageContext object. The use of the pageContext variable as an implicit variable name in tag files does not comply with the JSP Specification. If compilation errors occur for applications that use a local pageContext variable in their tag file, set the modifyPageContextVariable attribute to`true` to remove the use of the pageContext variable name in the generated Java code for tag files.

recompileJspOnRestart
Determines whether a JSP file is retranslated and recompiled after application startup for the first time the file is requested. If recompileJspOnRestart is false, a JSP file is still compiled, if necessary, on the first request to that JSP file unless the parameter disableJspRuntimeCompilation is true. The default for this parameter is false.

reloadEnabled
Determines whether or not a JSP file is translated and compiled at runtime if the JSP file or its dependencies (see trackDependencies) are modified.

If reloadEnabled is false, a JSP file is still compiled, if necessary, on the first request to it unless the parameter disableJspRuntimeCompilation is true. The default for this parameter is `false`.

If this JSP engine parameter is not specified, the equivalent Web container parameter for Web module class reloading is used. However, for an application whose deployment descriptor is at the Servlet 2.2 level, the default is true. This is done for the support of applications being migrated from WebSphere Application Server Version 4.x.

reloadInterval
If reloading is enabled, reloadInterval determines the delay between checks to see if a JSP file is outdated.

For example, if reloadInterval is 5, the JSP engine checks to see if a JSP file is outdated only when the last such check was done more than 5 seconds prior to the current request for the JSP file. The larger the reloadInterval, the less frequently the JSP engine checks for the need to reload a JSP file. If this JSP engine parameter is not specified, the equivalent Web container parameter for Web module class reloading is used. However, for an application whose deployment descriptor is at the Servlet 2.2 level, the default is 5 seconds. This is done for the support of applications being migrated from WebSphere Application Server Version 4.x.

scratchdir
Specifies the directory where the generated class files are created.

The system property com.ibm.websphere.servlet.temp.dir is used to set the scratchdir option on a server-wide basis. The JSP engine scratchdir parameter takes precedence over the system property com.ibm.websphere.servlet.temp.dir. The default for this parameter is *profile_root*/`temp`. This parameter requires regeneration of Java source.

trackDependencies
If reloading is enabled, trackDependencies determines whether the JSP engine tracks modifications to the requested JavaServer Pages files dependencies as well as to the JSP file itself.

The dependencies tracked by the JSP engine are :

1. files statically included in the JSP file
2. tag files referenced in the JSP file (excluding tag files that are in JARs)
3. TLD files referenced in the JSP file (excluding TLDs that are in JARs)

The default is `false`.

useFullPackageNames
If useFullPackageNames is true, the JSP engine generates and loads JSP classes using full package names.

The default is to generate all JSP classes in the same package. (For more information, see Packages and directories for generated .java and .class files). The JSP engine's class loader knows how to load JSP classes when they are all in the same package.

The default method of generating all JSP classes in the same package has the benefit of generating smaller file-system paths. Full package names has the benefit of enabling the configuration of precompiled JSP class files as servlets in the `web.xml` file without the use of the jsp-file attribute, resulting in a single class loader, the Web application's class loader, that is used to load all such JSP classes. Similarly, when the JSP engine's configuration attributes useFullPackageNames and disableJspRuntimeCompilation are both true, a single class loader is used to load all JSP classes, even if the JSP files are not configured as servlets in the `web.xml` file.

When useFullPackageNames is set to true, the batch compiler generates a file called `generated_web.xml` in the Web module's `WEB-INF` directory. This file contains servlet configuration information for each JSP file that was successfully translated and compiled. The information can optionally be copied into the Web module's `web.xml` file so that the JSP files are loaded as servlets by the Web container. Note that if a JSP file is configured as a servlet in this way, no reloading of the JSP file is done at runtime if the JSP file is modified. This is because the JSP file is treated as a regular servlet and requests for it do not pass through the JSP engine. This parameter requires regeneration of Java source.

useImplicitTagLibs
The JSP engine implicitly recognizes tsx and jsx as tag library prefixes for tag libraries supplied by the JSP engine. If tsx or jsx are used as prefixes for a customer's tag library, the customer's tag library overrides the implicit tag library. However, the implicit tag library is still cached by the JSP engine. Explicitly setting this parameter to false tells the engine not to cache the implicit tag library, and save resources. The default for this parameter is `true`.

useInMemory
Specifies that the JSP engine translate and compile Java code in the system memory.

When this option is not set, the JSP engine must perform the following steps:

1. Write the translated Java file to the file system
2. Load the Java file from the file system
3. Compile the code into a class file
4. Write the class to the file system
5. Load the class file into a classloader.

Note: No .class file or .java file will be written to the system disk. For debugging or creating a JAR file from precompiled JSP code, you will need to disable this option.

useJikes
Specifies whether Jikes is used for compiling Java sources.

NOTE: Jikes is not shipped with WebSphere Application Server. The default for this parameter is `false`. This parameter requires regeneration of Java source.

usePageTagPool
*Enables or disables the reuse of custom tag handlers on an individual JavaServer Pages basis. The default for this parameter is `false`. This parameter requires regeneration of Java source.

useThreadTagPool
When thread-level tag handler pooling is used, tag handlers may be reused among separate occurrences of a custom action across all JSP pages in a single Web module across separate requests. The default for this parameter is `false`. This parameter requires regeneration of Java source.

Enabling custom tag handler reuse might reveal problems in the tag handler code with regard to the tag's ability to be reused. A custom tag handler should always do two things:

- The release method of the tag handler should reset its state and release any private resources that it might have used. The JSP engine ensures the release method is called before the tag handler is garbage collected.
- In the doEndTag method, all instance states associated with this instance must be reset.

verbose
Indicates that the compiler generates verbose output when compiling the generated Java source code. The effect of setting this parameter to true is that the -verbose option is passed to the Java compiler. The default for this parameter is `false`. This parameter requires regeneration of Java source.

JavaServer Pages troubleshooting tips
Use this tips to troubleshoot problems with JavaServer Pages.

JavaServer Pages source code shown by the Web server

If you share the document root of the WebSphere Application Server with the Web server document root, a security exposure can result as the Web server might display the JavaServer Pages (JSP) source file as plain text.

Problem

You can use the WebSphere Web server plug-in set of rules to determine whether a given request will be handled by the WebSphere Application Server. When an incoming request fails to match those rules, the Web server plug-in returns control to the Web server so that the Web server can fulfill the request. In this case, the unknown host header causes the Web server plug-in to return control to the Web server because the rules do not indicate that the WebSphere Application Server should handle it. Therefore, the Web server looks for the request in the Web server document root. Since the JSP source file is stored in the document root of the Web server, the Web server finds the file and displays it as plain text.

Suggested solution

Move the WebSphere Application Server JSP source file outside of the Web server document root. Then, when this request comes in with the unknown host header, the plug-in returns control to the Web server and the JSP source file is not found in the document root. Therefore, the Web server returns a `404 File Not Found` error rather than the JSP source file.

Problems displaying double-byte character set (DBCS) characters when using the @include directive

JavaServer Pages files that use the @include directive might experience problems when displaying double-byte character set (DBCS) characters. Some applications that are migrated to WebSphere Application Server Version 6.0 and above might need to be modified to comply with the JSP 2.0 specification as a result of backwards compatibility issues. The JSP 2.0 specification requires that each statically included resource must set a page encoding or content type because the character encoding for each file is determined separately, even if one file includes another using the include directive.

Problems using the JavaServer Pages (JSP) engine

If you are having difficulty using the JavaServer Pages (JSP) engine, try these steps:

1. Determine whether other resources such as `.html` files or servlets are being requested and displayed correctly. If they are not, the problem probably lies at a deeper level, such as with the HTTP server.
2. If other resources are being displayed correctly, determine whether the JSP processor has started normally:
   - Browse the logs of the server hosting the JSP files you are trying to access. The following messages indicate that the JSP processor has started normally:

     ```
     Extension Processor [class com.ibm.ws.jsp.webcontainerext.JSPExtensionProcessor]
     was initialized successfully.
     Extension Processor [class com.ibm.ws.jsp.webcontainerext.JSPExtensionProcessor]
     has been associated with patterns [*.jsp *.jspx *.jsw *.jsv ].
     ```

     If the JSP processor fails to load, you will see a message such as

     ```
     No Extension Processor found for handling JSPs.
     JSP Processor not defined. Skipping : jspfilename.
     ```

     in the server log files.
3. If the JSP engine has started normally, the problem may be with the JSP file itself.
   - The JSP may have invalid JSP syntax and could not be processed by the JSP Processor. Examine the server log files of the target application for invalid JSP directive syntax messages. Errors similar to the following in a browser indicate this kind of problem:

     ```
     Message: /filename.jsp(2,1)JSPG0076E: Missing required attribute page for jsp
                 element jsp:include
     ```

     This example indicates that line 2, column 1 of the named JavaServer Pages file is missing a mandatory attribute for the jsp:include action. Similar messages are displayed for other syntax errors.
   - Examine the target application server's `SystemErr.log` files for problems with invalid Java syntax. Errors similar to **`Message: Unable to compile class for JSP`** in a browser indicate this kind of problem.

     The error message output from the Javac compiler will be found in the `SystemErr.log`. It might look like:

     ```
     JSPG0091E: An error occurred at line: 2 in the file: /myJsp.jsp
     JSPG0093E: Generated servlet error: c:\WASROOT\temp\ ...
     test.war\_myJsp.java:16: myInt is already defined in com.ibm.ws.jsp20._myJsp
     int myInt = 122; String myString = "number is 122"; static int myStaticInt=22;
     int myInt=121;
              ^  1 error
     ```

     Correct the error in the JSP file and retry the file.
   - Examine the target application server's server log files files for problems with invalid Java syntax. Errors similar to **`Message: Unable to compile class for JSP`** in a browser indicate this kind of problem.

     The error message output from the Javac compiler will be found in the server log files. It might look like:

```
JSPG0091E: An error occurred at line: 2 in the file: /myJsp.jsp
JSPG0093E: Generated servlet error: c:\WASROOT\temp\ ...
test.war\_myJsp.java:16: myInt is already defined in com.ibm.ws.jsp20._myJsp
int myInt = 122; String myString = "number is 122"; static int myStaticInt=22;
int myInt=121;
         ^  1 error
```

Correct the error in the JSP file and retry the file.

JavaServer Pages fail to compile when using precompile

Symptom

JavaServer Pages fail to compile during deployment through the administrative console when precompile is selected.

```
SystemErr R com.ibm.websphere.management.exception.AdminException:
ADMA0021E: Error in compiling jsps - xyz.war (rc=1)
```

Problem

JavaServer Pages fail to compile during deployment through the administrative console when precompile is selected when there is a dependency on another Java archive (JAR) file that is not available on any class path.

Suggested solution

You may use wsadmin scripting to precompile JSP files during enterprise application deployment. However if you want to use the administrative console, then compile all JSP files before packaging the application.

1. Add the dependent JAR to the deployment manager in a cell environment.
   a. Click **System Administration > Deployment manager > Java and Process Management > Process Definition > Java Virtual Machine** in the console navigation.
   b. Add fully qualified dependent JAR in class path field.
   c. Click OK.
   d. Restart deployment manager.

JSPG0089E: Mismatch found between page directive encoding Shift_JIS and xml prolog encoding UTF-8

Symptom

The following error appears:

```
JSP Processing Error

HTTP Error Code: 500

Error Message: /test.jsp(2,1) /test.jsp(2,1) JSPG0089E: Mismatch found between
page directive encoding Shift_JIS and xml prolog encoding UTF-8
```

Problem

The pageEncoding attribute in the jsp:directive.page element is not UTF-8.

Suggested solution

JavaServer Pages must specify a prolog that matches the encoding specified in the page directive. For example,

```
<?xml version="1.0" encoding="Shift_JIS"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
<jsp:directive.page language="java" contentType="text/html";
 charset=Shift_JIS" pageEncoding="Shift_JIS"/>
<jsp:text>XXXXXjsp:text>XXXXX>
</jsp:root>
```

For additional information, see section JSP.4.1, Page Character Encoding, in the JavaServer Pages specification and section 4.3.3 and appendix F.1 of the Extensible Markup Language (XML) specification

If none of these steps solves the problem, check to see if the problem is identified and documented using the links in Diagnosing and fixing problems: Resources for learning. If you do not see a problem that resembles yours, or if the information provided does not solve your problem, contact IBM support for further assistance.

For current information available from IBM Support on known problems and their resolution, see the IBM Support page. The IBM Support page contains documents that can save you time gathering information needed to resolve this problem.

Developing Web applications

Before you begin

Design a Web application and the required components.

About this task

There are two basic approaches to selecting tools for developing Web applications:

- You can use one of the available integrated development environments (IDEs). IDE tools automatically generate significant parts of the servlet and JavaServer Pages (JSP) code, and Hypertext Markup Language (HTML) files. They also contain integrated tools for packaging and testing the Web application components.
- If you decide to develop Web components without an IDE, you need at least an ASCII text editor. You can also use tools available in the Java SE Development Kit 6 and in this product to assemble, test, and deploy the Web application components.

The following steps support the second approach, development without an IDE.

1. If necessary, migrate any pre-existing code to the required version of the servlet and JSP specification.
2. Write and compile the components of the Web application. To access classes that were extended, compile your code using the `-classpath` option on the `javac` compiler. This option allows you to reference the `j2ee.jar` file in the product directory:
   - *<install_root>*`\lib`

   To compile that same servlet on the Windows NT® version of WebSphere Network Deployment, specify:

   ```
   javac -classpath D:\Program Files\WebSphere\DeploymentManager\lib\j2ee.jar MyServlet.java
   ```
3. **(Optional)** Disable JavaServer Pages (JSP) runtime compilation, if necessary.

What to do next

Assemble the application components in one or more Web modules.

JavaServer Faces

JavaServer Faces (JSF) is a user interface framework or application programming interface (API) that eases the development of Java based Web applications. WebSphere Application Server version 7.0 supports JavaServer Faces 1.2 at a runtime level, therefore using JSF reduces the size of the Web application since runtime binaries no longer need to be included in your Web application.

The JSF runtime also :

- Makes it easy to construct a user interface from a set of reusable user interface components
- Simplifies migration of application data to and from the user interface
- Helps manage user interface state across server requests
- Provides a simple model for wiring client-generated events to server-side application code
- Supports custom user interface components to be easily build and reused

Both the SUN Reference Implementation and Apache MyFaces implementation are shipped with the product.

The Sun JSF Reference Implementation provides the foundation of the code used for the JSF support in WebSphere Application Server. However, some dependencies on Jakarta APIs have been removed and replaced with Application Server specific solutions as a result of potential problems that may occur when open source APIs are included in the Application Server runtime. For example, when included in the Application Server runtime, these open source APIs are made available to all applications installed within the Application Server, therefore bringing versioning, support and legal issues. The version of the JSF runtime provided by the Application Server resides in the normal runtime library location and is available to all Web applications that leverage JSF APIs. The loading of the JSF servlet works in the same manner as if the runtime was packaged with the Web application.

The following open source dependencies are replaced with other APIs or in-house versions:

- Jakarta Commons BeanUtils
- Jakarta Commons Collections
- Jakarta Commons Digester
- Jakarta Commons Logging
- Mozilla Assert API

The JSF Specification requires JavaServer Pages Standard Tag Library (JSTL) as a dependency, therefore the required version of the JSTL from Jakarta is made available in the Application Server runtime.

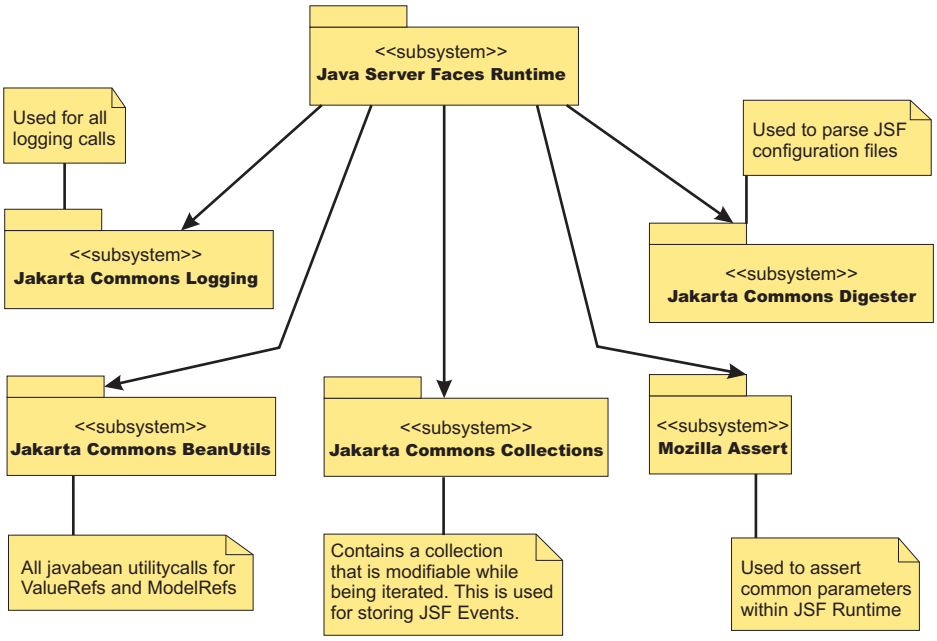
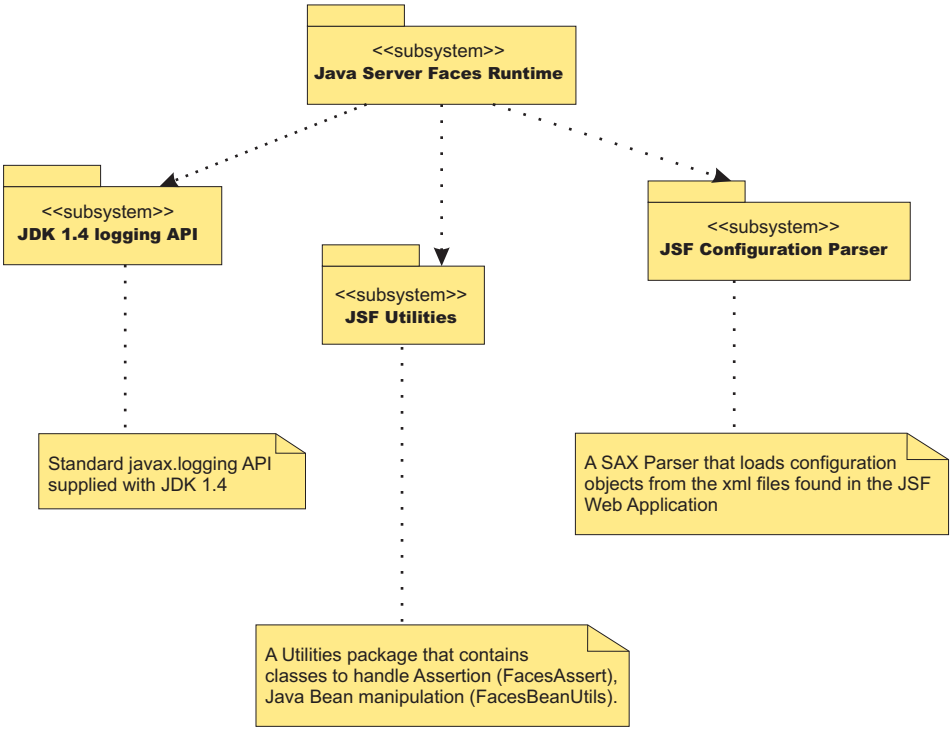


Figure 1. Current external API dependencies from the Sun based JSF runtime

Figure 2. Replacement APIs



The specification related classes (javax.faces.*) for JSF and the IBM modified version of the JSF Sun reference implementation are packaged in the Application Server runtime.

Typically, Web applications that leverage this API/Framework embed the JSF API and implementation Java archive (JAR) files within their Web archive (WAR) file. This is not required when these Web applications are deployed and run within WebSphere Application Server. Only the removal of these JAR files along with any JSTL JAR files from the WAR file is required. However, because JavaServer Faces 1.2 is a part of the Java Platform, Enterprise Edition (Java EE) platform, a Web application does not need to bundle a JavaServer Faces implementation when it runs on a Web container that is Java EE technology compliant. If a JavaServer Faces implementation is bundled with a Web application, it is ignored as the JavaServer Faces implementation provided by the platform always takes precedence.

The JSF runtime for WebSphere Application Server does not support the use of a single class loader for the entire application. This support is not available when the application contains multiple Web modules and one of those modules is a JSF module. A single class loader for the entire application is not supported because the FacesConfig initialization requires a single class loader for each JSF module to perform the initialization. Therefore, you must use multiple class loaders when the application contains multiple Web modules and at least one JSF module.

For using different implementations of JSF, the WAS JSF engine determines if the SUN RI or Apache MyFaces is used from the application server runtime. After the JSF engine determines the implementation that is used, the correct listener class is registered with the Web container. You not need to add the com.sun.faces.ConfigureListener or the org.apache.myfaces.StartupConfigureListener to your web.xml file.

If you want to use a third party JSF implementation that is not shipped with the product, leave the configuration set to the SUN RI, add the third party listener to the web.xml file that is required and add the third party implementation JAR files to the application as an isolated shared library. Using an isolated shared library, the Web application version of the JSF or JSTL classes load before the Application Server.

FacesBeanUtils class

The FacesBeanUtils class provides static method replacements for methods used in the Jakarta Commons BeanUtils API. The FacesBeanUtils class has no life cycle.

FacesBeanUtils
+ getProperty ([in] bean : Object , [in] property : String) : Object
+ getPropertyType ([in] bean : Object , [in] property : String) : Class
+ getSimpleProperty ([in] bean : Object, [in] property : String , [in] value : Object)
+ getProperty ([in] bean : Object , [in] property : String , [in] value : Object)
+ convertFromString ([in] value : String, [in] valueClass : Class) : Object
+ convert ([in] targetType : Class , [in] bean : String) : Object

JavaServer Faces widget library (JWL)

JavaServer Faces widget library (JWL) is a IBM JSF-based Web widget library that integrates widgets from a number of sources. The IBM JSF-based Web widget library is deprecated, however, you can obtain the latest version from Rational® Application Developer version 6 (RAD) to work with JSF 1.2.

JWL includes the JSF components from Rational Application Developer with the exception of the base JSF components, which are included in the Application Server runtime. This includes the IBM extended JSF components and the extended FacesClient Component. JWL also extends JSF with client-side features for rich browser-based experiences in the form of the FacesClient Component.

JWL Java archive files

JWL is packaged into two Java archive (JAR) files, odc-jsf.jar and jsf-ibm.jar, which are located in the ${WAS_HOME}\optionalLibraries\IBM\jwl\2.0 directory.

To include JWL in your application, you can use the JWL shared library named JWLLib, which is created at install time. To assign the library to an application, see the topic, Using installed optional packages.

Configuring JavaServer Faces implementation

Use this task to specify which JavaServer Faces implementation to use. You can use the Apache MyFaces or the SUN Reference Implementation of JSF 1.2 or your own implementation.

Before you begin

Ensure that your application is configured for JavaServer Faces (JSF) using the specific web.xml context parameters for the implementation that you have chosen.

About this task

The Application Server JSF engine determines if the SUN Reference Implementation 1.2 or Apache MyFaces 1.2 is used from the Application Server runtime. If either is used, the correct listener class is registered with the Web container. You do not need to add the com.sun.faces.ConfigureListener or the org.apache.myfaces.StartupConfigureListener to your web.xml file.

If the you want to use a third party JSF implementation that is not shipped with the product, keep the configuration set to the SUN RI, add the third party listener to the web.xml file that is required and add the third party implementation Java archive (JAR) files to the application as an isolated shared library.

You can also configure the JSF implementation on the **Provide JSP reloading options for Web modules** panel for application installation and update wizards.

1. In the administrative console panel, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **JSP and JSF options**. Select one of the following implementations:

Option	Description
Sun Reference Implementation 1.2	Select this option to use the Sun Reference Implementation 1.2 JSF implementation. This is the default JSF implementation.
MyFaces 1.2	Select this option to use the MyFaces 1.2 JSF implementation.

2. To use your own JSF implementation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **JSP and JSF options** in the administrative console.
   a. Select Sun Reference Implementation 1.2 as the JSF implementation.
   b. Add your implementation as an isolated shared library. See Creating shared libraries for details.
   c. Associate the isolated shared library to the Web module class loader in the application.
3. Click **Ok**.
4. Recompile the JSP files that contain the JSF implementation. You can set the JSF engine configuration paramater, com.ibm.ws.jsf.JSF_IMPL_CHECK, to true to automatically mark the JSP files to recompile at application startup.

Results

If MyFaces is selected, then the MyFaces implementation is added to the application through an isolated shared library and the entire application uses the MyFaces implementation. Using both the SUN Reference Implementation and the MyFaces implementation within the same application is not supported.

What to do next

Configure JSF engine parameters as necessary.

Related concepts

"JavaServer Faces" on page 52
JavaServer Faces (JSF) is a user interface framework or application programming interface (API) that eases the development of Java based Web applications. WebSphere Application Server version 7.0 supports JavaServer Faces 1.2 at a runtime level, therefore using JSF reduces the size of the Web application since runtime binaries no longer need to be included in your Web application.

Related tasks

Managing JavaServer Faces implementations using scripting
JavaServer Faces (JSF) is a user interface framework or application programming interface (API) that eases the development of Java based Web applications. The product supports JSF at a runtime level, which reduces the size of Web applications since runtime binaries no longer need to be included in your Web application. Use the wsadmin tool to set the JSF implementation as the Sun Reference 1.2 implementation or the Apache MyFaces 1.2 project.

Related reference

"JSF engine configuration parameters" on page 58
In WebSphere Application Server, you can configure the JavaServer Faces (JSF) engine configuration parameters for optimal performance in a production server environment and for the needs of developers in a development environment.

Configuring JSF engine parameters

About this task

WebSphere Application Server does not support the modification of deployment descriptor extension parameters through the administrative console or through administrative scripting.

To add, change or delete JSF engine configuration parameters, complete the following steps:

1. Open the `WEB-INF/web.xml` file.

   JSP engine configuration parameters are stored in a Web module's configuration directory or in a Web modules's binaries directory in the `WEB-INF/web.xml` file. Open the `WEB-INF/web.xml` file from:

   - The configuration directory, as in the following example: `{WAS_ROOT}/profiles/`*`profilename`*`/config/cells/`*`cellname`*`/applications/`*`enterpriseappname`*`/ deployments/`*`deployedname`*`/`*`webmodulename`*
   - The binaries directory if an application was deployed into WebSphere Application Server with the flag "Use Binary Configuration" set to true. An example of a binaries directory is: `{WAS_ROOT}/profiles/`*`profilename`*`/installedApps/`*`nodename`*`/`*`EnterpriseAppName`*`/`*`WebModuleName`*`/`
2. Edit the `WEB-INF/web.xml` file.
   - To add configuration parameters, use the following format:

     ```
     <context-param>
       <description>descriptive text</description>
       <param-name>parameter name</param-name>
       <param-value>parameter value</param-value>
     </context-param>
     ```
   - To delete configuration parameters, either delete the line from the file, or enclose the statement with `<!-- -->` tags.
3. Save the file.
4. Restart the Enterprise Application. It is not necessary to restart the server for parameter changes to take effect. However, some JSP engine configuration parameters affect the Java source code that is generated for a JSP. If such a parameter is changed, then you must retranslate the JSP files in the Web module to regenerate Java source. You can use the batch compiler to retranslate all JSP files in a Web module. The batch compiler uses the JSP engine configuration parameters that you have set in the `web.xml` file, unless you specifically override them. The topic JSP engine configuration parameters" identifies the parameters that affect the generated Java source.

Example

The following is a sample of the WEB-INF/web.xml file. The lines in bold text are JSP engine configuration parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<webappext:WebAppExtension xmi:version="2.0" xmlns:xmi=http://www.omg.org/XMI
   xmlns:webappext="webappext.xmi" xmlns:webapplication="webapplication.xmi" xmi:id="WebAppExtension_1"
   reloadInterval="9" reloadingEnabled="true" defaultErrorPage="error.jsp" additionalClassPath=""
   fileServingEnabled="true" directoryBrowsingEnabled="false" serveServletsByClassnameEnabled="true"
   autoRequestEncoding="true" autoResponseEncoding="false"
   <webApp href="WEB-INF/web.xml#WebApp_1"/>
   <jspAttributes xmi:id="JSPAttribute_1" name="useThreadTagPool" value="true"/>
   <jspAttributes xmi:id="JSPAttribute_2" name="verbose" value="false"/>
   <jspAttributes xmi:id="JSPAttribute_3" name="deprecation" value="false"/>
   <jspAttributes xmi:id="JSPAttribute_4" name="reloadEnabled" value="true"/>
   <jspAttributes xmi:id="JSPAttribute_5" name="reloadInterval" value="5"/>
   <jspAttributes xmi:id="JSPAttribute_6" name="keepgenerated" value="true"/>
   <!--<jspAttributes xmi:id="JSPAttribute_7" name="trackDependencies" value="true"/> -->

</webappext:WebAppExtension>
```

JSF engine configuration parameters:

In WebSphere Application Server, you can configure the JavaServer Faces (JSF) engine configuration parameters for optimal performance in a production server environment and for the needs of developers in a development environment.

The JSF engine parameters are case sensitive. If the value specified for a parameter is comprised of two or more words separated by spaces, you must add quotation marks around the value.

JSF options using SUN RI

- **com.sun.faces.numberOfViewsInSession**

  Specifies the number of views that are stored in the session when Server-Side State Saving is used. If set to true while client-side state saving is being used, reduces the number of bytes sent to the client by compressing the state before it is encoded and written as a hidden field. The default for this parameter is 15.
- **com.sun.faces.numberOfLogicalViews**

  Specifies the number of logical views that are stored in the session when Server-Side State Saving is used. The default for this parameter is 15.
- **com.sun.faces.enableHighAvailability**

  If set to true while server-side state saving is used, a serialized representation of the view is stored on the server. This provides failover and sever clustering support. The default for this parameter is false.
- **com.sun.faces.injectionProvider**

  Defines an injection provider that is used for JSF annotations.
- **com.sun.faces.serializationProvider**

  Defines a serialization provider that is used for serializing JSF objects into session.
- **com.sun.faces.responseBufferSize**

  Define the size of the response buffer for a JSF response. The default for this parameter is 1048.
- **com.sun.faces.clientStateWriteBufferSize**

  The default for this parameter is 8192.
- **com.sun.faces.expressionFactory**

  Specifies the default EL Expression Factory to use. The default for this parameter is org.apache.el.ExpressionFactoryImpl.
- **com.sun.faces.clientStateTimeout**

  The timeout value used for client side state saving. Once the value set has been reached then the state is lost. Default is infinite.

- **com.sun.faces.displayConfiguration**

  The default for this parameter is `false`.
- **com.sun.faces.validateXml**

  The default for this parameter is `false`.
- **com.sun.faces.verifyObjects**

  The default for this parameter is `false`.
- **com.sun.faces.forceLoadConfiguration**

  The default for this parameter is `false`.
- **com.sun.faces.diableVersionTracking**

  The default for this parameter is `false`.
- **com.sun.faces.enableHtmTagLibValidator**

  The default for this parameter is `false`.
- **com.sun.faces.prerefXHTML**

  The default for this parameter is `false`.
- **com.sun.faces.compressViewState**

  The default for this parameter is `true`.
- **com.sun.faces.compressJavaScipt**

  The default for this parameter is `true`.
- **com.sun.faces.sendPoweredByHeader**

  The default for this parameter is `true`.
- **com.sun.faces.enableJSStyleHiding**

  The default for this parameter is `false`.
- **com.sun.faces.writeStateAtFormEnd**

  The default for this parameter is `true`.
- **com.sun.faces.enableLazyBeanValidation**

  The default for this parameter is `ture`.
- **com.sun.faces.enableLoadBundle11Compatibility**

  The default for this parameter is `false`.
- **com.sun.faces.enableRestoreView11Compatibilty**

  The default for this parameter is `false`.
- **com.sun.face.serializeServerState**

  The default for this parameter is `false`.

JSF options for MyFaces

- **org.apache.myfaces.RESOURCE_VIRTUAL_PATH**

  The default for this parameter is `/faces/myFacesExtensionResource`.
- **org.apache.myfaces.PRETTY_HTML**

  The default for this parameter is `true`.
- **org.apache.myfaces.ALLOW_JAVASCRIPT**

  The default for this parameter is `true`.
- **org.apache.myfaces.DETECT_JAVASCRIPT**

  The default for this parameter is `false`.
- **org.apache.myfaces.AUTO_SCROLL**

  The default for this parameter is `false`.
- **org.apache.myfaces.ADD_RESOURCE_CLASS**

  The default for this parameter is `org.apache.myfaces.renderkit.html.util.DefaultAddResource`.
- **org.apache.myfaces.CHECK_EXTENSIONS_FILTER**

  The default for this parameter is `true`.
- **org.apache.myfaces.READONY_AS_DISABLED_FOR_SELECTS**

  The default for this parameter is `true`.

- **org.apache.myfaces.SERIALIZE_STATE_IN_SESSION**

  Set this option to `true` to serialize the state to a byte stream before it is written to the session. If this option is set to `false`, the state is not serialized to a byte stream. This option is only applicable if the state saving method is set to server. The default for this parameter is `true`.
- **org.apache.myfaces.COMPRESS_STATE_IN_SESSION**

  Set this option to `true` to compress the serialized state before it is written to the session. If this option is set to `false`, the state is not compressed. This option is only applicable if the state saving method is set to server and if org.apache.myfaces.SERIALIZE_STATE_IN_SESSION is set to `true`. The default for this parameter is `true`.
- **org.apache.myfaces.NUMBER_OF_VIEWS_IN_SESSION**

  Defines the number of the latest views that are stored in session. This option is only applicable if the state saving method is set to server. The default for this parameter is `20`.

JSF options using SUN RI or MyFaces

The following options are valid for both the SUN RI and the MyFaces implementations.
- **javax.faces.STATE_SAVING_METHOD**

  Specifies the location where state information is saved. Valid values are 'server', which is saved in HttpSession, and 'client', which is saved as a hidden field in the form. The default for this parameter is `server`.
- **javax.faces.CONFIG_FILES**

  Use this parameter to specify a comma-delimited list of context-relative resource paths under which the JSF implementation looks for application configuration resources before loading a configuration resource named /WEB-INF/facesconfig.xml, if a resource exists.
- **javax.faces.DEFAULT_SUFFIX**

  Specifies the default suffix for extension-mapped resources that contain JSF components. The default for this parameter is `.jsp`.
- **javax.faces.LIFECYCLE_ID**

  Use this parameter to configure an alternate lifecycle ID.
- **com.ibm.ws.jsf.JSF_IMPL_CHECK**

  Specifies that the JSP files in a Web module must be recompiled when the application is restarted because the implementation of JSF that is used has changed. After the application is restarted, the next time a JSP files is accessed for this module the JSP is recompiled against the selected implementation of JSF specified in the administration console. Subsequent calls to the JSP do not cause a recompile. The default setting for this option is false. Use this option for development and not in a production environment.

IBM options for JSF runtime
- **com.ibm.ws.jsf.JSP_UPDATE_CHECK**

  This parameter monitors Faces JaaverServer Pages (JSP) files for modifications and synchronizes a running server with the changes without restarting the server. If this parameter is set to false or removed from the deployment descriptor, any changes made to Faces JSP files might not be seen by the server until it is restarted. Set this parameter to true while developing and debugging the Faces JSP files to improve the performance of the development environment.
- **com.ibm.ws.jsf.LOAD_FACES_CONFIG_AT_STARTUP**

  Specifies to load the JSF runtime when the application server starts up. If this parameter is set to false or removed, JSF runtime is loaded and initialized when the first JSF request is processed. This might disable custom JSF extensions such as factories defined in the project.
- **com.ibm.ws.jsf.JSF_IMPL_CHECK**

  Set the com.ibm.ws.jsf.JSF_IMPL_CHECK parameter to true to check at application restart if the SUN RI and MyFaces implementations were switched. If the implementation has switched, then the runtime removes any generated JSP files from the temp directory and the JSP file is retranslated the next time it is requested.

JavaServer Faces custom properties

You can configure name-value pairs of data, where the name is a property key and the value is a string value that you can use to set internal system configuration properties. Defining a new property enables you to configure a setting beyond what is available in the administrative console. The following is a list of the available JavaServer Faces custom properties.

com.ibm.ws.jsf.disableStylePassthroughForCheckboxList:

This custom property prevents passing the style information into the items in the check box list. This property defaults to false to maintain the current behavior. Define and set the com.ibm.ws.jsf.disableStylePassthroughForCheckboxList context parameter to `true` in the web.xml file prevent passing style information into items in the check box list. Use the following code as an example.

```
<context-param>
<description>
Set to true if style information should not be passed into items of check box list
</description>
<param-name>com.ibm.ws.jsf.disableStylePassthroughForCheckboxList</param-name>
<param-value>true</param-value>
</context-param>
```

com.ibm.ws.jsf.associateLabelWithId:

The com.ibm.ws.jsf.associateLabelWithId custom property changes the rendering behavior for both the <h:selectOneRadio> and <h:selectManyCheckbox> components. The label no longer wraps the input element. Instead, each input element has a unique ID and the label is associated with that ID used for that attribute. Define and set the com.ibm.ws.jsf.associateLabelWithId context parameter to `true` in the web.xml file. Use the following code as an example.

```
<context-param>
<description>
Set to true to explicitly associate labels with their input elements for select one radio buttons
and select many check box lists.
</description>
<param-name>com.ibm.ws.jsf.associateLabelWithId</param-name>
<param-value>true</param-value>
</context-param>
```

enableRestoreView11Compatibility:

Because JSF 1.2 is supported, a JSF 1.2 application, might create the ViewExpiredException exception under load. If your view is not found in session, you can use a compatibility mode in JSF to create a new view. This can have adverse behaviors because it is a new view, and items that are usually in the view, such as state, are no longer be there. Use the following code as an example to add the com.sun.faces.enableRestoreView11Compatibility context parameter to `true` in the web.xml file.

```
<context-param>
   <param-name>com.sun.faces.enableRestoreView11Compatibility</param-name>
   <param-value>true</param-value>
</context-param>
```

com.ibm.ws.jsf.loadExternalDtd:

When parsing the faces-config.xml file from included libraries, the Faces configuration parser attempts to load the DTD even when validation is disabled. The Faces configuration parser uses a SAXParser to read the faces-config.xml. The default behavior of the SAXParser parser is to always load the DTD even if validation is disabled. This behavior can lead to errors initializing the Faces Servlet on systems isolated from the internet.

In your web.xml file, set the com.ibm.ws.jsf.loadExternalDtd context paramater to `false` to have the Faces configuration parser set the ″http://apache.org/xml/features/nonvalidating/load-external-dtd″ feature to false.

```
<context-param>
<description>
When set to false, this property sets a feature on the SAX parser to prevent loading the external DTD.
</description>
<param-name>com.ibm.ws.jsf.loadExternalDtd</param-name>
<param-value>false</param-value>
</context-param>
```

Assembling Web applications

Assemble a Web module to contain servlets, JavaServer page (JSP) files, and related code artifacts. (Group enterprise beans, client code, and resource adapter code in separate modules). After assembling a Web module, you can install it as a standalone application or combine it with other modules into an enterprise application.

Before you begin

This topic assumes that you have created and unit tested Servlets, JavaServer Pages (JSP) files and other Web components that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Use an assembly tool to assemble a Web module in any of the following ways:
- Import an existing Web module (WAR file).
- Create a new Web module.
- Copy code artifacts (such as servlets) from one Web module into a new Web module.

Although you can input various properties for Web archives, available properties are specific to the Servlet, JSP, and Java Platform, Enterprise Edition (Java EE) specification level.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on Java EE modules. Ensure that **J2EE** and **Web** capabilities are enabled.
3. Migrate WAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an assembly tool. To migrate files, import your WAR files to the assembly tool.
4. Create a new Web module.
5. Copy code artifacts (such as servlets) from one Web module into a new Web module.

Results

A Web project is migrated or created. Files for the Web project are shown in the Project Explorer view under **Enterprise Applications** and **Web Projects**.

What to do next

You can now deploy your Web project to an application server.

Web component security

A Web module consists of servlets, JavaServer Pages (JSP) files, server-side utility classes, static Web content, which includes HTML, images, sound files, cascading style sheets (CSS), and client-side classes or applets. You can use development tools such as Rational Application Developer to develop a Web module and enforce security at the method level of each Web resource.

You can identify a Web resource by its URI pattern. A Web resource method can be any HTTP method (GET, POST, DELETE, PUT, for example). You can group a set of URI patterns and a set of HTTP

methods together and assign this grouping a set of roles. When a Web resource method is secured by associating a set of roles, grant a user at least one role in that set to access that method. You can exclude anyone from accessing a set of Web resources by assigning an empty set of roles. A servlet or a JavaServer Pages (JSP) file can run as different identities before invoking another enterprise bean component. All the secured Web resources require the user to log in by using a configured login mechanism. Three types of Web login authentication mechanisms are available: basic authentication, form-based authentication and client certificate-based authentication.

In WebSphere Application Server Version 6.1, a portlet resource that is part of a web module can also be protected when it is accessed directly through URL. The protection is similar to other Web based resources. For more information, see “Portlet URL security” on page 96.

For more detailed information on Web security, see the product architectural overview article.

Securing Web applications using an assembly tool

You can use three types of Web login authentication mechanisms to configure a Web application: basic authentication, form-based authentication and client certificate-based authentication. Protect Web resources in a Web application by assigning security roles to those resources.

About this task

To secure Web applications, determine the Web resources that need protecting and determine how to protect them.

Note: This procedure might not match the steps that are required when using your assembly tool, or match the version of the assembly tool that you are using. You should follow the instructions for the tool and version that you are using.

The following steps detail securing a Web application using an assembly tool:

1. In an assembly tool, import your Web archive (WAR) file or an application archive (EAR) file that contains one or more Web modules.
2. In the Project Explorer folder, locate your Web application.
3. Right-click the deployment descriptor and click **Open With > Deployment Descriptor Editor**. The Deployment Descriptor window opens. To see online information about the editor, press F1 and click the editor name. If you select a Web archive (WAR) file, a Web deployment descriptor editor opens. If you select an enterprise application (EAR) file, an application deployment descriptor editor opens.
4. Create security roles either at the application level or at the Web module level. If a security role is created at the Web module level, the role also displays in the application level. If a security role is created at the application level, the role does not display in all of the Web modules. You can copy and paste a security role at the application level to one or more Web module security roles.
   - Create a role at a Web-module level. In a Web deployment descriptor editor, click the Security tab. Under **Security Roles**, click **Add.**. Enter the security role name, describe the security role, and click **Finish**.
   - Create a role at the application level. In an application deployment descriptor editor, click the Security tab. Under the list of security roles, click **Add**. In the Add Security Role wizard, name and describe the security role and then click **Finish**.
5. Create security constraints. Security constraints are a mapping of one or more Web resources to a set of roles.
   a. On the Security tab of a Web deployment descriptor editor, click **Security Constraints**. On the Security Constraints tab, you can do the following actions:
      - Add or remove security constraints for specific security roles.
      - Add or remove Web resources and their HTTP methods.
      - Define which security roles are authorized to access the Web resources.

- Specify None, Integral, or Confidential constraints on user data.
  None The application does not require transport guarantees.
  Integral
  Data cannot be changed in transit between the client and the server.
  Confidential
  Data content cannot be observed while it is in transit.
  Integral and Confidential usually require the use of SSL. When deploying applications that are available over public networks, specify Confidential for your Web Applications constraints

b. Under Security Constraints, click **Add**.

c. Under Constraint name, specify a display name for the security constraint and click **Next**.

d. Type a name and description for the Web resource collection.

e. Select one or more HTTP methods. The HTTP method options are: GET, PUT, HEAD, TRACE, POST, DELETE, and OPTIONS.

f. Beside the Patterns field, click **Add**.

g. Specify a URL Pattern. For example, type - `/*, *.jsp, /hello`. Consult the Servlet specification Version 2.4 for instructions on mapping URL patterns to servlets. The security runtime uses the exact match first to map the incoming URL with URL patterns. If the exact match is not present, the security runtime uses the longest match. The wild card (`*.,*.jsp`) URL pattern matching is used last.

h. Click **Finish**.

i. Repeat these steps to create multiple security constraints.

6. Map security-role-ref and role-name elements to the role-link element. During the development of a Web application, you can create the security-role-ref element. The security-role-ref element contains only the role-name field. The role-name field contains the name of the role that is referenced in the servlet or JavaServer Pages (JSP) code to determine if the caller is in a specified role. Because security roles are created during the assembly stage, the developer uses a logical role name in the Role-name field and provides enough description in the Description field for the assembler to map the role actual. The Security-role-ref element is at the servlet level. A servlet or JavaServer Pages (JSP) file can have zero or more security-role-ref elements.

   a. Go to the References tab of a Web deployment descriptor editor. On the References tab, you can add or remove the name of an enterprise bean reference to the deployment descriptor. You can define five types of references on this tab:
      - EJB reference
      - Service reference
      - Resource reference
      - Message destination reference
      - Security role reference
      - Resource environment reference

   b. Under the list of Enterprise JavaBeans (EJB) references, click **Add**.

   c. Specify a name and a type for the reference in the **Name** and **Ref Type** fields.

   d. Select either **Enterprise Beans in the workplace** or **Enterprise Beans not in the workplace**.

   e. Optional: If you select **Enterprise Beans not in the workplace**, select the type of enterprise bean in the **Type** field. You can specify either an entity bean or a session bean.

   f. Optional: Click **Browse** to specify values for the local home and local interface in the **Local home** and **Local** fields before you click **Next**.

   g. Map every role-name that is used during development to the role using the previous steps. Every role name that is used during development maps to the actual role.

7. Specify the RunAs identity for servlets and JSP files. The RunAs identity of a servlet is used to invoke enterprise beans from within the servlet code. When enterprise beans are invoked, the RunAs identity

is passed to the enterprise bean for performing an authorization check on the enterprise beans. If the RunAs identity is not specified, the client identity is propagated to the enterprise beans. The RunAs identity is assigned at the servlet level.

a. On the Servlets tab of a Web deployment descriptor editor, under **Servlets and JSP**, click **Add**. The Add Servlet or JSP wizard opens.
b. Specify the servlet or JavaServer Pages (JSP) file settings, including the name, initialization parameters, and URL mappings and click **Next**.
c. Specify the class file destination.
d. Click **Next** to specify additional settings or click **Finish**.
e. Click **Run As** on the **Servlets** tab, select the security role and describe the role.
f. Specify a RunAs identity for each servlet and JSP file that is used by your Web application.

8. Configure the login mechanism for the Web module. This configured login mechanism applies to all the servlets, JavaServer Pages (JSP) files and HTML resources in the Web module.
   a. Click the **Pages** tab of a Web deployment descriptor editor and click **Login**. Select the required authentication method. Available method values include: Unspecified, Basic, Digest, Form, and Client-Cert.
   b. Specify a realm name.
   c. If you select the Form authentication method, select a login page and an error page Web address. For example, you might use `/login.jsp` or `/error.jsp`. The specified login and error pages are present in the `.war` file.
   d. Install the client certificate on the browser Web Client and place the client certificate in the server trust keyring file, if ClientCert certificate is selected. The public certificate of the clients certificate authority must be placed in the servers RACF® keyring. If the registry is a local OS registry, use the RACDCERT MAP or the equivalent System Authorization Facility (SAF) command to enable an MVS™ identity creation using the client certificate.
9. Close the deployment descriptor editor and, when prompted, click **Yes** to save the changes.

Results

After securing a Web application, the resulting Web archive (WAR) file contains security information in its deployment descriptor. The Web module security information is stored in the `web.xml` file. When you work in the Web deployment descriptor editor, you also can edit other deployment descriptors in the Web project, including information on bindings and IBM extensions in the `ibm-web-bnd.xmi` and `ibm-web-ext.xmi` files.

What to do next

After using an assembly tool to secure a Web application, you can install the Web application using the administrative console. During the Web application installation, complete the steps in Deploying secured applications to finish securing the Web application.

Security constraints

Security constraints determine how Web content is to be protected.

These properties associate security constraints with one or more Web resource collections. A constraint consists of a Web resource collection, an authorization constraint and a user data constraint.

- A Web resource collection is a set of resources (URL patterns) and HTTP methods on those resources. All requests that contain a request path that matches the URL pattern described in the Web resource collection are subject to the constraint. If no HTTP methods are specified, then the security constraint applies to all HTTP methods.

- An authorization constraint is a set of roles that users must be granted in order to access the resources described by the Web resource collection. If a user who requests access to a specified Uniform Resource Identifier (URI) is not granted at least one of the roles specified in the authorization constraint, the user is denied access to that resource.

  Previously the http-methodType schema limited HTTP methods to DELETE, GET, HEAD, OPTIONS, POST, PUT and TRACE. The http-methodType schema has changed. The http-methodType schema has been changed as follows:

  ```
  <xsd:simpleType name="http-methodType">
  <xsd:annotation>
  <xsd:documentation>
  A HTTP method type as defined in HTTP 1.1 section 2.2.
  </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
  <xsd:pattern value="[\p{L}-[\p{Cc}\p{Z}]]+"/>
  </xsd:restriction>
  </xsd:simpleType>
  ```

  This requires elements to be a token. Based upon the pattern value, tokens can contain any character except for control characters and separators.
- A user data constraint indicates that the transport layer of the client or server communications process must satisfy the requirement of either guaranteeing content integrity (preventing tampering in transit) or guaranteeing confidentiality (preventing reading while in transit).

Security settings

Use the administrative console to modify the security settings for all applications.

You can enable security for applications by selecting the **Enable application security** option on the Global security panel.

The default settings are used as a template or starting point for configuring individual applications. The administrator should still explicitly configure security settings for each application.

The following security settings are specified during application assembly:

Security role settings

When using the Assembly Toolkit at an application level (Enterprise Archive (EAR) file), security roles are synchronized with the security roles defined for the embedded modules of the application.

If a security role is manually added to the EAR file, it can be automatically removed when the file is saved if an embedded module does not reference the role, or the role is in conflict with an existing role. In this case, remove the manually added role, but then all roles with the same name are removed.

The role is automatically added again when the file is saved if it is still referenced in an embedded module file. If a duplicate role is added in an embedded module file, delete all roles with the same name and manually read the correct role.

Security constraints

Security constraints declare how to protect Web content. These properties associate security constraints with one or more Web resource collections. A *constraint* consists of a Web resource collection, an authorization constraint, and a user data constraint.

Security constraints are set when configuring a Web application in the Assembly Toolkit.

Security role references

Web application developers or Enterprise JavaBeans (EJB) providers must use a role-name in the code when using the available programmatic security Java Platform, Enterprise Edition (Java EE) application programming interfaces (APIs) isUserInRole(String roleName) and isCallerInRole(String roleName).

The roles used in the deployed run-time environment might not be known until the Web application and EJB components (for example, Web archive (WAR) files and `ejb-jar.xml` files) are assembled into an enterprise archive (EAR) file. Therefore, the role names used in the Web application or EJB component code are logical role names which the application assembler maps to the actual run-time environment roles during application assembly. The security role references provide a level of indirection that insulate Web application component and EJB developers from having to know the actual roles in the run-time environment.

The definition of the logical roles and the mapping to the actual run-time environment roles are specified in the security-role-ref element of both the Web application and the EJB JAR file deployment descriptors, `web.xml` and `ejb-jar.xml` respectively. Use the assembly tools to define the role names and map them to the actual run-time roles in the environment with the role-link element.

The following code sample is an example of a security-role-ref from an EJB `ejb-jar.xml` deployment descriptor.

```
... <enterprise-beans>
... <entity>
<ejb-name>AardvarkPayroll</ejb-name>
<ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
...
<security-role-ref>
<description>
```

This role should be assigned to the employees of the payroll department. Members of this role have access to the payroll record of everyone. The role has been linked to the payroll-department role. This role should be assigned to the employees of the payroll department. Members of this role have access to all payroll records. The role has been linked to the payroll-department role.

```
</description> <role-name>payroll</role-name>
<role-link>payroll-department</role-link>
</security-role-ref>
 ...
</entity>
 ...
</enterprise-beans>
```

In the previous example, the string `payroll`, which appears in the <role-name> element, is what the EJB provider uses as the argument to the isCallerInRole() API. The <role-link> element is what ties the logical role to the actual role used in the run-time environment.

Note that for enterprise beans, the security-role-ref element must appear in the deployment descriptor even if the logical role name is the same as the actual role name in the environment.

The rules Web application components are slightly different. If no security-role-ref element matching a security-role element is declared, the container must default to checking the role-name element argument against the list of security-role elements for the Web application. The isUserInRole method references the list to determine whether the caller is mapped to a security role. The developer must be aware that the use of this default mechanism can limit the flexibility in changing role names in the application without having to recompile the servlet making the call.

See the EJB Version 2.0 and Servlet Version 2.3 specification in the Security: Resources for Learning article for complete details on this specification.

File serving

In file serving, Web applications can serve static file types, such as HTML. File-serving attributes are used by the servlet that implements file-serving behavior.

The file-serving behavior is implemented by setting the fileservingenabled property to true when configuring the Web module.

Example attributes:

bufferSize
Sets buffer size that is used for serving static files.

extendedDocumentRoot
Path that specifies the directory where static files are sent. Use this attribute in addition to the contextRoot attribute.

file.serving.patterns.allow
Specifies that only files matching the specified pattern are served.

file.serving.patterns.deny
Specifies that files that match the specified file pattern are denied

Defining an extension for the registry filter

The registry filter specifies if an extensions is applicable to all registry instances or to specified instances.

Before you begin

You must have an extensible application to define an extension for the registry filter.

About this task

Complete the following steps to filter out extensions for an application.

1. Define an extension for the registry filter extension point for a named registry instance in the plugin.xml file.

```
<extension point="org.eclipse.extensionregistry.RegistryFilter">
 <filter name="AdminConsole*"
   class="com.ibm.ws.admin.AdminConsoleExtensionFilter"/>
 </extension>
```

2. Add the filter implementation to the application by creating a class to implement the com.ibm.workplace.extension.IExtensionRegistryFilter interface.

```
package com.ibm.ws.admin;
import com.ibm.workplace.extension.IExtensionRegistryFilter;
public class AdminConsoleExtensionFilter implements IExtensionRegistryFilter {
        :
}
```

3. The extensible application declares the registry name by defining an extension for the RegistryInstance extension point. This way, the registry can prepare an IExtensionRegistry instance and put it in JNDI in advance.

```
<extension point="org.eclipse.extensionregistry.RegistryInstance">
  <registry name="AdminConsole"/>

</extension>
```

4. The extensible application obtains a named instance of the registry to activate any associated filters:

```
InitialContext ic = new InitialContext();
String lookupName = "services/extensionregistry/AdminConsole";
IExtensionRegistry reg = (IExtensionRegistry)ic.lookup( lookupName );
```

Application extension registry

WebSphere Application Server has enabled the Eclipse extension framework for applications to use. Applications are extensible when they contain a defined extension point and provide the extension processing code for the extensible area of the application.

An application can be plugged in to another extensible application by defining an extension that adheres to what the target extension point requires. The extension point can find the newly added extension dynamically and the new function is seamlessly integrated in the existing application. It works on a cross Java 2 Platform, Enterprise Edition (J2EE) module basis. The application extension registry uses the Eclipse plug-in descriptor format and application programming interfaces (APIs) as the standard extensibility mechanism for WebSphere applications. Developers that build WebSphere application modules can use WebSphere Application Server extensions to implement their functionality to an extensible application, which defines an extension point. This is done through the application extension registry mechanism.

The architecture of extensible J2EE applications follow a modular design to add new functional modules or to replace an existing module, particularly by those outside of its core development team. Each module is a pluggable unit, or plug-in that is either deployed into the portal or removed from the J2EE application using a deployment tool that is based upon standard J2EE and portal Web module deployment tooling. A plug-in module describes where it is extensible and what capability it provides to other plug-ins in the plugin.xml file. The plugin.xml manifest file can be created with a simple text editor or in Eclipse's Plug-in Development Environment (PDE), which provides a simplified view of the same underlying XML data.

You can find additional information about the Eclipse Plug-in Architecture at http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

WebSphere Application Server implementations to the Eclipse model

Some minor differences exist in the WebSphere Application Server implementation of this architecture because of platforms, specifically, Eclipse Workbench or Java 2 Platform, Enterprise Edition (J2EE). The highlights of the WebSphere Application Server implementation include:

- Implementing all of the extension registry-related interfaces from Eclipse 3.1.
- The identical plugin.xml syntax, however, some attributes are not used, for example, <runtime>.
- The discovery and addition of plug-ins to the registry, when the containing J2EE module starts, and plug-ins are dismissed and removed from the registry when the containing J2EE module stops.
- Access to an IExtensionRegistry object is through the Java Naming and Directory Interface (JNDI), instead of by using the Platform.getExtensionRegistry method in the Eclipse Workbench.
- Filtering capability is available by providing a filter implementation and using a named registry instance that finds and invokes the filter as necessary. See the developer API documentation for the IExtensionRegistryFilter interface for more details.

Available Eclipse 3.1 interfaces

The following Eclipse 3.1 interfaces are available on WebSphere Application Server:

- Extension registry API
- Extension point API
- Extension API
- Configuration element API
- Registry change listener API
- Registry change event API
- Extension delta API
- Status API

The following interfaces are recognized and processed the same as in Eclipse:

- Executable extension API
- Executable extension factory API

Application extension registry filtering

The extension registry exposes the registry filter extension point. The registry filter removes elements within the extension registry for client applications. Extensions that are attached to the registry filter extension point and that also implement this interface are called as necessary when a client operates on a named registry instance that matches the target specification.

You can create a filter extension for all registry instances or for named instances that are specified by the extension. In the first case, the filter is applied to all instances of the extension registry, and all client applications use the filter without requesting the filter. In the latter case, a client application must predefine the registry name by defining an extension, called *RegistryInstance*, which is another extension point that is exposed by the extension registry. After the registry name is defined, the client can obtain the named registry instance and use that registry instance. The filter extension is invoked by the named registry instance as necessary.

Registry filter API

Supported arguments include:

org.eclipse.core.runtime.IExtension[]
doFilter(org.eclipse.core.runtime.IExtension[] extensions)

This code returns an array of IExtension objects that are included in the valid extension list.

Registry instance extension point

The extension registry exposes the *RegistryInstance* . The instance name is declared in the application's plugin.xml file, and the application requests an registry instance for that name at runtime.

plugin.xml file

A plug-in is described in an XML manifest file, called plugin.xml, which is part of the plug-in deployment files. The manifest file tells the portal application's runtime what it needs to know to register and activate the plug-in. The manifest file essentially serves as the contract between the pluggable component and the portal application's runtime. Although the WebSphere Application Server plugin.xml closely follows the one provided for the Eclipse workbench, it does diverge from the Eclipse workbench in several places as outlined below.

Location

The `plugin.xml` file must reside in the WEB-INF directory under the context of the hierarchy of directories that exist for a Web application or when included in the Web application archive file.. The `plugin.xml` file must reside in the root directory when the`plugin.xml` file is placed in an Enterprise JavaBeans Java archive (JAR) file or shared library JAR file. The extension registry service includes the `plugin.xml` file as the participating components are loaded and started on the application server.

Usage notes

- Is this file read-only?

  No
- Is this file updated by a product component?

  ???
- If so, what triggers its update?

  Rational Application Developer updates the `web.xml` file when you assemble Web components into a Web module, or when you modify the properties of the Web components or the Web module.
- How and when are the contents of this file used?

  WebSphere Application Server functions use information in this file during the configuration and deployment phases of Web application development.

- The manifest markup definitions below make use of various naming tokens and identifiers. To eliminate ambiguity, the following are productions rules for these naming conventions. In general, all identifiers are case-sensitive.

```
SimpleToken := sequence of characters from ('a-z','A-Z','0-9')
ComposedToken := SimpleToken | (SimpleToken '.' ComposedToken)
PlugInId := ComposedToken
PlugInPrereq := PlugInId
ExtensionId := SimpleToken
ExtensionPointId := SimpleToken
ExtensionPointReference := ExtensionPointId | (PlugInId '.' ExtensionPointId)
```

Sample file entry

The entire plug-in manifest DTD is as follows. XML Schema is not used to define the manifest since the current Eclipse tooling for plug-in's requires a DTD. The XML DTD construction rule element* means zero or more occurrences of the element; element? means zero or one occurrence of the element; and element+ means one or more occurrences of the element.

```
<?xml encoding="US-ASCII"?>

<!ELEMENT plugin (requires?, extension-point*, extension*)>
<!ATTLIST plugin
  name CDATA #IMPLIED
  id CDATA #REQUIRED
  version CDATA #REQUIRED
  provider-name CDATA #IMPLIED
>
<!ELEMENT requires (import+)>
<!ELEMENT import EMPTY>
<!ATTLIST import
  plugin CDATA #REQUIRED
  version CDATA #IMPLIED
  match (exact | compatible | greaterOrEqual) #IMPLIED
>
<!ELEMENT extension-point EMPTY>
<!ATTLIST extension-point
  name CDATA #IMPLIED
  id CDATA #REQUIRED
  schema CDATA #IMPLIED
>
<!ELEMENT extension ANY>
<!ATTLIST extension
  point CDATA #REQUIRED
  id CDATA #IMPLIED
  name CDATA #IMPLIED
>
```

WebSphere Application Server differences

The `plugin.xml` file closely follows the `plugin.xml` file provided for the Eclipse workbench. However it diverges within the following elements.

The `plugin` element

The plugin element provided in this manifest does not contain class attributes. The class attribute is unnecessary since the plug-in mechanism does not require the plug-in developer to extend or use any specific classes as is required by the Eclipse workbench. Also, the plugin element does not contain a runtime element since standards such as J2EE that already define the location of runtime libraries for the applications.

The `import` element

The requires element does not contain export attribute since J2EE modules are encouraged to be self-contained to improve manageability. In addition to eliminating the export attribute, the match attribute has an option for a greater than or equal to match for versions (greaterOrEqual).

The `extension-point` element

The extension-point element has the name attribute as optional since it has no real use in this J2EE implementation.

you can find details regarding the plug-in manifest in the Eclipse documentation, under Platform Plug-In Developer Guide>Other reference information>Plug-in manifest.

The following is an example of how adding a link to an existing page can be accomplished by an extension point. The plug-in manifest of this plug-in declares an extension point (linkExtensionPoint) and an extension to this extension point (linkExtension). The plug-in declaring the extension point does not need to be the plug-in that implements the extension point. Another plug-in can also define an extension to the link extension point in its plug-in manifest by including the contents of the <extension> and </extension> tags in its manifest.

```
<?xml version="1.0"?>
<!--the plugin id is derived from the vendor domain name -->
<plugin
   id="com.ibm.ws.console.core"
   version="1.0.0"
   provider-name="IBM WebSphere">

  <!--declaration of prerequisite plugins-->
 <requires>
    <import plugin="com.ibm.data" version="2.0.1" match="compatible"/>
    <import plugin="com.ibm.resources" version="3.0" match="exact"/>
 </requires>

 <!--declaration of link extension point -->
 <extension-point
    id="linkExtensionPoint"
    schema="/schemas/linkSchema.xsd"/>


 <!--declaration of an extension to the link extension point -->
 <extension
    point="com.ibm.ws.console.core.linkExtensionPoint"
    id="linkExtension">

   <link
     label="Example.displayName"
     actionView="com.ibm.ws.console.servermanagement.forwardCmd.do?
        forwardName=example.config.view&
        lastPage=ApplicationServer.config.view">
   </link>
 </extension>
</plugin>
```

Task overview: Assembling applications using remote request dispatcher

Remote request dispatcher (RRD) is a pluggable extension to the Web container which allows application frameworks, servlets and JavaServer Pages (JSP) to include content from outside the currently executing resource's Java Virtual Machine (JVM) as part of the response sent to the client.

Before you begin

You must have WebSphere Application Server Network Deployment installed to use remote request dispatcher function. You should also familiarize yourself the limitations of remote request dispatcher. See article, "Remote request dispatcher considerations" on page 78 for details.

1. Installing enterprise application files with the console
2. Configure the sending of include requests between the application and remote resources.

- “Configuring Web applications to dispatch remote includes” on page 76
- “Configuring Web applications to service remote includes” on page 76

3. Optional: Modify your application to locate resources located in two different contexts using the servlet programming model.

   The Servlet Programming Model for including resources remotely does not require you to use any non-Java 2 Platform, Enterprise Edition (J2EE) Servlet Application Programming Interfaces (APIs). The remote request dispatcher (RRD) component follows the same rules to obtain a ServletContext and a remote resource. By using JavaServer Pages standard tag library (JSTL), your application is further removed from obtaining a ServletContext object or RequestDispatcher that is required in the framework example in the following step because the JSTL custom tag does this implicitly. Study the following example of a sample JavaServer Pages application to learn how to locate resources that are in two different contexts, investments and banking.

```
<HEAD>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8059-1"
isELIgnored="false"
%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" $>
</HEAD>
<BODY>

<%--

Programming example using JavaServer Pages and JavaSever Pages
Standard Tag Library (JSTL).
JSTL provides a custom tag to import contents (in servlet and JSP
terms include) in the scope of the same request from outside of
the current Web module context by specifying a context parameter.

JSTL restriction: The Web module that is imported
must run inside of the same JVM as the calling resource
if imported URL is not fully qualified.

RRD extends this functionality by permitting the Web module to
be located within the scope of the current WebSphere Application
Server core group versus the scope of the JVM.
--%>


<hr size="5"/>
<%--  Include resource investmentSummary.jsp located in the
  Web application with context root of /investments. --%>

<c:import url="investmentSummary.jsp" context="/investments"/>

<hr size="5"/>
<%--  Include resource accountSummary.jsp located in the
  Web application with context root of /banking. --%>

<c:import url="accountSummary.jsp" context="/banking"/>

<hr size="5"/>

</BODY>
</HTML>
```

4. Optional: Modify your application to locate resources located in two different contexts using the framework programming model.

   The Framework Programming Model for including resources remotely does not require you to use any non-Java 2 Platform, Enterprise Edition (J2EE) Servlet Application Programming Interfaces (APIs). When a request is initiated for a ServletContext name that is not presently running inside of the current

Web container, the remote request dispatcher (RRD) component returns a ServletContext object that can locate a resource that exists anywhere inside a WebSphere Application Sever Network Deployment environment provided that the resource exists and RRD is enabled for that ServletContext object. Study the following sample framework snippet that demonstrates how to locate resources located in two different contexts, investments and banking.

```
/*
Programming example using a generic framework.
Servlet Specification provides an API to obtain
a servlet context in the scope of the same request
different from the current Web module context by
specifying a context parameter.

Servlet Specification restriction: The Web module that obtain
must run inside of the same JVM as the calling resource.

RRD extends this functionality by permitting the Web module to be located
within the scope of the current WebSphere Application Server core group
versus the scope of the JVM.

*/

protected void frameworkCall (ServletContext context, HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException(

   PrintWriter writer = response.getWriter();

   writer.write("<HTML>");
   writer.write("<HEAD>");
   writer.write("</HEAD>");
   writer.write("<BODY>");
   writer.write("<hr size=\"5/">);

   //Include resource investmentSummary.jsp located in Web application
   //with context root of /investments.
   RequestDispatcher rd = getRequestDispatcher ( context, "/investments", "/investmentSummary.jsp");
   rd.include(request, response);

   writer.write("<hr size=\"5/">);

   //Include resource accountSummary.jsp located in Web application
   //with context root of /banking.
   rd = getRequestDispatcher ( context, "/banking", "/accountSummary.jsp");
   rd.include(request, response);

   writer.write("</BODY>");
   writer.write("</HTML>");
}
private RequestDispatcher getRequestDispatcher (ServletContext context, String contextName, String resource) {
   return context.getContext(contexName).getRequestDispatcher(resource);
}
```

Results

After enabling at least one enterprise application to dispatch remote includes and at least one enterprise application to service remote includes, RRD is now enabled.

What to do next

Restart the modified applications if already installed or start newly installed applications to enable RRD on each application.

Remote request dispatcher

Remote Request Dispatcher (RRD) is a pluggable extension to the Web container that allows application frameworks, servlets and JavaServer Pages to include content from outside of the current executing resource's Java virtual machine (JVM) as part of the response sent to the client.

Remote request dispatcher is an extensible infrastructure to allow other components and stack products to add custom extensions like generators and handlers, to the RRD extension. The remote request dispatcher extension enhances the standard J2EE `javax.servlet.RequestDispatcher` implementation to be aware of locating remote resources using Web services to communicate between machines within a Network Deployment (ND) core group. The remote request dispatcher extension reports any errors that occur on the remote server back to the originating server. It can also leverage SSL for secure communications and WS-Security security context propagation between servers. See rrdSecurity.props file for more information.

RRD portlet support carries forward the remote request dispatcher concept to portlets and enhances the portlet container to allow invocation of portlets outside of the current executing resource's JVM.

By utilizing the RRD extension, you can share request load across multiple machines and JVMs by including remote servers within the cell. If RRD resource is memory or processor intensive, the calling resource is not affected as much as a standard RequestDispatcher running within the same JVM. RRD solves this problem by separating resources into a different JVM.

Capabilities

- Requests on remote server are treated as include requests. Filters and request listeners are invoked as if the dispatch type is INCLUDE.
- Serializable request attributes and query parameters are sent to remote server.
- Security context is sent to a remote server through LTPA tokens.
- Servlet parameters and OutputStream

  Request parameters are passed to remote server.
- Response headers that are set by the remotely included resource are ignored similar to includes on a local server. Internal headers such as Set-Cookie can still be set and are propagated back.
- All original request headers are passed to remote server
  - Similar to WebSphere Application Server's plugin
  - Method calls return the state as if they are on local server. For example, getServer returns the local server name or isSecure returns whether the request to the 'local' server has been secure.
- Cookies and sessions
  - Cookies are passed to the remote server as part of headers.
  - Sessions in local and remote servers use the same cookie or session id for a given client which is similar to includes in the same server. If a session exists on a remote server, the session cookie contains the information for both the servers to maintain the affinity to the remote server.
- Exceptions
  - If there is a exception on the remote server, the server returns an RRD specific Web services fault which wraps the original exception created by the application.
  - Attempt to recreate the original exception on the local server if the exception class exists on both servers. If the original exception cannot be recreated, an RRD specific ServletException is constructed and used instead.
  - The exception is recreated by the local server for error handling purposes.
- Dynamic cache

  When dynamic cache is enabled, caching is performed on the local and remote machine.
- Security

You can use SSL to encrypt RRD messages between application servers. This is enabled by default, however, you must also pass security context needs through RRD to ensure that the security state is available in the remote machine. RRD leverages WS-Security to pass this information, but this security context propagation is disabled by default. See the topic, rrdSecurity.props file, for additional information.

Configuring Web applications to dispatch remote includes

You can configure Web modules in an application as remote request dispatcher clients to dispatch include requests to resources across Web modules that are in different Java virtual machines in a managed node environment through the standard request dispatcher.

Before you begin

You must have WebSphere Application Server for Network Deployment installed to use remote request dispatcher function. You should also familiarize yourself the limitations of remote request dispatcher. See article, “Remote request dispatcher considerations” on page 78 for details.

About this task

You can also configure this property when you install the application installation. See Installing enterprise application files with the console

1. In the administrative console, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Request dispatcher properties**.
2. Select **Allow dispatching includes to remote resources**.

Results

Web modules included in this application are enabled as remote request dispatcher clients that can dispatch remote includes.

What to do next

You must enable remote request dispatcher on both the remote server and local server to use remote request dispatcher. If you have not yet enabled remote request dispatcher on the remote server that includes the resource that you want to include, see article, “Configuring Web applications to service remote includes.”

Configuring Web applications to service remote includes

You can configure Web modules in an application as remote request dispatcher servers that are remotely included by other Web applications.

Before you begin

You must have WebSphere Application Server Network Deployment installed to use remote request dispatcher function. You should also familiarize yourself the limitations of remote request dispatcher. See article, “Remote request dispatcher considerations” on page 78 for details.

About this task

The purpose of this task is to specify whether an application can service an include request from another application. You can also configure this property when you install the application installation. Some statement about the process used to arrive here. See Installing enterprise application files with the console

1. In the administrative console, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Request dispatcher properties**.

2. Select **Allow servicing includes from remote resources**.

Results

Web modules included in this application are enabled as remote request dispatcher servers that are resolved to service remote includes from another application.

What to do next

You must enable remote request dispatcher on both the remote server and local server to use remote request dispatcher. If you have not yet enabled remote request dispatcher on the local server to include a remote resource, see article, “Configuring Web applications to dispatch remote includes” on page 76.

Configuring remote request dispatcher caching

The creation and handling of the remote request dispatcher (RRD) message and the transfer of this data across a network creates substantial overhead. To improve performance, the local machine can receive the cache rules of the remote server and know when to cache the response locally to prevent the RRD call altogether.

About this task

Unsupported rules are cached remotely. However, remote request dispatcher does not support the entire set of rules of the cachespec.xml file for local caching. The following rules for the cachespec.xml file are supported for local caching.

parameters
Retrieves the named parameter value.

cookie
Retrieves the named cookie value.

header
Retrieves the named request header.

locale
Retrieves the request locale.

requestType
Retrieves the HTTP request method from the request.

Complete the following steps to enable RRD dynamic cache support.

1. Enable servlet caching on the local server See Configuring servlet caching for more information
2. Construct and install an application with a valid `cachespec.xml file` policy with RRD supported rules. See the article, Configuring cacheable objects with the cachespec.xml file, for additional information.
3. Enable servlet caching on the remote server. See Configuring servlet caching for more information
4. Restart WebSphere Application Server. See Managing application servers for more information.

Results

To verify that dynamic cache is enabled locally, you can enable RRD trace and verify that there is a CACHE HIT printed out for multiple requests to a resource that has an appropriate cache policy. You also have the option to use CacheMonior instead of turning on tracing. If the CacheMonitor on the remote machine is not receiving cache hits, then the response is cached locally. The local CacheMonitor will not receive hits because RRD uses its own custom cache.

Remote dispatcher property settings

Use this page to configure the sending of include requests between the application and remote resources.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Request dispatcher properties**.

Allow dispatching includes to remote resources

Specifies whether an application can dispatch include requests to resources across Web modules that are in different Java virtual machines in a managed node environment through the standard request dispatcher.

Data type	Boolean
Default	false

Allow servicing includes from remote resources

Specifies whether an application can service an include request from another application.

Data type	Boolean
Default	false

Asynchronous request dispatching type

Changes the behavior of the servlet request dispatcher to support cross-JVM dispatching and asynchronous dispatching.

Disabled	Specifies that asynchronous request dispatching is not supported for this application
Server side	Specifies that the response content is aggregated on the server
Client side	Specifies that the response content is aggregated on the client/browser

Remote request dispatcher considerations

This topic presents some considerations of which you need to be aware when using remote request dispatcher.

- If an application expects parameters in certain encoding, the application should set character encoding, as in a normal include, before the remote request dispatch (RRD) occurs.
  - ServletInputStream data of local server is not available to remote server. The local server parses POST data prior to sending the RRD request to the remote server and include parameters as request parameters. Multipart form data is inaccessible to remote server. An UnsupportedOperationException is created if the remote server attempts to obtain inputstream from the request.
  - No access to original request reference on the remote server.
  - Request and response wrappers created in the local server are not available in remote server. This cannot be done due to ServletRequestWrappers and WebSphere internal ServletRequest objects not implementing Serializable.
- Request attributes need to be serializable.
  - Class definition of attributes need to be available in both local and remote servers.
  - Request attributes are propagated to the remote server and back to the local server.
- HTTP Sessions
  - You cannot have cross session access between different Web applications when Web applications are remote.

  - When all Web applications are in a local server, an application can share sessions across Web applications by storing the session in a table that is accessible to multiple Web applications. This is not possible with RRD and not recommended in local case either.
  - Servlet Programming Model: You cannot access sessions in different Web applications.
  - Normal programming model in local case as well as remote case.
  - In local mode, application can cache away reference and share the session across the Web applications, which is not feasible in RRD case.
  - A session object that is stored as request attribute is not available on remote server as the Session class does not implement Serializable.
- Thread local variables that are set on the local server are not available on the remote server.
- Not all methods defined on ServletContext object are available for the RRD ServletContext object. See the SPI documentation for com.ibm.wsspi.rrd.context.RemoteServletContext for details.
- The remote server does not have access to output of local server when using RRD.
- Cookies and ServletRequestWrappers

  If customer application wrappers the HttpServletRequest.getCookies method and returns additional cookies or removes cookies, the modified cookies are not sent to the remote server because javax.servlet.http.Cookie does not implement Serializable. Cookies from the original request headers are sent to the remote server.

Servlet extension interfaces

You can use extension generators and handlers to add content to the remote message for servlet and portlets calls. The servlet extension can also modify existing behavior by leveraging the filter concept. The Remote Request Dispatcher (RRD) extension framework relies on extension generators, which attach arbitrary data to an outbound RRD request, and extension handlers, which consume the data and perform actions based on the data received.

- " Extension generators"
- "Extension handlers" on page 80
- "Extension delegators" on page 81
- "Custom EMF packages" on page 81

For more information on the `com.ibm.wsspi.rrd.extension` package, see Additional Application Programming Interfaces (APIs) for administrators.

Extension generators

Extension generators, which are part of an extension generator chain, are invoked prior to initiating an RRD request. This extension generator chain is defined in the com.ibm.wsspi.rrd.generators extension point of the `plugin.xml` file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the `WEB-INF` directory of an RRD-enabled local Web application.

Each extension generator is defined by a generator element, which contains an id attribute, which is used to assign a unique identifier to the extension generator such that any extension generator can be targeted by extensions added to RRD response data. The class attribute is used to specify the class name of the extension generator, which must implement the `com.ibm.wsspi.rrd.extension.generator.ExtensionGenerator` interface. Each extension generator can also have an execution order associated with it via the order attribute, which is useful for enforcing extension generator execution order in an environment where multiple extension generator descriptor files are present. Additionally, a generator has a mandatory attribute called type that defines the type of the generator. For servlet RRD, the value is "servlet" and the class must implement the `com.ibm.wsspi.rrd.extension.generator.ExtensionGenerator` interface.

Additionally, each extension generator may be provided with an arbitrary number of initialization parameters, which are specified by including zero or more `init-param` elements as children of the generator element. An example extension generator declaration follows:

```
<extension point="com.ibm.wsspi.rrd.generators">
  <generator id="int1"
        class="com.ibm.ws.rrd.example.extension.IntExtensionGenerator"
        order="1"
        type="servlet">
    <init-param>
            <param-name>intValue</param-name>
            <param-value>100</param-value>
    </init-param>
   </generator>
   <generator id="string1"
        class="com.ibm.ws.rrd.example.extension.StringExtensionGenerator"
        order="2"
        type="servlet">
      <init-param>
            <param-name>stringValue</param-name>
            <param-value>This is an example string</param-value>
      </init-param>
   </generator>
   <generator id="int2"
        class="com.ibm.ws.rrd.example.extension.IntExtensionGenerator"
        order="3"
        type="servlet">
      <init-param>
            <param-name>intValue</param-name>
            <param-value>200</param-value>
      </init-param>
   </generator>
</extension>
```

For more information on the `com.ibm.wsspi.rrd.extension.generator` package, see Additional Application Programming Interfaces (APIs) for administrators.

Extension handlers

Extension handlers, which are part of an extension handler chain, are invoked after an RRD request has been received. This extension handler chain is defined in the `com.ibm.wsspi.rrd.handlers` extension point of the `plugin.xml` file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the `WEB-INF` directory of an RRD-enabled local Web application.

Each extension handler is defined by a handler element, which contains namespaceURI and localName attributes, the combination of which defines the qualified name of the extension data that the extension handler can process. Each extension handler additionally requires a unique identifier, specified by the id attribute. The value specified by this attribute must correspond to an extension generator, which generates extension data of a matching qualified name and identifier. The class attribute is used to specify the class name of the extension handler, which must implement the com.ibm.wsspi.extension.handler.ExtensionHandler interface. Additionally a handler has a mandatory attribute called type that defines the type of the handler. The value is “servlet” and the class must implement the com.ibm.wsspi.rrd.extension.handler.ExtensionHandler interface.

Additionally, each extension handler may be provided with an arbitrary number of initialization parameters, which are specified by including zero or more `init-param` elements as children of the handler element. An example extension handler declaration follows:

```
  <extension point="com.ibm.wsspi.rrd.handlers">
      <handler id="int1"
               class="com.ibm.ws.rrd.example.extension.IntExtensionHandler"
               namespaceURI="http://www.ibm.com/ws/rrd/ext/types"
               localName="SimpleType" order="1"
               type="servlet"/>
      <handler id="string1"
               class="com.ibm.ws.rrd.example.extension.StringExtensionHandler"
               namespaceURI="http://www.ibm.com/ws/rrd/ext/types"
               localName="SimpleType" order="2"
               type="servlet"/>
      <handler id="int2"
               class="com.ibm.ws.rrd.example.extension.IntExtensionHandler"
               namespaceURI="http://www.ibm.com/ws/rrd/ext/types"
               localName="SimpleType" order="3"
               type="servlet"/>
</extension>
```

For more information on the `com.ibm.wsspi.rrd.extension.handler` package, see Additional Application Programming Interfaces (APIs) for administrators.

Extension delegators

An extension delegator enables RRD to handle arbitrary servlet containers by allowing users to specify the specific extension generator and handler chain instances that are to be used during an RRD call. RRD maintains a user-extendable list of extension delegators and selects an appropriate delegator at runtime based on the type of servlet request being issued.

Custom extension delegators may be defined in the `com.ibm.wsspi.rrd.rrd-extension-delegator` extension point of the `plugin.xml` file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the WEB-INF directory of an RRD-enabled local Web application.

Each extension delegator is defined by an `ExtensionDelegator` element, which contains a priority attribute for defining the relative order in which an extension delegator is initiated, and a classname attribute that defines the implementing class for a particular extension delegator, which must implement the `com.ibm.wsspi.rrd.extension.factory.ExtensionDelegator` interface. Note that the execution order of two or more extension delegators with the same priority is not predictable. An example extension delegator declaration follows:

```
<extension point="com.ibm.wsspi.rrd.rrd-extension-delegator">
  <ExtensionDelegatorRegistration>
    <ExtensionDelegator priority="1" classname="com.ibm.ws.rrd.extension.PortletExtensionDelegator"/>
    <ExtensionDelegator priority="2" classname="com.ibm.ws.rrd.extension.ServletExtensionDelegator"/>
  </ExtensionDelegatorRegistration>
</extension>
```

Custom EMF packages

Extension data produced by an extension generator and consumed by an extension handler is serialized using the Eclipse Modeling Framework (EMF). Users intending to use custom extension data should utilize the `com.ibm.wsspi.rrd.rrd-emf-packages` extension point in order to ensure that the proper EMF packages are initialized prior to use by RRD. This extension point is part of the plugin.xml file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the `WEB-INF` directory of an RRD-enabled local Web application.

Each EMF package is defined by an `emfPackage` element, which contains a className element that must point to the generated EMF factory implementation class for a particular EMF package (the generated model class which implements `org.eclipse.emf.ecore.impl.EFactoryImpl`). An example EMF package declaration follows:

```
<extension point="com.ibm.wsspi.rrd.rrd-emf-packages">
  <emfPackages>
    <emfPackage className="com.ibm.ws.rrd.webservices.types.emf.impl.TypesFactoryImpl" />
  </emfPackages>
</extension>
```

Note: If the same generated EMF model code is shared among multiple Web applications that the EMF model code must be part of a shared server library, but only in the case that all Web applications are running in the same application server. In production, this is usually not the case, and common EMF model code may exist at the Web application level.

Related tasks

"Task overview: Assembling applications using remote request dispatcher" on page 72
Remote request dispatcher (RRD) is a pluggable extension to the Web container which allows application frameworks, servlets and JavaServer Pages (JSP) to include content from outside the currently executing resource's Java Virtual Machine (JVM) as part of the response sent to the client.

Chapter 2. Portlet applications

Task overview: Managing portlets

You can use this task to manage deployed portlet applications.

Before you begin

Before you begin this task, you must have a portlet application installed. See Installing enterprise application files for additional information.

About this task

You can complete the following steps to manage portlets.

- Render a portlet.
  - Access a single portlet using “Portlet Uniform Resource Locator (URL) addressability” on page 90.
  - Access multiple portlets using “Portlet aggregation using JavaServer Pages” on page 84.
  - Access portlets using “Remote request dispatcher” on page 75.
- Change the location of “Portlet preferences” on page 92. By default, portlet preferences for each portlet window are stored in a cookie. However, you can change the location of where to store portlet preferences.
- Disable URL addressability. By default, you can access a portlet through an Uniform Resource Locator (URL), however, you can disable this feature.
- Enable portlet fragment caching. Portlet fragment caching is disabled by default.

Portlets

Portlets are reusable Web modules that provide access to Web-based content, applications, and other resources. Portlets can run on WebSphere Application Server because it has an embedded JSR 286 Portlet container. The JSR 286 API provides backwards compatibility. You can assemble portlets into a larger portal page, with multiple instances of the same portlet displaying different data for each user.

From a user’s perspective, a portlet is a window on a portal site that provides a specific service or information, for example, a calendar or news feed. From an application development perspective, portlets are pluggable Web modules that are designed to run inside a portlet container of any portal framework. You can either create your own portlets or select portlets from a catalog of third-party portlets.

Each portlet on the page is responsible for providing its output in the form of markup fragments to be integrated into the portal page. The portal is responsible for providing the markup surrounding each portlet. In HTML, for example, the portal can provide markup that gives each portlet a title bar with minimize, maximize, help, and edit icons.

You can also include portlets as fragments into servlets or JavaServer Pages files. This provides better communication between portlets and the Java Platform, Enterprise Edition (Java EE) Web technologies provided by the application server.

If you use Rational Application Developer version 6 (RAD) to create your portlets, you must remove the following reference to the std-portlet.tld from the web.xml file to run the portlets outside of RAD:

```
<taglib id="PortletTLD">
 <taglib-uri>http://java.sun.com/portlet</taglib-uri>
 <taglib-location>/WEB-INF/tld/std-portlet.tld</taglib-location>
</taglib>
```

Also if you use RAD version 6 to create portlets, note that portlets created by using the Struts Portlet Framework are not supported on WebSphere Application Server.

Portlet applications

If the portlet application is a valid Web application written to the Java Portlet API, the portlet application can operate on both the Portal Server and the WebSphere Application Server without requiring any changes. JSR 168 and JSR 286 compliant portlet applications must not use extended services provided by WebSphere Portal to operate on the WebSphere Application Server.

Portlet container

The *portlet container* is the runtime environment for portlets using the JSR 286 Portlet specification, in which portlets are instantiated, used, and finally destroyed. The JSR 286 Portlet API provides standard interfaces for portlets and backwards compatibility for JSR 168 portlets. Portlets based on this JSR 286 Portlet Specification are referred to as standard portlets.

A simple portal framework is provided by the PortletServingServlet servlet. The PortletServingServlet servlet registers itself for each Web application that contains portlets. You can use the PortletServingServlet servlet to directly render a portlet into a full browser page by a URL request and invoke each portlet by its context root and name. See “Portlet Uniform Resource Locator (URL) addressability” on page 90 for additional information. If you want to aggregate multiple portlets on the page, you need to use the aggregation tag library. See the article “Portlet aggregation using JavaServer Pages” for additional information. The PortletServingServlet servlet can be disabled in an extended portlet deployment descriptor called the ibm-portlet-ext.xmi file.

Remote request dispatcher support for portlets

The remote request dispatcher (RRD) support allows the invocation of portlets outside of the current Java virtual machine (JVM) within an Network Deployment single core group environment. The request related data is passed to the remote JVM where the portlet is invoked. The response is transmitted back and processed on the local JVM. Thus it guarantees that URLs contained in the portlet markup are created according to the local portal context. The remote request dispatcher support is only provided for JSR 168 compliant portlets.

Portlet container settings

Use this page to configure and manage the portlet container of this application server.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Portlet Container Settings** → **Portlet container**.

Enable portlet fragment cache

Specifies whether to create a cached entry when a portlet is invoked, similar to servlet caching of the Web container settings.

Portlet fragment caching requires that servlet caching is enabled. Therefore, enabling portlet fragment caching automatically enables servlet caching. Disabling servlet caching automatically disables portlet fragment caching.

Portlet aggregation using JavaServer Pages

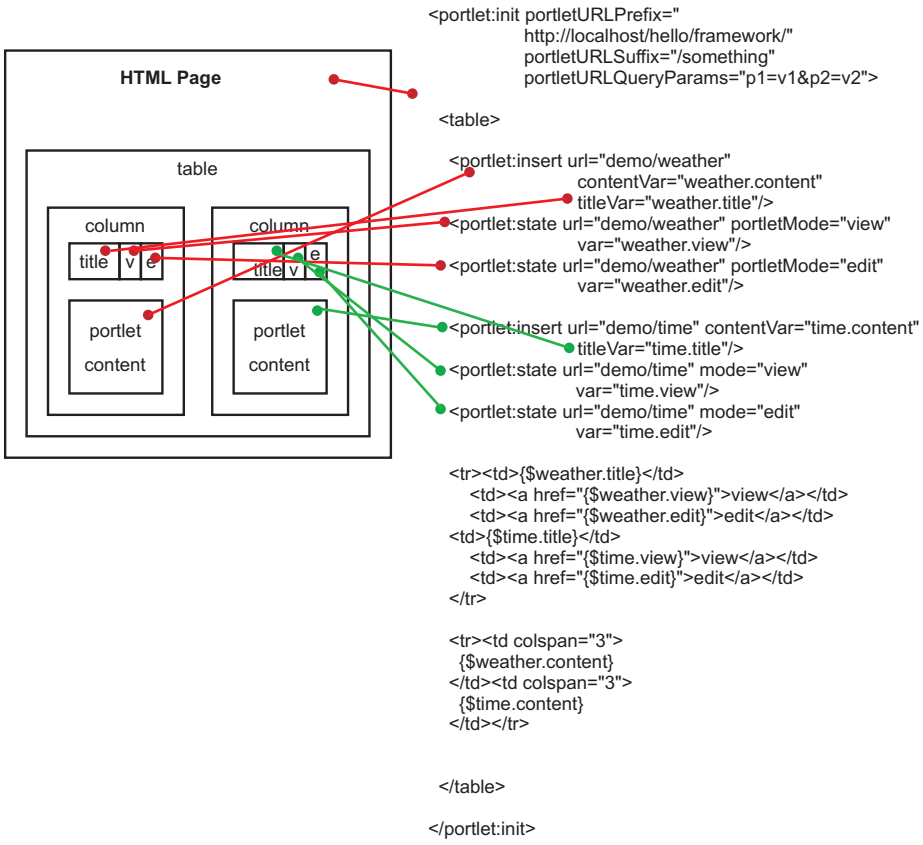
The aggregation tag library generates a portlet aggregation framework to address one or more portlets on one page If you write JavaServer Pages, you can aggregate multiple portlets on one page using the aggregation tag library. This tag library does not provide full featured portal aggregation implementation, but provides a good migration scenario if you already have aggregating servlets and JavaServer Pages and want to switch to portlets.

To allow the customer to create a simple portal aggregation, the aggregation tag library also provides the following features.

- Invoke a portlet's action method
- Render multiple portlets on one page
- Provide links to change the portlet's mode or window state
- Display the portlet's title
- Retain the portlet cookie state

The aggregation tag library and JavaServer Pages that use the aggregation tag library will only work with the WebSphere Application Server portlet container implementation because the protocol between the tags and the container is not standardized.

The following diagram depicts how an HTML page would look like and what tags are used in order to create the page. See "Aggregation tag library attributes" on page 86 for information on the aggregation tag library attributes.



When you use the aggregation tag library, you must set the portletUrlPrefix attribute of the init tag to the aggregating application. You need to:

- Ensure that the portletUrlPrefix attribute is set to the following in the aggregator page.

  `"http://" + <server_address> + ":" + <server_port> + "/" + <aggregator context> + "/" <aggregator mapping>`
- Reference the aggregation JSP page within the web.xml file through a servlet mapping ending with `/*`. For example, `/aggregation/*`

When aggregating multiple portlets on a single page, special care must be used with the naming conventions of form attribute names in your portlets. Because your portlets are all on the same page, they all share the same HttpServletRequest. When one portlet is viewed the entire page is refreshed and form

data is re-posted. Therefore, if there are multiple portlets that are aggregated on a single page with the same form attribute names, there could be logic corruption when form data is re-posted.

Aggregation tag library attributes

The aggregation tag library is used to aggregate multiple portlets on one page. This topic describes the attributes within the aggregation tag library.

Supported arguments include:

init

This tag initializes the portlet framework and has to be used in the beginning of the JSP. All other tags described in this section are only valid in the body of this tag, therefore the init tag usually encloses the whole body of a JSP. In case the current URL contains an action flag the action method of the corresponding portlet is called. The state and insert tags are sub-tags of the init tag.

The init tag has the following attributes:

- portletURLPrefix = ”<any string>”

  This URL defines the prefix used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet’s render method, which is called by using the insert tag. This is a required attribute.
- portletURLSuffix = ”<any string>”

  This URL defines the suffix used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet’s render method, which is called by using the insert tag. This is attribute optional.
- portletURLQueryParams = ”<any string>”

  This URL defines the query parameters used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet’s render method, which is called by using the insert tag. This is attribute optional.

scope, portlet

The scope tag and portlet tag are used to provide information that is necessary when a portlet application is installed under a multiple part context root, for example, /context1/context2. These tags add a render parameter to the newly created URL.

The urlParam tag has the following attributes:

- context = ″/<context1>/<context2>″

  Specifies the context root of the portlet application in which the portlet is deployed. This attribute is required.
- portletname = ″<portlet-name>″

  Specifies the portlet-name. This attribute is required.
- windowId = ”<any string>”

  Defines the window ID for the concrete portlet instance. This attribute is required.

The following is an example of how to use the scope and portlet tags:

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>

<portlet:scope>
         <portlet:portlet context="/myportletcontext1/myportletcontext2" portletname="MyPortlet" windowId="sample"/>
</portlet:scope>

<portlet:init portletURLPrefix="/myportalcontext/ ">
....
</portlet:init>
```

state

The state tag creates a URL pointing to the given portlet using the given state. You can place this URL either into a variable specified by the var attribute or you can write it directly to the output stream. This

tag is useful to create URLs for HTML buttons, images, and other items such that when the URL is invoked, the state changes defined in the URL are applied to the given portlet.

The state tag has the following attributes:

- url = ”<context>/<portlet-name>”

  Identifies the portlet for this tag by using the context and portlet-name to address the portlet. This attribute is required.
- windowId = ”<any string>”

  Defines the window ID for the portlet URL created by this tag. This is attribute optional.
- var = ”<any string>”

  If defined the URL is written into a variable with the given scope and name, not to the output stream. This is attribute optional.
- scope = ”page|request|session|application”

  This attribute is only valid if the var attribute is specified. If defined, the URL is not written to the output stream but a variable is created in the given scope with the given name. The default is page. This is attribute optional.
- portletMode = ”view|help|edit|<custom>”

  This attribute sets the portlet mode.
- portletWindowState = ”maximized|minimized|normal|<custom>”

  This attribute sets the window state.
- action = ”true/false”

  This attribute defines whether this is an action URL. This is attribute optional. The default is false.

urlParam

Adds a render parameter to the newly created URL.

The urlParam tag has the following attributes:

- name = ”<any string>”

  Indicates the name of the parameter. This is attribute required.
- value = ”<any string>”

  Indicates the value of the parameter. This is attribute required.

insert

This tag calls the render method of the portlet and retrieves the content as well as the title. You can optionally place the content and title of the specified portlet into variables using the contentVar and titleVar attributes.

The insert tag has the following attributes:

- url = ”<context>/<portlet-name>” (mandatory) Identifies the portlet for this tag by using the context and portlet-name to address the portlet

  This is attribute required.
- windowId = ”<any string>”

  Defines the window ID of the portlet. This is attribute optional.
- contentVar = ”<any string>”

  If defined, the portlet’s content is not written to the output stream but written into a variable with the given scope and name. This is attribute optional.
- contentScope = ”page|request|session|application”

  This attribute is only valid if the contentVar tag is used. If defined, the portlet’s content is written into a variable with the given scope and name, not to the output stream. The default is page. This is attribute optional.
- titleVar = ”<any string>”

If defined the portlet's title is written into a variable with the given scope and name. If it is not defined, the title is ignored and not written to the output stream. This is attribute optional.

- titleScope = "page|request|session|application"

  This attribute is only valid if titleVar tag is used. If defined, the portlet's title is written into a variable with the given scope and name, not to the output stream. The default is page. This is attribute optional.

Example: Using the portlet aggregation tag library

You can use the aggregation tag library to aggregate multiple portlets to have multiple and different content on one page. The library can be used by every JavaServer Pages (JSP) file that has been included by a servlet.

To use the portlet aggregation tag library, you must declare the tag-lib at the top of the JSP file using, `<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>`, as in the following example. The following JSP file example shows how to aggregate portlets on one page.

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>
<%@ page isELIgnored="false" import="java.util.Enumeration"%>

<portlet:init portletURLPrefix="/dummy/portletTagTest/" portletURLSuffix="/more" portletURLQueryParams="p1=v1&p2=v2">


   <portlet:insert url="worldclock/StdWorldClock" contentVar="worldclockcontent" titleVar="worldclocktitle"/>
    <portlet:state url="worldclock/StdWorldClock" portletMode="view" var="worldclockview"
       portletWindowState="maximized">
        <portlet:urlParam name="namea" value="valuea"/>
        <portlet:urlParam name="nameb" value="valueb"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" portletMode="edit" var="worldclockedit" portletWindowState="normal">
        <portlet:urlParam name="name1" value="value1"/>
        <portlet:urlParam name="name2" value="value2"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" portletMode="view" var="worldclockmin"
       portletWindowState="minimized">
        <portlet:urlParam name="namemin" value="valuemin"/>
        <portlet:urlParam name="namemin" value="valuemin"/>
    </portlet:state>

    <portlet:insert url="worldclock/StdWorldClock" windowId="min" contentVar="simplecontent" titleVar="simpletitle"/>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="view" var="simpleview"
  portletWindowState="maximized">
        <portlet:urlParam name="name3" value="value3"/>
        <portlet:urlParam name="name4" value="value4"/>
        <portlet:urlParam name="name5" value="value5"/>
        <portlet:urlParam name="name5" value="value5a"/>
        <portlet:urlParam name="name5" value="value5b"/>
        <portlet:urlParam name="name5" value="value5c"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="edit" var="simpleedit"
  action="true" portletWindowState="normal">
        <portlet:urlParam name="name6" value="value6"/>
        <portlet:urlParam name="name6" value="value6z"/>
            </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="view" var="simplemin"
  portletWindowState="minimized">
        <portlet:urlParam name="name1" value="value1"/>
        <portlet:urlParam name="name2" value="value2"/>
    </portlet:state>

    <portlet:insert url="test/TestPortlet1" contentVar="testcontent" titleVar="testtitle"/>
    <portlet:state url="test/TestPortlet1" portletMode="view" var="testview" portletWindowState="maximized"/>
    <portlet:state url="test/TestPortlet1" portletMode="edit" var="testedit" portletWindowState="maximized"/>
```

```
<!--  This table is the outtermost table for creating two-column portal layout  -->
<TABLE border="0" CELLPADDING="3" CELLSPACING="8" WIDTH="100%">
<TR>
<TD VALIGN="top">

<!--  This table is the top portlet in the first column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet top left">
    <tr><td class="portletTitle" NOWRAP>worldclock title:${worldclocktitle}</td>
        <td CLASS="portletTitleControls"  NOWRAP>
            <a href="${worldclockview}">view</a>
            <a href="${worldclockedit}">edit</a>
            <a href="${worldclockmin}">minimize</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${worldclockcontent}
     </td>
    </tr>
  </table>

  <BR/>

<!--  This table is the bottom portlet in the first column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet bottom left">
    <tr>
    <td class="portletTitle" NOWRAP>test title:${testtitle}</td>
        <td CLASS="portletTitleControls" NOWRAP>
            <a href="${testview}">view</a>
            <a href="${testedit}">edit</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${testcontent}
     </td>
    </tr>
  </table>


  </TD>



  <TD VALIGN="top">


<!--  This table is the top portlet in the second column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet top right">
  <tr>
        <td class="portletTitle" NOWRAP>simple title:${simpletitle}</td>
        <td CLASS="portletTitleControls" NOWRAP>
            <a href="${simpleview}">view</a>
            <a href="${simpleedit}">edit</a>
            <a href="${simplemin}">minimize</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${simplecontent}
     </td>
    </tr>
  </table>
```

```
    </TD>
    </TR>
    </table>


</portlet:init>
```

You can include the following formatting to the previous example JSP file immediately after declaring the tag library.

```
<STYLE TYPE="TEXT/CSS">
BODY {
    font-family:Verdana,sans-serif; font-size:70%
}
.portletTitle {
    text-align: left;border-top: #000000 1px solid; border-bottom: #000000 1px solid; FONT-SIZE: 60.0%;
    COLOR: #ffffff; FONT-FAMILY: Verdana, Arial, Helvetica, sans-serif; BACKGROUND-COLOR: #5495d5;
}
.portletTitleControls {
    text-align: right;border-top: #000000 1px solid; border-right: #000000 1px solid; border-bottom: #000000
    1px solid; FONT-SIZE: 60.0%; COLOR: #ffffff; FONT-FAMILY: Verdana, Arial, Helvetica, sans-serif;
    BACKGROUND-COLOR: #5495d5;
}
.portletTitleControls A {
    COLOR: #ffffff; text-decoration:none; border:#5495d5 1px solid;border-left:white 1px solid;
    padding-left:0.5em; padding-right:0.5em;
}
.portletTitleControls A:hover {
    COLOR: #ffffff; text-decoration:none; border-top:white 1px solid;
   border-bottom:white 1px solid;border-right:white 1px solid;
}
.minimizeControl {
    font-weight:bold; font-size:100%;
}
.portletTable {
    border-left: gray 1px solid;
    border-bottom: gray 1px solid;
    border-right: gray 1px solid;
}
.portletBody {
    font-family:Verdana,sans-serif; font-size:70%
}

</STYLE>
```

Portlet Uniform Resource Locator (URL) addressability

You can request a portlet directly through a Uniform Resource Locator (URL) to display its content without portal aggregation. The PortletServingServlet servlet registers each Web application that contains portlets. It is similar to the FileServingServlet servlet of the Web container that serves resources. The PortletServingServlet servlet supports direct rendering of portlets into a full browser page by a URL request.

You can invoke each portlet by its context root and name with the URL mapping /<portlet-name> that is created for each portlet. The context root and name has the following format:

```
http://<host>:<port>/<context-root>/<portlet-name> For example,
 http://localhost:9080/portlets/TestPortlet1
```

The context root identifies the Web archive (WAR) file that contains the portlet. The portlet name uniquely identifies the portlet with a portlet application of a WAR file. The DefaultDocumentFilter servlet only supports HTML, UTF8 encoding and an extended URL form based on the basic URL form as shown above.

You can only display one portlet at a time using the PortletServingServlet servlet. If you want to aggregate multiple portlets on the page, you need to use the aggregation tag library. See the article “Portlet aggregation using JavaServer Pages” on page 84 for additional information.

Because a portlet only delivers fragment output whereas a servlet usually delivers document output, a mechanism is introduced to convert the fragment into a document, called the PortletDocumentFilter filter. By default, the PortletDocumentFilter filter only supports converting HTML. The conversion is implemented using a servlet filter before the PortletServingServlet servlet is initiated to return the portlet‘s content inside of a document. This default document servlet filter only applies to URL requests, not for includes or forwards using the RequestDispatcher method. You can create servlet filters to support other markups additional document servlet filters. See the article, Converting portlet fragments to an HTML document, for additional information.

The PortletServingServlet servlet does not persist portlet preferences in a XML file or database. It places the portlet preferences directly into a cookie to store the preferences persistently. See the article, “Portlet preferences” on page 92, for additional information on how to change this behavior.

Portlet URL syntax

You can add additional portal context such as portlet mode or window state. You can access the PortletServingServlet servlet by using a URL mapping that has the following structure:

```
http://host:port/context/portlet-name [/portletwindow[/ver [/action |
 /resource[/id=custom-id][/cacheability]] [/mode] [/state] [rparam][/?name]]]
```

Any differing URL structure results in a com.ibm.wsspi.portletcontainer.InvalidURLException exception. Empty strings are not permitted as parameter values and creates an InvalidURLException exception. The following is a list of valid parameters:

http:// host:port/context/*portlet-name*
: This is the minimum URL required to access a portlet. A default portlet window called ‘default’ is created. The *portlet-name* variable is case-sensitive.

/*portletwindow*
: This parameter identifies the portlet window. You must set this parameter if you choose to add more portal context information to the URL.

/ver=*major.minor*
: This optional parameter is used to define the version of the portlet API that is used. You must set this parameter if you choose to add more portal context information to the URL. Only versions ’1.0’ and ’2.0’ are supported. Any other version creates an InvalidURLException exception.

/action
: This is a required parameter if you call the action method of the portlet. The action parameter causes the action process of the portlet to be called. After the action has been completed, a redirect is automatically issued to call the render process. To control the subsequent render process, a document servlet filter can set a request attribute with name ’com.ibm.websphere.portlet.action’ and value ’redirect’ to specify that the portlet serving servlet directly returns after action without calling the render process.

/mode=view | edit | help | *custom-mode*
: This optional parameter defines the portlet mode that is used to render the portlet. The default mode is ‘view’. The value is not case-sensitive. For example, ‘View’, ‘view’ or ‘VIEW’ results in the same mode.

/state=normal | maximized | minimized | *custom-state*
: This optional parameter defines the window state that is used to render the portlet. The default state is ‘normal’. The value is not case-sensitive, for example, ‘Normal’, ‘normal’, or ‘NORMAL’ results in the same state.

*** [/rparam=***name* ***[=***value***]]**
This optional parameter specifies render parameters for the portlet. Repeat this parameter chain to provide more than one render parameter. For example, /rparam=invitation/ rparam=days=Monday=Tuesday.

?name=*value***&***name2***=***value2* **...**
Query parameters may follow optionally. They are not explicitly supported by the portlet container, but they do not invalidate the URL format.

/action | /resource
This parameter defines the methods of the portlet that is called. Valid values are no, action or resource parameter. No specific method defined calls the render method. The resource parameter is only supported for JSR 286 portlets.

/resource [/id=custom-id] [/cacheability=cacheLevelFull | cacheLevelPortlet | cacheLevelPage]
Set this parameter to define the method of the portlet to be called. No redirection occurs. No other method of the portlet is called. To control the resource parameter, you can add an additional ID parameter to provide a resource serving identifier that is passed through to the portlet. The cacheability parameter defines the cache level of this resource URL. This parameter is only supported with JSR 286 portlets .

The following list includes examples of valid JSR 168 and JSR 286 URLs:

- http:// localhost:9080/sample/WorldClock
- http:// localhost:9080/sample/WorldClock/myPortlet/ver=1.0/mode=edit/rparam=timezone=UTC
- http:// localhost:9080/sample/WorldClock/myPortlet/ver=1.0/action/state=maximized?timezone=UTC
- http://localhost:9080/sample/WorldClock/myPortlet/ver=2.0/resource/id=somePicture.jpg

Portlet preferences

Preferences are set by portlets to store customized information. By default, the PortletServingServlet servlet stores the portlet preferences for each portlet window in a cookie. However, you can change the location to store them in either a session, an .xml file, or a database.

Storing portlet preferences in cookies

The attributes of the cookie are defined as follows:

Path
context/*portlet-name*/*portletwindow*

Name:
The name of the cookie has the fixed value of **PortletPreferenceCookie**.

Value
The value of the cookie contains a list of preferences by mapping to the following structure:

```
*['/' pref-name *['=' pref-value]]
```

All preferences start with '/' followed by the name of the preference. If the preference has one or more values, the values follow the name separated by the '=' character. A null value is represented by the string '#*!0_NULL_0!*#'. As an example, the cookie value may look like, /locations=raleigh=boeblingen/ regions=nc=bw

Customizing the portlet preferences storage

You can override how the cookie is handled to store preferences in a session, an .xml file or database. To customize the storage, you must create a filter, servlet or JavaServer Pages file as new entry point that

wraps the request and response before calling the portlet. Examine the following example wrappers to understand how to change the behavior of the PortletServingServlet to store the preferences in a session instead of cookies.

The following is an example of how the main servlet manages the portlet invocation.

```
public class DispatchServlet extends HttpServlet
{
    ...
    public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
 response.setContentType("text/html");

 // create wrappers to change preference storage
 RequestProxy req = new RequestProxy(request);
 ResponseProxy resp = new ResponseProxy(request, response);

 // create url prefix to always return to this servlet
                ...
                req.setAttribute("com.ibm.wsspi.portlet.url.prefix", urlPrefix);

 // prepare portlet url
 String portletPath = request.getPathInfo();
                ...

 // include portlet using wrappers
 RequestDispatcher rd = getServletContext().getRequestDispatcher(modifiedPortletPath);
                rd.include(req, resp);
    }
}
```

In the following example, the request wrapper changes the cookie handling to retrieve the preferences out of the session.

```
public class RequestWrapper extends HttpServletRequestWrapper
{
    ...
    public Cookie[] getCookies() {
        Cookie[] cookies = (Cookie[]) session.getAttribute("SessionPreferences");
        return cookies;
    }
}
```

In the following example, the response wrapper changes the cookie handling to store the preferences in the session:

```
public class ResponseProxy extends HttpServletResponseWrapper
{
   ...
    public void addCookie(Cookie cookie) {
        Cookie[] oldCookies = (Cookie[]) session.getAttribute("SessionPreferences");
        int newPos = (oldCookies == null) ? 0 : oldCookies.length;
        Cookie[] newCookies = new Cookie[newPos+1];
        session.setAttribute("SessionPreferences", newCookies);

        if (oldCookies != null) {
        System.arraycopy(oldCookies, 0, newCookies, 0, oldCookies.length);
        }
        newCookies[newPos] = cookie;
    }
}
```

Example: Configuring the extended portlet deployment descriptor to disable PortletServingServlet

Portlet URL serving supports direct access to all functions and states of a portlet by creating the appropriate URLs. In a production setup where the portlet is served through an enterprise portal application that applies its own access control, is considered a security risk. By setting the portletServingEnabled property to false, an administrator can ensure that a sensitive portlet is never accessed by direct URL serving.

Extensions for the portlet deployment descriptor are defined within a file called ibm-portlet-ext.xmi. This deployment descriptor is an optional descriptor that you can use to configure WebSphere extensions for the portlet application and its portlets. For example, you can disable the PortletServingServlet servlet for the portlet application in the extended portlet deployment descriptor.

The ibm-portlet-ext.xmi extension file is loaded during application startup. If there are no extension files specified with this setting, the default values of the portlet container are used.

The default for the portletServingEnabled attribute is true. The following is an example of how to configure the application so that a PortletServingServlet servlet is not created for any portlet on the portlet application.

```
<?xml version="1.0" encoding="UTF-8"?>
<portletappext:PortletApplicationExtension xmi:version="1.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:portletappext="portletapplicationext.xmi"
    xmlns:portletapplication="portletapplication.xmi"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmi:id="PortletApp_ID_Ext"
    portletServingEnabled="false">
  <portletappext:portletApplication href="WEB-INF/portlet.xml#myPortletApp"/>
</portletappext:PortletApplicationExtension>
```

Portlet container custom properties

You can configure name-value pairs of data, where the name is a property key and the value is a string value that you can use to set internal system configuration properties. Defining a new property enables you to configure a setting beyond that which is available in the administrative console. The following is a list of the available Portlet container custom properties.

To specify Portlet container custom properties:

1. In the administrative console click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Portlet Container Settings** → **Portlet container**.
2. Under **Additional Properties** select **Custom Properties**.
3. On the Custom Properties page, click **New**.
4. On the settings page, enter the name of the custom property that you want to configure in the **Name** field and the value that you want to set it to in the **Value** field.
5. Click **Apply** or **OK**.
6. Click **Save** on the console task bar to save your configuration changes.
7. Restart the server.

Following is a list of custom properties provided with the Application Server.

Using short names for Portlet and PortletApplication MBeans

Portlet MBeans are registered by both their short name and full name. To enable the use of short MBean names, create the following name-value pair:

Name	Value
useShortMBeanNames	true

The default is false.

MBeans registered by the full identifiable name, have the following format:

```
<ApplicationName>#<WARfilename.war>_portlet.<portlet_name> for the Portlet MBean
<ApplicationName>#<WARfilename.war>_portlet for the PortletApplication MBean
```

where <..> is replaced by the corresponding application data. For example, `SampleApplication#SamplePortlet.war_portlet.SamplePortlet`.

Portlet and PortletApplication MBeans

The MBeans of type portlet and portletapplication provide information about a given portlet application and its portlets. Through the MBean of type portletapplication, you can retrieve a list of names of all portlets that belong to a portlet application. By querying the MBean of type portlet with a given portlet name, you can retrieve portlet specific information from the MBean of type portlet.

Each MBean that corresponds to a portlet or portlet application is uniquely identifiable by its name. Portlet applications are not required to have a name set within the portlet.xml. The MBean name for MBeans of the portletapplication type is the enterprise archive (EAR) file name followed by ″#″ and the Web module name concatenated with the string ″_portlet″. For example, portletapplication type MBeans have the following format:

```
<EarFileName>#<WarFileName>_portlet
```

The name chosen for the MBean of type portlet is the name of the MBean of type portletapplication that the portlet belongs to, concatenated with the portlet name:

```
<EarFileName>#<WarFileName>_portlet.<portletname>
```

The following is an example of the resulting PortletApplication MBean name and portlet names:

```
EarName            SampleEar
WebModule            SampleWar.war

PortletApplication MBean name:   SampleEar#SampleWar_portlet
Portlet:            SampleEar#SampleWar_portlet.BookmarkPortlet
```

The MBean names have been changed compared to version 6.1, because the old naming patterns are not unique and can lead to problems under certain circumstances. If you rely on the old naming pattern, you can set the portlet container custom property, `useShortMBeanNames`, to true to activate the previous known MBean names. Because this is a performance impact, you might not want to activate the old naming pattern if it is not necessary.

A full stop separates the preceding Web module name from the portlet name. Review the Portlet and PortletApplication MBean type API documentation for additional information. The generated API documentation is available in the information center table of contents from the path, **Reference** > **Administrator** > **API documentation** > **MBean interfaces**.

The following code is an example of how to invoke the MBean of type portletapplication for an application with the name, SampleWar.

```
String myPortletApplicationName = "SampleEar#SampleWar_portlet";
This name is composed by the Ear file name followed by "#" and
the Web module name concatenated with the substring "_portlet"


com.ibm.websphere.management.AdminService adminService =
```

```
  com.ibm.websphere.management.AdminServiceFactory.getAdminService();
javax.management.ObjectName on =
  new ObjectName("WebSphere:type=PortletApplication,name=" + myPortletApplicationName + ",*");

Iterator onIter = adminService.queryNames(on, null).iterator();
while(onIter.hasNext())
{
 on = (ObjectName)onIter.next();
}

String ctxRoot = (java.lang.String)adminService.getAttribute(on, "webApplicationContextRoot");
```

In the previous example, the MBeanServer is first queried for an MBean of type portletapplication. If this query is successful, the webApplicationContextRoot attribute is retrieved on that MBean or the first MBean that is found. The result is stored in the ctxRoot variable. This variable now contains the context root of the Web application that contains the portlet application that was searched. The variable is similar to ″/bookmark″.

The next code example demonstrates how to invoke the MBean of type portlet for a portlet with the name, BookmarkPortlet.

```
String myPortletName = "SampleEar#SampleWar_portlet.BookmarkPortlet";
This name is composed by the name of the MBean of type portletapplication and
the portlet name, separated by a full stop because the same portlet name may
be used within different Web modules, but must be unique within the system.


com.ibm.websphere.management.AdminService adminService =
  com.ibm.websphere.management.AdminServiceFactory.getAdminService();
javax.management.ObjectName on =
  new ObjectName("WebSphere:type=Portlet,name=" + myPortletName + ",*");
Iterator iter = adminService.queryNames(on, null).iterator();

while(iter.hasNext())
{
 on = (ObjectName)iter.next;
}

java.util.Locale locale = (java.util.Locale) adminService.getAttribute(on, "defaultLocale");
```

The locale returned by the method getAttribute method for the MBean is the default locale defined for this portlet.

Full names for Portlet and PortletApplication MBeans

MBeans are also registered by the full identifiable name:

```
<ApplicationName>#<WARfilename.war>_portlet.<portlet_name> for the Portlet MBean
<ApplicationName>#<WARfilename.war>_portlet for the PortletApplication MBean
```

where <..> is replaced by the corresponding application data. For example, `SampleApplication#SamplePortlet.war_portlet.SamplePortlet`. You can enable the short MBean names by setting the `useShortMBeanNames` portlet container custom property to true.

Portlet URL security

WebSphere Application Server enables direct access to portlet Uniform Resource Locators (URLs), just like servlets. This section describes security considerations when accessing portlets using URLs.

For security purposes, portlets are treated similar to servlets. Most portlet security uses the underlying servlet security mechanism. However, portlet security information resides in the `portlet.xml` file, while the

servlet and JavaServer Pages files reside in the web.xml file. Also, when you make access decisions for portlets, the security information, if any, in the web.xml file is combined with the security information in the portlet.xml file.

Portlet security must support both programmatic security, that is isUserInRole, and declarative security. The programmatic security is exactly the same as for servlets. However, for portlets, the isUserInRole method uses the information from the security-role-ref element in portlet.xml. The other two methods used by programmatic security, getRemoteUser and getUserPrincipal, behave the same way as they do when accessing a servlet. Both of these methods return the authenticated user information accessing the portlet.

The declarative security aspect of the portlets is defined by the security-constraint information in the portlet.xml file. This is similar to the security-constraint information used for the servlets in the web.xml file with the following differences:

- The auth-constraint element, which lists the names of the roles that can access the resources, does not exist in the portlet.xml file. The portlet.xml file contains only the user-data-constraint element, which indicates what type of transport layer security (HTTP or HTTPS) is required to access the portlet.
- The security-constraint information in the portlet.xml file contains the portlet-collection element, while the web.xml file contains the web-resource-collection element. The portlet-collection element contains only a list of simple portlet names, while the web-resource-collection contains the url-patterns as well as the HTTP methods that need protection.

The portlet container does not deal with the user authentication directly. For example, it does not prompt you to collect the credential information. The portlet container must, instead, use the underlying servlet container for the user authentication mechanism. As a result, there is no auth-constraint element in the security-constraint information in the portlet.xml file.

In WebSphere Application Server, when a portlet is accessed using a URL, the user authentication is processed based on the security-constraint information for that portlet in the web.xml file. This implies that to authenticate a user for a portlet, the web.xml file must contain the security-constraint information for that portlet with the relevant auth-constraints contained in it. If a corresponding auth-constraint for the portlet does not exist in the web.xml file, it indicates that the portlet is not required to have authentication. In this case, unauthenticated access is permitted just like a URL pattern for a servlet that does not contain any auth-constraints in the web.xml file. An auth-constraint for a portlet can be specified directly by using the portlet name in the url-pattern element, or indirectly by a url-pattern that implies the portlet.

Note: You cannot have a servlet or JSP with the same name as a portlet for WebSphere Application Server security to work with portlet.

The following examples demonstrate how the security-constraint information contained in the portlet.xml and web.xml files in a portlet application are used to make security decisions for portlets. The security-role-ref element, which is used for isUserInRole calls, is not discussed here because it is used the same way for servlets.

In the examples below (unless otherwise noted), there are four portlets (MyPortlet1, MyPortlet2, MyPortlet3, MyPortlet4) defined in portlet.xml. The portlets are secured by combining the information, if any, in the web.xml file when they are accessed directly through URLs.

All of the examples show the contents of the web.xml and portlet.xml files. Use the correct tools when creating these deployment descriptor files as you normally would when assembling a portlet application.

Example 1: The web.xml file does not contain any security-constraint data

In the following example, the security-constraint information is contained in portlet.xml:

```
<security-constraint>
   <display-name>Secure Portlets</display-name>
   <portlet-collection>
   <portlet-name>MyPortlet1</portlet-name>
   <portlet-name>MyPortlet3</portlet-name>
   </portlet-collection>
   <user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

In this example, when you access anything under MyPortlet1 and MyPortlet3, and these portlets are accessed using the unsecured HTTP protocol, you are redirected through the secure HTTPS protocol. The transport-guarantee is set to use secure connections. For MyPortlet2 and MyPortlet4, unsecured (HTTP) access is permitted because the transport-guarantee is not set. There is no corresponding security-constraint information for all four portlets in the `web.xml` file. Therefore, all of the portlets can be accessed without any user authentication and role authorization. The only security involved in this instance is the transport-layer security using Secure Sockets Layer (SSL) for MyPortlet1 and MyPortlet3.

The following table lists the security constraints that are applicable to the individual portlets.

URL	Transport Protection	User Authentication	Role Based Authorization
/MyPortlet1/*	HTTPS	None	None
/MyPortlet2/*	None	None	None
/MyPortlet3/*	HTTPS	None	None
/MyPortlet4/*	None	None	None

Example 2: The `web.xml` file contains portlet specific security-constraint data

In the following example, the security-constraint information that corresponds to the portlet is contained in `web.xml`. The `portlet.xml` file is the same as that shown in the previous example.

```
<security-constraint id="SecurityConstraint_1">
    <web-resource-collection id="WebResourceCollection_1">
      <web-resource-name>Protected Area</web-resource-name>
      <url-pattern>/MyPortlet1/*</url-pattern>
      <url-pattern>/MyPortlet2/*</url-pattern>
      </web-resource-collection>
    <auth-constraint id="AuthConstraint_1">
      <role-name>Employee</role-name>
    </auth-constraint>
</security-constraint>
```

The security-constraint information contained in the `web.xml` file in this example indicates that the user authentication must be performed when accessing anything under the MyPortlet1 and MyPortlet2 portlets. When you attempt to access these portlets directly using URLs, and there is no authentication information available, you are prompted to enter their credentials. After you are authenticated, the authorization check is performed to see if you are listed in the Employee role. The user/group to role mapping is assigned during the portlet application deployment. In the `web.xml` file listed above, note the following:

- Because the `web.xml` file uses url-pattern, the portlet names have been modified slightly. MyPortlet1 is now /MyPortlet1/*, which indicates that everything under the MyPortlet1 URL is protected. This matches the information in the `portlet.xml` file because the security runtime code converts the portlet-name element in the `portlet.xml` file to url-pattern (for example, MyPortlet1 to /MyPortlet1/*), even for the transport-guarantee.
- The http-method element in the `web.xml` file is not used in the example because all HTTP methods must be protected.

The following table lists the new security constraints that are applicable to the individual portlets.

URL	Transport Protection	User Authentication	Role Based Authorization
MyPortlet1/*	HTTPS	Yes	Yes (Employee)
MyPortlet2/*	None	Yes	Yes (Employee)
MyPortlet3/*	HTTPS	None	None
MyPortlet4/*	None	None	None

Example 3: The `web.xml` file contains generic security-constraint data implying all portlets.

In the following example, the security-constraint information is contained in the `web.xml` file that corresponds to the portlet. The `portlet.xml` file is the same as that shown in the first example.

```
<security-constraint id="SecurityConstraint_1">
    <web-resource-collection id="WebResourceCollection_1">
      <web-resource-name>Protected Area</web-resource-name>
      <url-pattern>/*</url-pattern>
      </web-resource-collection>
    <auth-constraint id="AuthConstraint_1">
      <role-name>Manager</role-name>
    </auth-constraint>
    </security-constraint>
```

In this example, /* implies that all resources that do not contain their own explicit security-constraints should be protected by the Manager role as per the URL pattern matching rules. Because the `portlet.xml` file contains explicit security-constraint information for MyPortlet1 and MyPortlet3, these two portlets are not protected by the Manager role, only by the HTTPS transport. Because the `portlet.xml` file cannot contain the auth-constraint information, any portlets that contain security-contraints in it are rendered unprotected when an implying URL (`/*` for example) is listed in the `web.xml` file because of the URL matching rules.

In the case above, both MyPortlet1 and MyPortlet3 can be accessed without user authentication. However, because MyPortlet2 and MyPortlet4 do not have security-constraints in the `portlet.xml` file, the /* pattern is used to match these portlets and are protected by the Manager role, which requires user authentication.

The following table lists the new security constraints that are applicable to the individual portlets with this setup.

URL	Transport Protection	User Authentication	Role Based Authorization
MyPortlet1/*	HTTPS	None	None
MyPortlet2/*	None	Yes	Yes (Manager)
MyPortlet3/*	HTTPS	None	None
MyPortlet4/*	None	Yes	Yes (Manager)

If in the example above, if you must also protect a portlet contained in the `portlet.xml` file (for example, MyPortlet1), the `web.xml` file should contain an explicit security-constraint entry in addition to `/*` as shown in the following example:

```
<security-constraint id="SecurityConstraint_1">
         <web-resource-collection id="WebResourceCollection_1">
            <web-resource-name>Protected Area</web-resource-name>
            <url-pattern>/*</url-pattern>
            </web-resource-collection>
         <auth-constraint id="AuthConstraint_1">
            <role-name>Manager</role-name>
          </auth-constraint>
```

```
</security-constraint>
<security-constraint id="SecurityConstraint_2">
   <web-resource-collection id="WebResourceCollection_2">
      <web-resource-name>Protection for MyPortlet1</web-resource-name>
      <url-pattern>/MyPortlet1/*</url-pattern>
      </web-resource-collection>
   <auth-constraint id="AuthConstraint_1">
      <role-name>Manager</role-name>
    </auth-constraint>
</security-constraint>
```

In this case, MyPortlet1 is protected by the Manager role and requires authentication. The data-constraint of `CONFIDENTIAL` is also applied to it because the information in the `web.xml` file and the `portlet.xml` file are combined. Because MyPortlet3 is not explicitly listed in the `web.xml` file, it is still not protected by the Manager role and does not require user authentication.

The following table shows the effect of this change.

URL	Transport Protection	User Authentication	Role Based Authorization
MyPortlet1/*	HTTPS	Yes	Yes (Manager)
MyPortlet2/*	None	Yes	Yes (Manager)
MyPortlet3/*	HTTPS	None	None
MyPortlet4/*	None	Yes	Yes (Manager)

Chapter 3. SIP applications

Using Session Initiation Protocol to provide multimedia and interactive services

Follow these procedures for creating SIP applications and configuring the SIP container.

About this task

Session Initiation Protocol (SIP) is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging. A *SIP application* in WebSphere is a Java program that uses at least one Session Initiation Protocol (SIP) servlet. A SIP servlet is a Java-based application component that is managed by a SIP servlet container.

The servlet container is a part of an application server that provides the network services over which requests and responses are received and sent. The servlet container decides which applications to invoke and in what order. A servlet container also contains and manages a servlet through its lifecycle.

SIP servlet containers can employ SIP proxy servers as surrogates to handle load balancing and security issues. For more information about the SIP proxy server, see: Session Initiation Protocol proxy server.

This topic is divided into the following subsections:

- Configure the SIP container: Information and instructions for configuring SIP container properties and timers.
- Developing SIP applications: Reference information for developers.
- Deploying SIP applications: Information for installing, starting, and stopping, your applications.
- Securing SIP applications: Instructions for enabling security providers and setting up a trust association interceptor (TAI).
- Tracing a SIP container: Troubleshoot SIP applications through traces on the SIP container.

SIP in WebSphere Application Server

WebSphere Application Server delivers rich SIP functionality throughout its infrastructure.

Session Initiation Protocol (SIP) has grown considerably since it first became an IETF standard in 1999. SIP was originally intended purely for video and audio but has now grown to become the control protocol for many interactive services, particularly in the peer-to-peer realm. SIP, and the standards surrounding SIP, provide the mechanisms to look up, negotiate, and manage connections to peers on any network over any other protocol.

The SIP Servlet 1.0 specification allows enterprise applications to use SIP and to support SIP-predominant applications in the Java EE environment.

WebSphere Application Server also provides tooling for the development environment and high performing Edge Components to handle distributed application environments.

In the application server, the Web container and SIP container are converged and are able to share session management, security and other attributes. In this model, an application that includes SIP servlets, HTTP servlets, and portlets can seamlessly interact, regardless of the protocol.

High availability, offered by the Network Deployment (ND) version of the product, of the converged applications is made possible because of the tight integration of HTTP and SIP in the base application server.

In front of a clustered application sits the proxy server, managing the traffic and workload of the SIP and HTTP traffic to the container. This proxy server is a stateless SIP proxy and a HTTP reverse proxy together, which uses the unified clustering framework and high availability (HA) manager services of the ND package to seamlessly monitor the health of the servers. The proxy server also can act as a stand-alone stateless SIP proxy in front of the SIP container in the application server when no HTTP traffic is present.

It's important to note that the SIP function in the proxy server is stateless. The SIP RFC defines two types of proxy servers, one stateful and one stateless. Normally, a SIP proxy is a stateful instance and stateless proxies are specified as such. A stateful proxy participates in the call flows and is implemented using SIP servlets.

The stateless SIP proxy functionality in the proxy server allows the proxy to handle the workload, routing, and session affinity needs of the SIP container with less complexity. Being stateless, the proxy server can be fronted by a simple IP sprayer such as the load balancer component included in the ND package. If a proxy server fails, the affinity is to the container and not to the proxy itself so there is one less potential failure along the message flow.

SIP Infrastructure

The SIP infrastructure is a multi-tiered architecture made up of SIP containers, SIP proxies and an IP sprayer. The SIP container is a general purpose SIP application server. The SIP infrastructure consists of:

- SIP container – Web container extension that implements JSR 116 plus a SIP protocol stack that implements all pertinent RFCs.
- SIP proxy – Stateless edge device that handles I/O concentration, load balancing, and other functions, in a similar manner to the reverse HTTP proxy. This is not the same as the SIP proxy defined by RFC 3261.
- Load balancer – SIP enabled to interoperate with SIP proxies and SIP containers. The extendable SIP proxy handles session affinity and load balancing. The load balancer functions as a highly available IP sprayer to dispatch messages to the proxies.

SIP is a key element for many new applications, especially when converged with HTTP, including:

- Click-To-Call
- Voice over IP
- Third Party Call Control and Call Monitoring
- Presence and Instance Messaging

SIP applications

A *SIP application* is a Java program that uses at least one Session Initiation Protocol (SIP) servlet.

A SIP servlet is a Java-based application component that is managed by a SIP servlet container and that performs SIP signaling. Like other Java-based components, servlets are platform-independent Java classes that are compiled to platform-neutral bytecode that can be loaded dynamically into and run by a Java-enabled SIP application server. Containers, sometimes called servlet engines, are server extensions that handle servlet interactions. SIP servlets interact with clients by exchanging request and response messages through the servlet container.

SIP is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging. ″Presence″ in this context refers to user status such as ″Active,″ ″Away,″ or ″Do not disturb.″ The standard that defines a programming model for writing SIP-based servlet applications is JSR 116.

SIP container

This product complies with the following SIP standards:

IETF

JCP

For a complete list of the supported Internet Engineering Task Force (IETF) and Java Community Process (JCP) industry standards, see the ″Compliance with industry SIP standards″ topic linked below.

SIP industry standards compliance

The product implementation of Session Initiation Protocol (SIP) complies with industry standards for both a SIP container and SIP applications.

SIP container

This product complies with the following SIP standards:

IETF

JCP

This product also complies with the Internet Engineering Task Force (IETF) and Java Community Process (JCP) industry standards for SIP. The following table contains a list of the IETF and JCP standards.

Table 1. WebSphere Application Server complies with these SIP standards

Standard	Description
RFC 2543	SIP: Session Initiation Protocol
RFC 3261	SIP: Session Initiation Protocol
RFC 3262	Reliability of provisional responses in SIP
RFC 3265	SIP-specific event notification
RFC 3326	The Reason Header field for the SIP
RFC 3515	The SIP Refer method
RFC 3824	Using E.164 numbers with the SIP
RFC 3903	SIP Extension for event state publication
RFC 3263	Locating SIP servers **Note:** SIP does not support use of DNS procedures for a server to send a response to a back-up client if the primary client fails.

SIP applications

This product complies with standards for SIP applications.

Table 2. Compliance with standards for SIP applications

Standard	Description
RFC 2848	The PINT Service Protocol: Extensions to SIP and Session Description Protocol (SDP) for internet protocol (IP) access to telephone call services
RFC 2976	The SIP INFO method
RFC 3050	Common gateway interface for SIP
RFC 3087	Control of service context using SIP request-URI
RFC 3264	An offer and answer model with SDP
RFC 3266	Support for IPv6 in SDP
RFC 3312	Integration of resource management and SIP

Table 2. Compliance with standards for SIP applications (continued)

Standard	Description
RFC 3313	Private SIP extensions for media authorization
RFC 3319	Dynamic Host Configuration Protocol (DHCPv6) options for SIP servers
RFC 3327	SIP Extension Header field for registering non-adjacent contacts
RFC 3372	SIP for telephones (SIP-T): context and architectures
RFC 3398	Integrated Services Digital Network (ISDN) User Part (ISUP) to SIP mapping
RFC 3428	SIP extension for instant messaging
RFC 3455	Private Header (P-Header) extensions to the SIP for the 3rd-Generation Partnership Project (3GPP)
RFC 3578	Mapping of Integrated Services Digital Network (ISDN) User Part (ISUP) overlap signaling to the SIP
RFC 3603	Private SIP proxy-to-proxy extensions for supporting the PacketCable distributed call signaling architecture
RFC 3608	SIP Extension Header field for service route discovery during registration
RFC 3665	SIP basic call flow examples
RFC 3666	SIP Public Switched Telephone Network (PSTN) call flows
RFC 3680	A SIP event package for registrations
RFC 3725	Best current practices for third-party call control (3pcc) in the SIP
RFC 3840	Indicating user agent capabilities in the SIP
RFC 3842	A message summary and message waiting indication event package for the SIP
RFC 3856	A presence event package for the SIP
RFC 3857	A watcher information event template package for the SIP
RFC 3959	The early session disposition type for the SIP
RFC 3960	Early media and ringing tone generation in the SIP
RFC 3976	Interworking SIP and intelligent network (IN) applications
RFC 4032	Update to the SIP preconditions framework
RFC 4092	Usage of the SDP Alternative Network Address Types (ANAT) semantics in the SIP
RFC 4117	Transcoding services invocation in the SIP using third-party call control (3pcc)
RFC 4235	An invite-initiated dialog event package for the SIP
RFC 4240	Basic network media services with SIP
RFC 4353	A framework for conferencing with the SIP
RFC 4354	A SIP event package and data format for various settings in support for the push-to-talk over cellular (PoC) service
RFC 4411	Extending the SIP Reason Header for preemption events
RFC 4457	The SIP P-user-database Private-Header (P-Header)
RFC 4458	SIP URIs for applications such as voicemail and interactive voice response (IVR)
RFC 4483	A mechanism for content indirection in SIP messages
RFC 4497	Interworking between the SIP and QSIG
RFC 4508	Conveying feature tags with the SIP REFER method

Runtime considerations for SIP application developers

You should consider certain product runtime behaviors when you are writing Session Initiation Protocol (SIP) applications.

Container may accept non-SIP URI schemes

The SIP container will not reject a message if it doesn't recognize the scheme in the request URI because the container cannot know which URI schemes are supported by the applications. SIP elements may support a request URI with a scheme other than `sip` or `sips`, for example, the `pres:` scheme has a particular meaning for presence servers, but the container does not recognize it. It is up to the application to determine whether to accept or to reject a specific scheme. SIP elements may translate non-SIP URIs using any mechanism available, resulting in SIP URIs, SIPS URIs, or other schemes, like the `tel` URI scheme of RFC 2806 [9].

Directing requests in a multiple-container environment

In a multiple-container environment (SIP proxy plus SIP containers), when your application sends a request intended initially to be sent externally but later received, it should use the host and ports of the front-most load balancing element (either an IP sprayer for multiple SIP proxies, or the SIP proxy if only one exists). If the application uses the host name of a container instead of the front-most element, the request may be lost in the event of a failure.

For example, an application sends an INVITE request to itself, but the request must pass through an external accounting system through a pushed Route header. The application should set the INVITE request's URI to the host and port of the foremost element to ensure that failover occurs. The request will be routed to the accounting system via the pushed Route, and then sent back to the front load balancing element for processing.

Invoking session listener events

SipSessionListener and SipApplicationSessionListener events are invoked only if an application requests the corresponding session object. You do this by using in your application the method shown in Table 3.

Table 3. Methods that invoke session listener events

Event	Method
SipSessionListener	getSession()
SipApplicationSessionListener	getApplicationSession()

Session activation and passivation

During normal operation, this product never migrates a session from one server to another. Session migration occurs only as a result of a server failure. Therefore the SipSessionActivationListener method's passivation callback is never invoked. However, the activation callback is invoked when a failure forces session failover to a different server.

External resources

If a SIP application performs intensive I/O or accesses an external database, it may be blocked for several milliseconds. If possible, use asynchronous APIs for these resources . Under stress, a blocked SIP application may trigger a Request Timeout or re-transmission.

SIP application attributes

Avoid hanging large objects or BLOBs as SIP Session attributes (via SIPSession.setAttribute API). This may damage the overall performance when combined with high availability (HA). The same recommendation applies for SIPApplicationSession.setAttribute. In most cases, the large object can be replaced by several simple or composed strings.

SIP IBM Rational Application Developer for WebSphere framework

This page provides information about the SIP IBM Rational Application Developer for WebSphere framework.

WebSphere Application Server includes IBM Rational Application Developer for WebSphere to meet all the basic development needs for Java EE applications. Included in IBM Rational Application Developer for WebSphere is support for developing SIP servlet applications. IBM Rational Application Developer for WebSphere provides graphical deployment descriptor editors and basic wizards to get you started writing SIP servlets.

IBM Rational Application Developer for WebSphere also includes many other pieces that integrate well in WebSphere Application Server deployments, such as the Unit Test Environment, which provides the WebSphere Application Server servlet container to run SIP servlets in the development phase of the product, as well as tools for server automation and application packaging.

IBM Rational Application Developer for WebSphere supports:

- SIP servlet development (JSR 116)
- Converged SIP/HTTP applications
- Import/Export SAR packages
- SIP samples (call forward, call block, third party call)

SIP container

A *SIP container* is a Web application server component that invokes the Session Initiation Protocol (SIP) action servlet and that interacts with the action servlet to process SIP requests.

The servlet container provides the network services over which requests and responses are received and sent. It decides which applications to invoke and in what order. The container also contains and manages servlets through their life cycle.

A SIP servlet container manages the network listener points on which it listens for incoming SIP traffic. A listener point is a combination of transport protocol, IP address, and port number. The SIP servlet container supports the transport protocols UDP, TCP, and TLS over TCP.

The SIP servlet container can employ a SIP proxy server to route, load balance, and improve response times between SIP requests and back-end SIP container resources. For more information about the SIP proxy server, see: Session Initiation Protocol proxy server.

SIP converged proxy

SIP in WebSphere offers a converged proxy.

The SIP converged proxy:

- Handles SIP and HTTP
- Fronts clusters of containers, SIP or HTTP
- Provides a highly scalable I/O concentration
- Handles session affinity
- Provides a framework for extending the base functions of proxy using an API consistent with proxy flows (Proxy Filter Layer)
- Provides first pass protocol validation
- Provides a framework for secure proxy functions – SSL termination, Outbound SSL, Client Side Certificates, etc.
- Allows for augmentation by our other products such as WebSphere XD
- Provides DMZ support.

SIP proxy setup considerations

- A single SIP proxy can front multiple SIP clusters.
- An IP sprayer is required for load balancing when deploying multiple SIP proxies into a single cell.
- Each SIP proxy must be configured with a default cluster. This is used to route inbound messages that do not match a cluster routing rule.
- When deploying converged applications, both HTTP and SIP should be enabled on the proxy.
- SIP proxies can be clustered.

SIP port relationships

When multiple servers, either containers or proxies are on the same host, each container or proxy must be configured with its own port.

SIP cluster routing and the default cluster

- A single SIP proxy can front multiple SIP clusters.
- Each SIP proxy must be configured with a default cluster which is used to route all messages that do not have an associated cluster routing rule.
- You can define cluster routing rules at each proxy. These dictate how messages are routed to the various backend clusters being fronted.
- By changing the default cluster, a SIP proxy can reroute messages to a new cluster containing an upgraded version of the deployed applications.

SIP timer summary

Request for Comments (RFC) 3261, “SIP: Session Initiation Protocol,” specifies various timers that SIP uses.

Table 4 summarizes for each SIP timer the default value, the section of RFC 3261 that describes the timer, and the meaning of the timer.

Table 4. Summary of SIP timers

Timer	Default value	Section	Meaning
T1	500 ms	17.1.1.1	Round-trip time (RTT) estimate
T2	4 sec.	17.1.2.2	Maximum retransmission interval for non-INVITE requests and INVITE responses
T4	5 sec.	17.1.2.2	Maximum duration that a message can remain in the network
Timer A	initially T1	17.1.1.2	INVITE request retransmission interval, for UDP only
Timer B	64*T1	17.1.1.2	INVITE transaction timeout timer
Timer C	> 3 min.	16.6 bullet 11	Proxy INVITE transaction timeout
Timer D	> 32 sec. for UDP	17.1.1.2	Wait time for response retransmissions
	0 sec. for TCP and SCTP		
Timer E	initially T1	17.1.2.2	Non-INVITE request retransmission interval, UDP only
Timer F	64*T1	17.1.2.2	Non-INVITE transaction timeout timer
Timer G	initially T1	17.2.1	INVITE response retransmission interval
Timer H	64*T1	17.2.1	Wait time for ACK receipt
Timer I	T4 for UDP	17.2.1	Wait time for ACK retransmissions
	0 sec. for TCP and SCTP		

Table 4. Summary of SIP timers (continued)

Timer	Default value	Section	Meaning
Timer J	64*T1 for UDP	17.2.2	Wait time for retransmissions of non-INVITE requests
	0 sec. for TCP and SCTP		
Timer K	T4 for UDP	17.1.2.2	Wait time for response retransmissions
	0 sec. for TCP and SCTP		

Developing SIP applications

A SIP application is a set of SIP servlets packaged in a SIP application archive file (SAR).

About this task

A SIP servlet is an application component managed by the SIP container that performs SIP signaling. The programming and deployment models are analogous to Web servlets and therefore will be mapped to the WebSphere administrative model accordingly. It is possible to include Web servlets in a SAR file (along with the required web.xml deployment descriptor) to create what is known as a converged application. See JSR 116 for details on SIP applications, servlets, converged applications, and status codes.

Developing a custom trust association interceptor

When you develop Session Initiation Protocol (SIP) applications, you can create a custom trust association interceptor (TAI).

Before you begin

You may want to familiarize yourself with the general TAI information contained in the Trust Associations documentation. Developing a SIP TAI is similar to developing any other custom interceptors used in trust associations. In fact, a custom TAI for a SIP application is actually an extension of the trust association interceptor model. Refer to the Developing a custom interceptor for trust associations section for more details.

About this task

TAI can be invoked by a SIP servlet request or a SIP servlet response. To implement a custom SIP TAI, you need to write your own Java class.

1. Write a Java class that extends the `com.ibm.wsspi.security.tai.BaseTrustAssociationInterceptor` class and implements the `com.ibm.websphere.security.tai.SIPTrustAssociationInterceptor` interface. Those classes are defined in the *WASProductDir*/plugins/com.ibm.ws.sip.container_1.0.0.jar file, where *WASProductDir* is the fully qualified path name of the directory in which WebSphere Application Server is installed.
2. Declare the following Java methods:

   `public int initialize(Properties properties) throws WebTrustAssociationFailedException;`
   This is invoked before the first message is processed so that the implementation can allocate any resources it needs. For example, it could establish a connection to a database. `WebTrustAssociationFailedException` is defined in the *WASProductDir*/plugins/com.ibm.ws.runtime_1.0.0.jar file. The value of the `properties` argument comes from the **Custom Properties** set in this step.

public void cleanup();

This is invoked when the TAI should free any resources it holds. For example, it could close a connection to a database.

public boolean isTargetProtocolInterceptor(SipServletMessage sipMsg) throws WebTrustAssociationFailedException;

Your custom TAI should use this method to handle the `sipMsg` message. If the method returns `false`, WebSphere ignores your TAI for `sipMsg`.

public TAIResult negotiateValidateandEstablishProtocolTrust (SipServletRequest req, SipServletResponse resp) throws WebTrustAssociationFailedException;

This method returns a TAIResult that indicates the status of the message being processed and a user ID or the unique ID for the user who is trying to authenticate. If authentication succeeds, the TAIResult should contain the status HttpServletResponse.SC_OK and a principal. If authentication fails, the TAIResult should contain a return code of HttpServletResponse.SC_UNAUTHORIZED (401), SC_FORBIDDEN (403), or SC_PROXY_AUTHENTICATION_REQUIRED (407). This only indicates whether or not the container should accept a message for further processing. To challenge an incoming request, the TAI implementation must generate and send its own SipServletResponse containing a challenge. The exception should be thrown for internal TAI errors. Table 5 describes the argument values and resultant actions for the negotiateValidateandEstablishProtocolTrust method.

Table 5. Description of negotiateValidateandEstablishProtocolTrust arguments and actions

Argument or action	For a SIP request	For a SIP response
Value of `req` argument	The incoming request	Null
Value of `resp` argument	Null	The incoming response
Action for valid response credentials	Return TAIResult.status containing SC_OK and a user ID or unique ID	Return TAIResult.status containing SC_OK and a user ID or unique ID
Action for incorrect response credentials	Return the TAIResult with the 4*xx* status	Return the TAIResult with the 4*xx* status

The sequence of events is as follows:

a. The SIP container maps initial requests to applications by using the rules in each applications deployment descriptor; subsequent messages are mapped based on JSR 116 mechanisms.

b. If any of the applications require security, the SIP container invokes any defined TAI implementations for the message.

c. If the message passes security, the container invokes the corresponding applications.

Your TAI implementation can modify a SIP message, but the modified message will not be usable within the request mapping process, because it finishes before the container invokes the TAI.

The com.ibm.wsspi.security.tai.TAIResult class, defined in the *WASProductDir*/plugins/com.ibm.ws.runtime_1.0.0.jar file, has three static methods for creating a TAIResult. The TAIResult `create` methods take an int type as the first parameter. WebSphere Application Server expects the result to be a valid HTTP request return code and is interpreted as follows:

If the value is HttpServletResponse.SC_OK, this response tells WebSphere Application Server that the TAI has completed its negotiation. The response also tells WebSphere Application Server use the information in the TAIResult to create a user identity.

The created TAIResults have the meanings shown in Table 6.

Table 6. Meanings of TAIResults

TAIResult	Explanation
public static TAIResult create(int status);	Indicates a status to WebSphere Application Server. The status should not be SC_OK because the identity information is provided.
public static TAIResult create(int status, String principal);	Indicates a status to WebSphere Application Server and provides the user ID or the unique ID for this user. WebSphere Application Server creates credentials by querying the user registry.
public static TAIResult create(int status, String principal, Subject subject);	Indicates a status to WebSphere Application Server, the user ID or the unique ID for the user, and a custom Subject. If the Subject contains a Hashtable, the principal is ignored. The contents of the Subject becomes part of the eventual user Subject.

`public String getVersion();`
This method returns the version number of the current TAI implementation.

`public String getType();`
This method's return value is implementation-dependent.

3. Compile the implementation after you have implemented it. For example: `/opt/WebSphere/AppServer/java/bin/javac -classpath /opt/WebSphere/AppServer/plugins/com.ibm.ws.runtime_1.0.0.jar;/opt/WebSphere/AppServer/lib/j2ee.jar;/opt/WebSphere/AppServer/plugins/com.ibm.ws.sip.container_1.0.0.jar myTAIImpl.java`
   a. For each server within a cluster, copy the class file to a location in the WebSphere class path (preferably the *WASProductDir*/plugin/ directory).
   b. Restart all the servers.
4. Delete the default WebSEAL interceptor in the administrative console and click **New** to add your custom interceptor. Verify that the class name is dot-separated and appears in the class path.
5. Click the **Custom Properties** link to add additional properties that are required to initialize the custom interceptor. These properties are passed to the `initialize(Properties properties)` method of your implementation when it extends the `com.ibm.websphere.security.WebSphereBaseTrustAssociationInterceptor` as described in the previous step.
6. Save and synchronize (if applicable) the configuration.
7. Restart the servers for the custom interceptor to take effect.

Developing SIP applications that support PRACK

A SIP response to an INVITE request can be final or provisional. Final responses are always sent reliably, but provisional responses typically are not. For cases where you need to send a provisional response reliably, you can use the PRACK (Provisional response acknowledgement) method.

Before you begin

For you to be able to develop applications that support PRACK, the following criteria must be met:

- The client that sends the INVITE request must put a 100rel tag in the Supported or the Require header to indicate that the client supports PRACK.
- The SIP servlet must respond by invoking the sendReliably() method instead of the send() method to send the response.

About this task

PRACK is described in the following standards:

- RFC 3262 (“Reliability of Provisional Responses in the Session Initiation Protocol (SIP)”), which extends RFC 3261 (“SIP: Session Initiation Protocol”), adding PRACK and the option tag 100rel.
- Section 6.7.1 (“Reliable Provisional Responses”) of JSR 116 (“SIP Servlet API Version 1.0”).
- For an application acting as a proxy, do this:
  - Make your application generate and send a reliable provisional response for any INVITE request that has no tag in the To field.
- For an application acting as a user agent client (UAC), do this:
  - Make your application add the 100rel tag to outgoing INVITE requests. The option tag must appear in either the Supported header or the Require header.
  - Within your application’s doProvisionalResponse(...) method, prepare the application to create and send PRACK requests for incoming reliable provisional responses. The application must create the PRACK request on the response’s dialog through a SipSession.createRequest(...) method, and it must set the RAck header according to RFC 3262 Section 7.2 (“RAck”).
  - The application that acts as an UAC will not receive doPrack( ) methods. The UAC sends INVITE and receives Reliable responses. When the UAC receives the Reliable response, it sends PRACK a request to the UAS and receives a 200 OK on the PRACK so it should next implement doResponse( ) in order to receive it.
- For an application acting as a user agent server (UAS), do this:
  - If an incoming INVITE request requires the 100rel tag, trying to send a 101-199 response unreliably by using the send() method causes an Exception.
  - Make the application declare a SipErrorListener to receive noPrackReceived() events when a reliable provisional response is not acknowledged within 64*T1 seconds, where T1 is a SIP timer. Within the noPrackReceived() event processing, the application should generate and send a 5*xx* error response for the associated INVITE request per JSR 116 Section 6.7.1.
  - Make the application have at most one outstanding, unacknowledged reliable provisional response. Trying to send another one before the first’s acknowledgement results in an Exception.
  - Make sure that the application enforces the RFC 3262 offer/answer semantics surrounding PRACK requests containing session descriptions. Specifically, a servlet must not send a 2*xx* final response if any unacknowledged provisional responses contained a session description.

Setting up SIP application composition

The JSR 116 standard for SIP applications states in section 2.4 that multiple applications may be invoked for the same SIP request. The process of setting up applications to comply with this standard is called application composition.

Before you begin

About this task

Application composition requires that implementations use a cascaded services model. The cascaded services model requires that service applications triggered on the same host are triggered in sequence, as if the triggering occurred on different hosts. Therefore responses flow upstream and hit applications in the reverse order of the corresponding requests.

The JSR 116 standard does not specify how to implement application composition, thus there are many ways to comply with this standard. For WebSphere Application Server, composition of the application depends on the deployed application order, and on the order of mapping rules within the deployment descriptor of each application.

- For an initial incoming request, the SIP container tries each potential rule in order. When the container finds the n^{th} match, the container invokes the corresponding servlet.
- If the servlet must proxy the request, the container scans the rules again to search for additional matches. When the container finds the $(n+1)^{th}$ match, the container invokes the corresponding servlet.

- Any servlet in the same application as the previously invoked servlet is excluded from the matching process. No servlet can be invoked twice for the same SIP request.

You can specify load on start-up priority. The <load-on-startup> in the sip.xml defines the order in which servlets are initialized on startup. If this value is lower than zero, the servlets are initialized when the first request is matched to them according to matching rule and composition order. Zero is a legitimate weight for startup initialization order. If this tag does not exist or if it contains a negative value, the servlet does not initialize at startup.

You should also add <load-on-startup> to the same tag in the web.xml if you are changing it manually. It is the WebContainer that loads servlets (and siplets), and it looks only at the web.xml. When deploying a SAR, only the sip.xml needs to be changed. The web.xml is automatically constructed correctly after deployment.

The load-on-startup tag embedded in the SIP deployment descriptor tag for a servlet dictates the order that the application is loaded on start up of the server. It does not dictate the order that an application gets called when the application is a member of an application composition chain that matches rules to process a new message coming in.

The starting weight for applications and their modules is specified in the deployment.xml file. The order in which modules pickup requests on composition is evaluated by applications weight first and then modules weight. The following steps can be completed in any order to specify applications weight or modules weight from the administrative console.

1. To specify the applications (EARs) weight, expand **Enterprise Applications** → ***applicationName*** → **Startup Behavior** and set the startup order.
2. To specify the modules (WARs) weight, expand **Enterprise Applications** → ***applicationName*** → **Manage Modules** and set the starting weight.
3. Restart the changed applications.

Example

Note:

sip.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app
PUBLIC "-//Java Community Process//DTD SIP Application 1.0//EN"
"http://www.jcp.org/dtd/sip-app_1_0.dtd">
<sip-app>
 <display-name>SIPSampleProxy</display-name>

 <servlet>
  <servlet-name>SIPSampleProxy</servlet-name>
  <servlet-class>sipes.test.container.proxy.SIPSampleProxy</servlet-class>
  <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
  <servlet-name>SIPSampleProxy</servlet-name>
  <pattern>
   <equal>
    <var>request.uri.user</var>
    <value>SIPSampleProxy</value>
   </equal>
  </pattern>
 </servlet-mapping>

 <proxy-config>
  <sequential-search-timeout>1000</sequential-search-timeout>
 </proxy-config>
 <session-config>
  <session-timeout>12</session-timeout>
 </session-config>
</sip-app>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app id="WebApp">
 <display-name>SIPSampleProxy</display-name>
 <servlet>
  <servlet-name>SIPSampleProxy</servlet-name>
  <display-name>SIPSampleProxy</display-name>
  <servlet-class>sipes.test.container.proxy.SIPSampleProxy</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>SIPSampleProxy</servlet-name>
  <url-pattern>/SIPSampleProxy</url-pattern>
 </servlet-mapping>
 <welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
 </welcome-file-list>
</web-app>
```

Note:

The following example is for a standalone server.

deployment.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <appdeployment:Deployment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:appdeployment="http://www.ibm.com/websphere/appserver/schemas/5.0/appdeployment.xmi"
xmi:id="Deployment_1137951186883">
- <deployedObject xmi:type="appdeployment:ApplicationDeployment" xmi:id="ApplicationDeployment_1137951186883"
deploymentId="0" startingWeight="1" binariesURL="$(APP_INSTALL_ROOT)/OrangeNode08Cell/SipContainerTestSuite.ear"
useMetadataFromBinaries="false" enableDistribution="true" createMBeansForResources="true" reloadEnabled="false"
appContextIDForSecurity="href:OrangeNode08Cell/SipContainerTestSuite"
filePermission=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755" allowDispatchRemoteInclude="false"
allowServiceRemoteInclude="false">
  <targetMappings xmi:id="DeploymentTargetMapping_1137951186883" enable="true" target="ServerTarget_1137951186883" />
  <classloader xmi:id="Classloader_1137951186883" mode="PARENT_FIRST" />
- <modules xmi:type="appdeployment:WebModuleDeployment" xmi:id="WebModuleDeployment_1137951186883"
deploymentId="1" startingWeight="10000" uri="sipunit.war">
  <targetMappings xmi:id="DeploymentTargetMapping_1137951186884" target="ServerTarget_1137951186883" />
  <classloader xmi:id="Classloader_1137951186884" /> </modules>
  <properties xmi:id="Property_1137951186883" name="validateinstall" value="warn" /> </deployedObject>
  <deploymentTargets xmi:type="appdeployment:ServerTarget" xmi:id="ServerTarget_1137951186883"
name="server1" nodeName="OrangeNode10" /> </appdeployment:Deployment>
```

SIP servlets

This topic describes SIP servlets.

The SIP Servlet 1.0 specification (JSR 116) is standardized through Java Specification Request (JSR) 116. The idea behind the specification is to provide a Java application programming interface (API) similar to HTTP servlets, which provides an easy-to-use SIP programming model. Like the popular HTTP servlet programming model, some flexibility is limited to optimize ease-of-use and time-to-value.

However, the SIP Servlet API is different in many ways from HTTP servlets because the protocol is so different. While SIP is a request-response protocol, there is not necessarily only one response to every one request. This complexity and a need for a high performing solution meant that it was easier to make the SIP servlets natively asynchronous. Also, unlike HTTP servlets, the programming model for SIP servlets sought to make client requests easy to create alongside the other logic being written because many applications act as a client or proxy to other servers or proxies.

SipServlet requests

Like HTTP servlets, each SIP servlet extends a base javax.servlet.sip.SipServlet class. All messages come in through the service method, which you can extend. However, because there is not a one-to-one mapping of requests to responses in SIP, the suggested practice is to extend the doRequest or doResponse methods instead. When extending the doRequest or doResponse methods, it is important to call the extended method for the processing to complete.

Each request method, which the specification must support, has a doxxx method just like HTTP. In HTTP, methods such as doGet and doPost exist for GET and POST requests. In SIP, doInvite, doAck, doOptions, doBye, doCancel, doRegister, doSubscribe, doNotify, doMessage, doInfo, and doPrack methods exist for each SIP request method.

Unlike an HTTP servlet, SIP servlets have methods for each of the response types that are supported. So, SIP servlets include the doProvisionalResponse, doSuccessResponse, doRedirectResponse, and doErrorResponse responses. Specifically, the provisional responses (1xx responses) are used to indicate status, the success responses (2xx responses) are used to indicate a successful completion of the transaction, the redirect responses (3xx responses) are used to redirect the client to a moved resource or entity, and the error responses (4xx, 5xx, and 6xx responses) are used to indicate a failure or a specific error condition. These types of response messages are similar to HTTP, but because the SIP Servlet programming model includes a client programming model, it is necessary to have responses handled programmatically as well.

Clarifications of JSR 116

JSR 289 has made some clarifications to JSR 116, as follows:

- JSR 289 Section 4.1.3: Contact Header Field
- JSR 289 Section 5.2: Implicit Transaction State
- JSR 289 Section 5.8: Accessibility of SIP Servlet Messages

SIP SipServletRequest and SipServletResponse classes

The SipServletRequest and SipServletResponse classes are similar to the HttpServletRequest and HttpServletResponse classes.

SipServletRequest and SipServletResponse classes

Each class gives you the capability to access the headers in the SIP message and manipulate them. Because of the asynchronous nature of the requests and responses, this class is also the place to create new responses for the requests. When you extend the doInvite method, only the SipServletRequest class is passed to the method. To send a response to the client, you must call the createResponse method on the Request object to create a response. For example:

```
protected void doInvite(SipServletRequest req) throws
 javax.servlet.ServletException, java.io.IOException {

 //send back a provisional Trying response
 SipServletResponse resp = req.createResponse(100);
resp.send();
```

Because of their asynchronous nature, SIP servlets can seem complicated. However, something as simple as the previous code sample sends a response to a client.

Here is a more complex example of a SIP servlet. With the following method included in a SIP servlet, the servlet blocks all of the calls that do not come from the example.com domain.

```
protected void doInvite(SipServletRequest req) throws
 javax.servlet.ServletException, java.io.IOException {

//check to make sure that the URI is a SIP URI
      if (req.getFrom().getURI().isSipURI()){
                SipURI uri = (SipURI)req.getFrom.getURI();
                if (!uri.getHost().equals("example.com")) {
                   //send forbidden response for calls outside domain
                   req.createResponse(SipServletResponse.SC_FORBIDDEN).send();
                   return;
                }
```

```
            }
            //proxy all other requests on to their original destination
            req.getProxy().proxyTo(req.getRequestURI);
        }
```

SIP SipSession and SipApplicationSession classes

Possibly the most complex portions of the SIP Servlet 1.0 specification are the SipSession and SipApplicationSession classes.

SIP SipSession and SipApplicationSession classes

Both of these classes have some useful purposes and can act as the primary place to store data in applications that are designed for distributed or highly available environments.

The SipSession class is the best representative of a specific point-to-point communication between two entities and is the closest to the HttpSession object. Because historically no proxying or forking existed for the HTTP request in HTTP servlets, the need for something higher than a single point-to-point session did not exist. However, even HTTP users can see the growing need for this type of function since portlets began essentially forking HTTP requests. The SIP users expect the proxying and forking activities that require multiple layers of SIP session management. The SipSession class is the lowest point-to-point layer.

The SipApplicationSession class represents the higher layer of SIP session management. One SipApplicationSession class can own one or more SipSession objects. However, each SipSession class can be related to one SipSession object only. The SipApplicationSession class also supports the attachment of any number of other protocol sessions. Currently, only HTTP sessions are supported by any implementations. The SipApplicationSession class has a getSessions method, which takes the requested protocol type as an argument.

You might find it useful for many applications to combine HTTP and SIP. For example, you might use this approach to tie together HTTP and SIP sessions to monitor a phone call or to start a phone call through a rich HTTP graphical user interface.

Example: SIP servlet simple proxy

This is a servlet example of a simple proxy.

Simple proxy

```
import java.io.IOException;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.sip.Proxy;
import javax.servlet.sip.SipFactory;
import javax.servlet.sip.SipServlet;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServletResponse;
import javax.servlet.sip.SipSession;
import javax.servlet.sip.SipURI;
import javax.servlet.sip.URI;


public class SimpleProxy extends SipServlet implements Servlet {


 final static  private String  SHUTDOWN_KEY = new String("shutdown");
 final static  private String  STATE_KEY = new String("state");
 final static  private int   INVITE_RECEIVED = 1;

 /* (non-Java-doc)
  * @see javax.servlet.sip.SipServlet#SipServlet()
  */
 public SimpleProxy() {
```

```
 super();
}

/* (non-Javadoc)
 * @see javax.servlet.sip.SipServlet#doInvite(javax.servlet.sip.SipServletRequest)
 */
protected void doInvite(SipServletRequest request) throws ServletException,
  IOException {

 //log("SimpleProxy: doInvite: TOP");

 try {
  if (request.isInitial() == true)
  {
   // This should cause the sip session to be created. This sample only uses the session on receiving
   // a BYE but the Tivoli performance viewer can be used to track the creation of calls by viewing the
   // active session count.
   Integer state = new Integer(INVITE_RECEIVED);
   SipSession session = request.getSession();
   session.setAttribute(STATE_KEY, state);
            log("SimpleProxy: doInvite: setting attribute");

   Proxy proxy = request.getProxy();

   SipFactory sipFactory = (SipFactory) getServletContext().getAttribute(SIP_FACTORY);
         if (sipFactory == null) {
             throw new ServletException("No SipFactory in context");
         }

         String callingNumber = request.getTo().toString();
         if (callingNumber != null)
         {
             String destStr = format_lookup(callingNumber);
             URI dest = sipFactory.createURI(destStr);

             //log("SimpleProxy: doInvite: Proxying to dest URI = " + dest.toString());

          if (((SipURI)request.getRequestURI()).getTransportParam() != null)
           ((SipURI)dest).setTransportParam(((SipURI)request.getRequestURI()).getTransportParam());

          proxy.setRecordRoute(true);
    proxy.proxyTo(dest);
         }
   else
   {
             //log("SimpleProxy: doInvite: Request is invalid. Did not contain a To: field.");
    SipServletResponse sipresponse = request.createResponse(400);
    sipresponse.send();
   }
  }
  else
  {
         //log("SimpleProxy: doInvite: target refresh, let container handle invite");
   super.doInvite(request);
  }
 }
 catch (Exception e){
  e.printStackTrace();
 }
}

/* (non-Javadoc)
 * @see javax.servlet.sip.SipServlet#doResponse(javax.servlet.sip.SipServletResponse)
 */
protected void doResponse(SipServletResponse response) throws ServletException,
  IOException {
  super.doResponse(response);
```

```
   // Example of using the session object to store session state.
  SipSession session = response.getSession();
  if (session.getAttribute(SHUTDOWN_KEY) != null)
  {
   //log("SimpleProxy: doResponse: invalidating session");
   session.invalidate();
  }
 }

 /* (non-Javadoc)
  * @see javax.servlet.sip.SipServlet#doBye(javax.servlet.sip.SipServletRequest)
  */
 protected void doBye(SipServletRequest request) throws ServletException,
   IOException {

  SipSession session = request.getSession();
  session.setAttribute(SHUTDOWN_KEY, new Boolean(true));

        //log("SimpleProxy: doBye: invalidate session when responses is received.");
  super.doBye(request);
 }

 protected String format_lookup(String toFormat){
  int start_index = toFormat.indexOf('<') + 1;
  int end_index = toFormat.indexOf('>');

  if(start_index == 0){
   //don't worry about it
  }
  if(end_index == -1){
   end_index = toFormat.length();
  }

  return toFormat.substring(start_index, end_index);
 }
}
```

Example: SIP servlet SendOnServlet class

The SendOnServlet class is a simple SIP servlet that would perform the basic function of being called on each INVITE and sending the request on from there.

SendOnServlet class

Function could easily be inserted to log this invite request or reject the INVITE based on some specific criteria.

```
package com.example;
import java.io.IOException;
import javax.servlet.sip.*;
import java.servlet.ServletException;
public class SendOnServlet extends SipServlet {
  public void doInvite(SipServletRequest req)
       throws ServletException, java.io.IOException {
     //send on the request
     req.getProxy().proxyTo(req.getRequestURI);
  }
}
```

The doInvite method could be altered to do something such as reject the invite for some specific criteria simply. In the example doInvite method below, all requests from domains outside of example.com will be rejected with a Forbidden response.

```
   public void doInvite(SipServletRequest req)
throws ServletException, java.io.IOException {
if (req.getFrom().getURI().isSipURI()){
```

```
    SipURI uri = (SipURI)req.getFrom.getURI();
    if (!uri.getHost().equals("example.com")) {
  //send forbidden response for calls outside domain
req.createResponse(SipServletResponse.SC_FORBIDDEN, "Calls outside example.com not accepted").send();
return;
    }
 }
 //proxy all other requests on to their original destination
 req.getProxy().proxyTo(req.getRequestURI());
   }


SendOnServlet deployment descriptor:
<sip-app>
    <display-name>Send-on Servlet</display-name>
    <servlet>
        <servlet-name>SendOnServlet</servlet-name>
        <servlet-class>com.example.SendOnServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>SendOnServlet</servlet-name>
        <pattern>
              <equal>
                <var>request.method</var>
                <value>INVITE</value>
              </equal>
        </pattern>
    </servlet-mapping>
</sip-app>
```

Example: SIP servlet Proxy servlet class

Proxy servlet class

After the initial INVITE, this application will be called on every subsequent SIP message. For each Request and Response, this class will simply print out the action and who it is to or from.

```
package com.example;
import java.io.IOException;
import javax.servlet.sip.*;
import java.servlet.ServletException;
public class ProxyServlet extends SipServlet {
  public void doInvite(SipServletRequest req)
       throws ServletException, java.io.IOException {
     //get the Proxy
     Proxy p=req.getProxy();
     //turn on supervised mode so that all events come through us
     //The default on this is true but it is set to emphasize the function.
     p.setSupervised(true);
     //set record route so we see the ACK, BYE, and OK
     p.setRecordRoute(true);
     //proxy on the request
     p.proxyTo(req.getRequestURI());
  }
public void doRequest(SipServletRequest req)
     throws ServletException, java.io.IOException {
   System.out.println(req.getMethod()+" Request from "+req.getFrom().getDisplayName());
   super.doRequest(req);
}
public void doResponse(SipServletResponse resp)
     throws ServletException, java.io.IOException {
   System.out.println(resp.getReasonPhrase()+" Response from "+resp.getTo().getDisplayName());
   super.doResponse(resp);
}
}
```

```
Proxy deployment descriptor
<sip-app>
    <display-name>ProxyServlet</display-name>
    <servlet>
        <servlet-name>ProxyServlet</servlet-name>
        <servlet-class>com.example.ProxyServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ProxyServlet</servlet-name>
        <pattern>
              <equal>
                <var>request.method</var>
                <value>INVITE</value>
              </equal>
        </pattern>
    </servlet-mapping>
</sip-app>
```

Deploying SIP applications

Use the administrative console to customize your Session Initiation Protocol (SIP) application installation

About this task

When you deploy a Session Initiation Protocol (SIP) application, you can perform various tasks such as installing, starting, stopping, upgrading, and uninstalling the application.

SIP applications are installed as Java Platform, Enterprise Edition (Java EE) applications. You can deploy a SIP application from a graphical interface or from a command line.

Deploying SIP applications through the console

You can deploy a Session Initiation Protocol (SIP) application through the administrative console.

Before you begin

SIP applications are deployed as Java 2 Platform Enterprise Edition (J2EE) applications. In order to process requests, a virtual host must be defined when deploying the SIP application. If there is no virtual host defined for the configured SIP container listen port, the installed application will be inaccessible.

1. Open the administrative console.

   In a browser, go to URL `http://`*`hostname`*`:9090/admin`, where *hostname* is the name of the host computer. Enter the appropriate login information, and click **OK**.
2. In the left frame click **Applications** → **Install New Application**.
3. Browse and select a SAR file. Specify the context root, beginning with a slash (/), in the **Context Root** field. For example, if your application is named ThisApplication, type `/ThisApplication`.
4. Click **Next** (under the **Context Root** field not beside the WebSphere Status title). If the SAR file has been assembled correctly, the screen will still have the title “Preparing for the application installation”, but the content will change. If an error message appears, check the contents of the SAR file; in particular, verify the web.xml file contents, and try to reload the SAR file.
5. Click **Next**. If you see a screen indicating “Application Security Warnings”, click **Continue**.
6. The **Install New Application** screen should appear with “Step 1: Select application options” highlighted. Select the options you need and click **Next**.
7. “Step 2: Map modules to servers” should appear highlighted now. You can choose the cluster or server where you want to install the application’s modules.
   - If you are installing the application in a stand-alone system, click **Next**.

- If you are installing the application in a clustered system, select **WebSphere:cell=*cellname*,cluster=*cluster_name*** in the **Clusters and Servers** field, select the check box beside the Web module that you want to install, and click **Apply** and **Next**.

8. Now “Step 3: Map virtual hosts for Web modules” should appear highlighted. To the right of the application name there should be a drop-down labeled **Virtual Host**.
   - If you are installing the application in a standalone system, set the value of the drop-down to **default_host**, and click **Next**.
   - If you are installing the application in a clustered system, set the value of the drop-down to the name of the virtual host that was chosen during setup, and click **Next**.

   Note: You must define a virtual host for your configured SIP container listen port or else you will not be able to access the application.
9. You should now see “Step 4: Summary” highlighted. In the right panel you will see a **Summary of installation options** table that details your selected options and their values. If you need to change an option, click **Previous** to return to the section where you can make your change. Click **Finish** to install the application with your settings. The screen should display, `Application` *appname*`_sar installed successfully`, where *appname* is the name of the application.
10. Click the **Save to Master Configuration** link. A Save to Master Configuration window appears.
11. In the Save to Master Configuration window, click **Save**. The application has now been saved in the current configuration.
12. To confirm that the installation succeeded, in the left frame click **Applications** → **Enterprise Applications**. The newly installed application should appear in the list of installed applications as *appname*_sar.
13. To start the application so that it can service SIP requests, check the box beside *appname*_sar, and click **Start**. You might also want to look at the logs for a successful startup message.

Results

The application can service SIP requests now.

Deploying SIP applications through scripting

You can deploy a Session Initiation Protocol (SIP) application not only from the GUI but also from the command line.

- Launch a scripting client. For more information, see AdminApp object for scripted administration.
- List applications.
- Install standalone archive files. For more information about installation, see Installation options for the AdminApp object.
- Edit application configurations.
- Uninstall applications.

Troubleshooting SIP applications

Use this page to troubleshoot SIP applications.

About this task

SIP container troubleshooting basics

- The Average CPU usage of the system should go no higher than 60%-70%.
- The container should use no more than 70% of the allocated VM heap size. Be sure that the system has enough physical memory to accommodate the VM heap size. Call loads and session timeouts will have a big affect on heap usage.

- The maximum garbage collection (GC) time of the VM on which the container is running should not exceed 500 ms and the average should be less than 400 ms. Verbose GC can be used to measure this and the PMI viewer can be used to view GC times, heap usage, active sessions, etc., in graphical form.

Initial troubleshooting checklist:

- Check the listening ports in the configuration.
- Use netstat –an to see listening ports.
- Check to see if virtual hosts are defined
- Check to see if host aliases are defined.
- Is an application installed? Is it started?
- For a proxy configuration: Is a default cluster configured? If proxy and server are on the machine, is there a port conflict?

Results

SIP container symptoms and solutions

If the problem is not resolved, check for specific symptoms.

- **Symptom:** Lots of retransmissions, CPU periodically drops to zero.

  Solution: This is typically a DNS issue caused by Reverse DNS lookups and can be confirmed using a tool like Ethereal. If you do a network capture and send lots of DNS queries that contain an IP address and get back a host name in the response, this could be the problem. Make sure that nscd is running if you are on HP or other platforms that require name service caching. (Windows does not require this.) Another solution is to add host names to the /etc/hosts file.
- **Symptom:** Lots of retransmissions, CPU periodically spikes to 100%.

  Solution: This is typically due to garbage collection and can be verified by turning on verbose GC (accessible on the admin console) and looking at the length of the GC cycles. The solution here is to enable Generational Garbage Collection by setting the JVM optional args to -Xgcpolicy:gencon.
- **Symptom:** Lots of retransmissions, CPU spikes to 100% for long periods of time and Generational Garbage Collection is enabled.

  Solution: This is typically due to SIP session objects either not being invalidated or not timing out over a long period of time. One solution is to set the session timeout value in the sip.xml of the application to a smaller value. The most efficient way to handle this is for the application to call invalidate on the session when the dialog completes (i.e. after receiving a BYE). The following entry in the SystemOut.log file will indicate the session timeout value is for each application installed on the container:

```
SipXMLParser  3 SipXMLParser getAppSessionTTL Setting Expiration time: 7 Minutes, For  App: TCK back-to-back user agent"
```

- **Symptom:** Lots of “480 Service Not Available” messages received from the container when sending new INVITE messages to the SIP container. You will also likely see the following message show up in the SystemOut.log when the server is in this state: ″LoadManager E LoadManager warn.server.oveloaded″.

  Solution: This is typically due to one of the SIP container configurable metrics being exceeded. This includes the ″Maximum Application Sessions″ value and the ″Maximum messages per averaging period″ value. The solution is to adjust these values higher.
- **Symptom:** Lots of resends and calls are not completing accompanied by OutOfMemory exceptions in the SystemErr.log.

  Solution: This usually means that the VM heap size associated with your container is not large enough and should be adjusted upwards. You can adjust this value from the admin console.
- **Symptom:** You receive a “503 Service Unavailable” when sending a SIP request to a SIP proxy.

  Solution: This usually means there is no default cluster (or cluster routing rule that matches the message) set up at the proxy. This can also happen when the SIP proxy is configured well but the backend SIP containers are stopped or have crashed.

- **Symptom:** You receive a “404 Not Found” when sending a SIP request to a SIP proxy.

  Solution: This usually means there is no virtual host set up for the containers that reside in the default cluster. It could also mean that the servers in the proxy’s default cluster do not contain a SIP application or that the message does not match one of the applications installed in the default cluster.
- **Symptom:** An ″out of memory″ type behavior is occurring.

  Solution: This may be due to the maximum heap size being set too low. SIP applications can consume a significant amount of memory because the sessions exist for a long call hold time. The maximum heap size of 512 MB does not provide sufficient memory for the SIP traffic workload. Set the maximum heap size for SIP applications to the minimum recommended value of 768 MB or higher.
- **Symptom:** You receive a “403 Forbidden” when sending a SIP request to a SIP container.

  Solution: This usually means there is no appropriate SIP application found to handle the received SIP request (no match rule that matched the message).

Chapter 4. EJB applications

Task overview: Using enterprise beans in applications

This article provides an overview of the tasks you must perform to use enterprise beans in a Java-based application.

About this task

Use the following steps to develop an Enterprise JavaBeans (EJB) application:

1. **EJB 3.0 beans:** Design a Java Platform, Enterprise Edition (Java EE) application and the enterprise beans that it needs.
2. **EJB 2.x beans:** Design a Java 2 Platform, Enterprise Edition (J2EE) application and the enterprise beans that it needs.
3. Develop any enterprise beans that your application uses.
4. Prepare for assembly. For your EJB 2.x-compliant entity beans, decide on an appropriate access intent policy.
5. Assemble the beans into one or more EJB modules using one of the assembly tools. This process includes setting security. For your EJB 2.x-compliant entity beans, you might also want to designate container-managed persistence (CMP) sequence groups.
6. **EJB 3.0 beans:** Assemble the beans into one or more EJB 3.0 modules using one of the assembly tools.
7. Assemble the modules into a J2EE application using the assembly tool.
8. For a given application server, update the EJB container configuration if needed for the application to be deployed.
9. For a given application server, update the EJB container configuration if needed for the application to be deployed, and determine if you want to batch commands or defer commands for container-managed persistence.
10. Deploy the application in an application server.
11. Test the modules.
    - As needed, debug problems with the container.
    - Debug access problems.
12. Assemble the production application using one of the assembly tools
13. Deploy the application to a production environment.
14. Manage the application:
    a. Manage installed EJB modules. After an application has been installed, you can manage its EJB modules individually through assembly tools.
    b. Manage other aspects of the Java application.
15. Update the module and redeploy it using one of the assembly tools.
16. Tune the performance of the application. See Best practices for developing enterprise beans.

Enterprise beans

An enterprise bean is a Java component that can be combined with other resources to create Java applications. There are three types of enterprise beans, *entity* beans, *session* beans, and *message-driven* beans.

All beans reside in Enterprise JavaBeans (EJB) containers, which provide an interface between the beans and the application server on which they reside.

EJB 2.1 and earlier versions of the specification define entity beans as a means to store permanent data, so they require connections to a form of persistent storage. This storage might be a database, an existing legacy application, a file, or another type of persistent storage.

The EJB 3.0 specification deprecates EJB 1.1-style entity beans. The Java Persistence API (JPA) specification is intended to replace the deprecated enterprise beans. While the JPA replacement is called an entity class, it should not be confused with entity enterprise beans. A JPA entity is not an enterprise bean and is not required to run in an EJB container.

Session beans typically contain the high-level and mid-level business logic for an application. Each method on a session bean performs a particular high-level operation. For example, submitting an order or transferring money between accounts. Session beans often invoke methods on entity beans in the course of their business logic.

Session beans can be either *stateful* or *stateless*. A stateful bean instance is intended for use by a single client during its lifetime, where the client performs a series of method calls that are related to each other in time for that client. One example is a *shopping cart* where the client adds items to the cart over the course of an online shopping session. In contrast, a stateless bean instance is typically used by many clients during its lifetime, so stateless beans are appropriate for business logic operations that can be completed in the span of a single method invocation. Stateful beans should be used only where absolutely necessary. Using stateless beans improves the ability to debug, maintain, and scale the application.

The EJB 3.0 specification supports stateless and stateful session beans. They follow a simple pattern such as:

- Define the business interface.
- Define the class that implements it.
- Add metadata with annotations or with XML deployment descriptors.

The end result of a simple EJB 3.0 stateful session bean looks like the following:

```
package ejb3demo;

@Stateful
public class Cart3Bean implements ShoppingCart {
    private ArrayList contents = new ArrayList();

public void addToCart (Object o) {
  contents.add(o);
    }

public Collection getContents() {
    return contents;

    }
}
```

EJB components can use annotations such as @EJB and other injectable @Resource references if the module is an EJB 3.0 module.

Web application clients and application clients can use deployment descriptor-defined EJB references. If the reference is for an EJB 3.0 session bean without a home interface, the reference should be defined with a null <home> or <local-home> setting in the deployment descriptor.

Web application clients and application clients can also use @EJB injections for references to EJB session beans within the same enterprise archive (EAR) file, but the binding must either use the AutoLink support within the container or the annotation must use the name of the reference that is defined by the deployment descriptor and bound when the application is installed. For more information about AutoLink, see the topic, ″EJB 3.0 application bindings support.″

Message-driven beans enable asynchronous message servicing.

- The EJB container and a Java Message Service (JMS) provider work together to process messages. When a message arrives from another application component through JMS, the EJB container forwards it through an onMessage method call to a message-driven bean instance, which then processes the message. In other respects, message-driven beans are similar to stateless session beans.
- The EJB container and a Java Connector Architecture (JCA) resource adapter work together to process messages from an enterprise information system (EIS). When a message arrives from an EIS, the resource adapter receives the message and forwards it to a message-driven bean, which then processes the message. The message-driven bean is provided services such as transaction support by the EJB container in the same way that other enterprise beans are provided service.

Beans that require data access use *data sources*, which are administrative resources that define pools of connections to persistent storage mechanisms.

EJB modules

An Enterprise JavaBeans (EJB) module is used to assemble one or more enterprise beans into a single deployable unit. An EJB module is stored in a standard Java archive (JAR) file.

An EJB module contains the following:

- One or more deployable enterprise beans.
- A deployment descriptor, stored in an Extensible Markup Language (XML) file. This file declares the contents of the module, defines the structure and external dependencies of the beans in the module, and describes how the beans are to be used at run time.

  It is not necessary to use XML deployment descriptors in EJB 3.0 modules, although XML descriptors are supported. Instead of deployment descriptors, you can use annotations to provide component metadata.

You can deploy an EJB module as a stand alone application, or combine it with other EJB modules or with Web modules to create a Java application. An EJB module is installed and run in an enterprise bean container.

If you want to package an EJB 3.0 module with a deployment descriptor, there are several ways to do it. You can package an EJB 3.0 module with an EJB 3.0 style session and/or message-driven beans exclusively; with an EJB 2.1 style session and/or message-driven beans exclusively, or a combination of 2.1 and 3.0 style beans. The XML deployment descriptor must be a Version 3.0 deployment descriptor. It is required that 2.1 entity beans are packaged in modules with 2.1 deployment descriptors.

EJB modules that contain EJB 3.0 beans must be at the EJB 3.0 specification level when running on the product. To set the EJB module to support EJB 3.0 beans, you can set the ejb-jar.xml deployment descriptor level to 3.0, or you can make sure that the module does not contain an ejb-jar.xml deployment descriptor. If the module level is EJB 2.1 or earlier, no EJB 3.0 functions, including annotation scanning or resource injection is performed at runtime.

For more information about packaging and deployment of EJB 3.0 beans, see the topic EJB 3.0 module packaging overview.

EJB containers

An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean's operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

One or more EJB modules, each containing one or more enterprise beans, can be installed in a single container.

The EJB container provides many services to the enterprise bean, including the following:
- Beginning, committing, and rolling back transactions as necessary.
- Maintaining pools of enterprise bean instances ready for incoming requests and moving these instances between the inactive pools and an active state, ensuring that threading conditions within the bean are satisfied.
- Most importantly, automatically synchronizing data in an entity bean's instance variables with corresponding data items stored in persistent storage.

By dynamically maintaining a set of active bean instances and synchronizing bean state with persistent storage when beans are moved into and out of active state, the container makes it possible for an application to manage many more bean instances than could otherwise simultaneously be held in the application server's memory. In this respect, an EJB container provides services similar to virtual memory within an operating system.

WebSphere Application Server provides significant flexibility in the management of database data with entity beans. The Entity EJBs Activate at and Load at configuration settings specify how and when to load and cache data from the corresponding database row data of an enterprise bean. These configuration settings provide the capability to specify enterprise bean caching Options A, B or C, as specified in the EJB 1.1 specifications. You can configure these settings with assembly tools. To read more about how to use the assembly tools see the assembly tool information center.

Between transactions, the state of an entity bean can be cached. The EJB container supports option A, B, and C caching.
- With option A caching, the application server assumes that the entity bean is used within a single container. Clients of that bean must direct their requests to the bean instance within that container. The entity bean has exclusive access to the underlying database, which means that the bean cannot be cloned or participate in workload management if option A caching is used.

  If you intend to use read-only scenarios, the product provides an alternate, higher-performance variation of option A entity beans. This caching option is called *Multithreaded Read-Only*. Similar to standard option A behavior, the EJB container continues to activate the bean just once and leave it active until the EJB container needs space in its active instance cache. However, the EJB container differs from standard option A in the following behaviors:
  - It reloads the state of the bean from persistent storage periodically in response to the user invoking a method on it to pick up any changes that may have been made to the persistent store since the last time the bean was loaded. You can configure this function through a *Reload Interval* setting in the bean's deployment descriptor. For more information, see "Developing read-only entity beans" on page 173.
  - The state of the bean is not written to persistent store by the EJB container at the end of the transaction, nor is the bean's ejbStore() method be invoked.
  - The EJB container permits method invocations from more than one client (thread) on the same bean instance. This differs from the standard EJB component for the internals of a bean. You must keep this aspect in mind when developing your bean, and ensure that any logic in the bean's business methods is overall thread-safe.
- With option B caching, the entity bean remains active in the cache throughout the transaction but is reloaded at the start of each method call.
- With option C caching (the default), the entity bean is always reloaded from the database at the beginning of each transaction. A client can attempt to access the bean and start a new transaction on any container that has been configured to host that bean. This is similar to the session clustering facility described for HTTP sessions in that the entity bean's state is maintained in a shared database that can be accessed from any server when required.

Option A provides maximum enterprise bean performance by caching database data outside of the transaction scope. Generally, Option A is only applicable where the EJB container has exclusive access to the given database. Otherwise, data integrity is compromised. Option B provides more aggressive caching

of Entity EJB object instances, which can result in improved performance over Option C, but also results in greater memory usage. Option C is the most common real-world configuration for Entity EJBs and is the default setting.

The Activate at setting specifies the point at which an enterprise bean is activated and placed in the cache. Removal from the cache and passivation are also governed by this setting. Valid values are Once and Transaction. The Once setting indicates that the bean is activated when it is first accessed in the server process, and passivated (and removed from the cache) at the discretion of the container, for example when the cache becomes full. The Transaction setting indicates that the bean is activated at the start of a transaction and passivated (and removed from the cache) at the end of the transaction. The default value is Transaction.

The Load at setting specifies when the bean loads its state from the database. The value of this property implies whether the container has exclusive or shared access to the database. Valid values are Activation and Transaction. Activation indicates the bean is loaded when it is activated and implies that the container has exclusive access to the database. Transaction indicates that the bean is loaded at the start of a transaction and implies that the container has shared access to the database. The default is Transaction. The settings of the Activate at and Load at properties govern which commit options are used. For Option A (exclusive database access), use Activate at = Once and Load at = Activation. This option reduces database input/output by avoiding calls to the ejbLoad function, but serializes all transactions accessing the bean instance. Option A can increase memory usage by maintaining more objects in the cache, but can provide better response time if bean instances are not generally accessed concurrently by multiple transactions.

Note: When using WebSphere Network Deployment and workload management is enabled, Option A cannot be used.

You must use settings that result in the use of Options B or C. For Option B (shared database access), use Activate at = Once and Load at = Transaction. Option B can increase memory usage by maintaining more objects in the cache. However, because each transaction creates its own copy of an object, there can be multiple copies of an instance in memory at any given time (one per transaction), requiring the database be accessed at each transaction. If an enterprise bean contains a significant number of calls to the ejbActivate function, using Option B can be beneficial because the required object is already in the cache. Otherwise, this option does not provide significant benefit over Option A. For Option C (shared database access), use Activate at = Transaction and Load at = Transaction. Load at = Transaction. This option can reduce memory usage by maintaining fewer objects in the cache. However, there can be multiple copies of an instance in memory at any given time (one per transaction). This option can reduce transaction contention for enterprise bean instances that are accessed concurrently but not updated.

This product supports the cloning of stateful session bean home objects among multiple application servers. However, it does not support the cloning of a specific instance of a stateful session bean. Each instance of a particular stateful session bean can exist in just one application server and can be accessed only by directing requests to that particular application server. State information for a stateful session bean cannot be maintained across multiple members of a server cluster. However, enabling stateful session bean failover and configuring the EJB container to use memory-to-memory replication does enable stateful session bean failover to be replicated to other servers in the cluster so that failover can occur to the backup server if the primary server for a stateful session bean stops for some reason. For more information about stateful session bean failover, see Stateful session bean failover for the EJB container.

By default, an EJB container runs in the **quick start** mode. The EJB container startup logic delays the loading and processing of all EJB types *except* Message Driven Beans, because message driven beans must exist before messages are posted for them; Startup Beans, which must be processed when the server starts; and EJB types that you specify to initialize when the server starts. .

All other EJB initialization is delayed until the first use of the EJB type. When using local interfaces, the first use is when you perform an InitialContext.lookup method for the type. For remote interfaces, it is when you call the first method on an EJB or its Home.

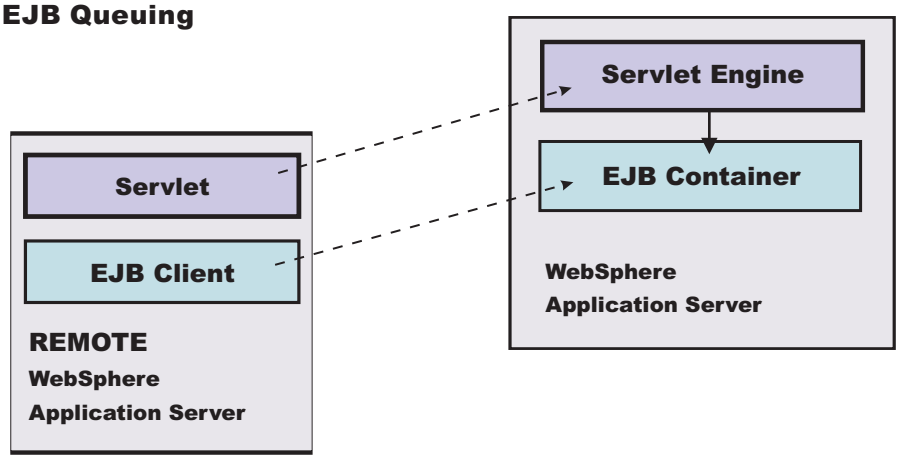
EJB method Invocation Queuing

Method invocations to enterprise beans are only queued for remote clients making the method call. An example of a remote client is an Enterprise JavaBeans (EJB) client running in a separate Java virtual machine (JVM) (another address space) from the enterprise bean. In contrast, no queuing occurs if the EJB client, either a servlet or another enterprise bean, is installed in the same JVM on which the EJB method runs, and on the same thread of execution as the EJB client.

Remote enterprise beans communicate by using the Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). Method invocations initiated over RMI-IIOP are processed by a server-side object request broker (ORB). The thread pool acts as a queue for incoming requests. However, if a remote method request is issued and there are no more available threads in the thread pool, a new thread is created. After the method request completes the thread is destroyed. Therefore, when the ORB is used to process remote method requests, the EJB container is an open or closed queue, due to the use of unbounded threads.

The following illustration depicts the two queuing options of enterprise beans.

EJB Queuing



Enterprise bean and EJB container troubleshooting tips

If you are having problems starting an Enterprise JavaBeans (EJB) container, or encounter error messages or exceptions that appear to be generated on by an EJB container, follow these steps to resolve the problem:

- Use the administrative console to verify that the application server which hosts the container is running.
- Browse the logs for the application server which hosts the container. Look for the message **server *server_name* open for e-business** in the server log files. If it does not appear, or if you see the message **problems occurred during startup**, browse the server log files for details.

If none of these steps solves the problem, check to see if the problem is identified and documented using the links in Diagnosing and fixing problems: Resources for learning. If you do not see a problem that resembles yours, or if the information provided does not solve your problem, contact IBM support for further assistance.

For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

Application client log error indicates missing JAR file

The following error message appears in the client log file because a Java archive (JAR) file is missing from the classpath on the client machine. The Object Request Broker (ORB) needs this file to unmarshal the nested exception that is part of the EJB exception, returned by the server to the client application. For example, if the EJB returns a DB2® JCC SQL exception nested inside of the EJB exception that it returns to the client, the ORB is not able to unmarshal the nested exception if the db2jcc.jar file that contains the DB2 SQL exception is not in the client classpath.

```
java.rmi.MarshalException: CORBA MARSHAL 0x4942f89a No; nested exception is:
org.omg.CORBA.MARSHAL: Unable to read value
from underlying bridge : Custom marshaling (4) Sender's class does not match
local class  vmcid: 0x4942f000  minor  code: 2202  completed: No*
```

To avoid this error, include the JAR file that contains the class for the nested exception that is returned in the EJB exception.

Enterprise bean cannot be accessed from a servlet, a JSP file, a stand-alone program, or another client

Use these troubleshooting tips for problems related to accessing enterprise beans.

What kind of error are you seeing?

- **javax.naming.NameNotFoundException: Name *name* not found in context ″local″ message** when access is attempted
- **BeanNotReentrantException** is thrown
- **CSITransactionRolledbackException / TransactionRolledbackException** is thrown
- Call fails, Stack trace beginning **EJSContainer E Bean method threw exception [exception_name]** found in JVM log file.
- Call fails, **ObjectNotFoundException or ObjectNotFoundLocalException** when accessing stateful session EJB found in JVM log file.
- Attempt to start container managed persistence (CMP) Enterprise JavaBeans (EJB) module fails with **javax.naming.NameNotFoundException: *dataSourceName***
-
- **Message BBOT0003W is issued**
- Symptom: **CNTR0001W: A Stateful SessionBean could not be passivated**

If the client is remote to the enterprise bean, which means, running in a different application server or as a stand-alone client, browse the logs of the application server hosting the enterprise bean, as well as log files of the client.

ObjectNotFoundException or ObjectNotFoundLocalException when accessing stateful session EJB

A possible cause of this problem is that the stateful session bean timed out and was removed by the container. This event must be addressed in the code, according to the EJB 2.1 and later specification. You can review the EJB 2.1 and 3.0 specifications at `http://java.sun.com/products/ejb/docs.html`.

Stack trace beginning ″EJSContainer E Bean method threw exception [exception_name]″ found in JVM log file

If the exception name indicates an exception thrown by an IBM class that begins with ″com.ibm...″, then search for the exception name within the information center, and in the online help as described below. If ″exception name″ indicates an exception thrown by your application, contact the application developer to determine the cause.

javax.naming.NameNotFoundException: Name name not found in context ″local″

A possible reason for this exception is that the enterprise bean is not local (not running in the same Java virtual machine [JVM] or application server) to the client JSP, servlet, Java application, or other enterprise bean, yet the call is to a ″local″ interface method of the enterprise bean . If access worked in a development environment but not when deployed to WebSphere Application Server, for example, it might be that the enterprise bean and its client were in the same JVM in development, but are in separate processes after deployment.

To resolve this problem, contact the developer of the enterprise bean and determine whether the client call is to a method in the local interface for the enterprise bean. If so, have the client code changed to call a remote interface method, or to promote the local method into the remote interface.

References to enterprise beans with local interfaces are bound in a name space local to the server process with the URL scheme of `local:`.

BeanNotReentrantException is thrown

This problem can occur because client code, typically a servlet or JSP file, is attempting to call the same stateful SessionBean from two different client threads. This situation often results when an application stores the reference to the stateful session bean in a static variable, uses a global (static) JSP variable to refer to the stateful SessionBean reference, or stores the stateful SessionBean reference in the HTTP session object. The application then has the client browser issue a new request to the servlet or JSP file before the previous request has completed.

To resolve this problem, ask the developer of the client code to review the code for these conditions.

CSITransactionRolledbackException / TransactionRolledbackException is thrown

An enterprise bean container creates these high-level exceptions to indicate that an enterprise bean call did not complete. When this exception is thrown, browse the logs to determine the underlying cause.

Some possible causes include:
- The enterprise bean might throw an exception that was not declared as part of its method signature. The container is required to roll back the transaction in this case. Common causes of this situation are where the enterprise bean or code that it calls creates a NullPointerException, ArrayIndexOutOfBoundsException, or other Java runtime exception, or where a BMP bean encounters a JDBC error. The resolution is to investigate the enterprise bean code and resolve the underlying exception, or to add the exception to the problem method signature.
- A transaction might attempt to do additional work after being placed in a `″Marked Rollback″`, `″RollingBack″`, or `″RolledBack″` state. Transactions cannot continue to do work after they are set to one of these states. This situation occurs because the transaction has timed out which, often occurs because of a database deadlock. Work with the application database management tools or administrator to determine whether database transactions called by the enterprise bean are timing out.
- A transaction might fail on commit due to dangling work from local transactions. The local transaction encounters some ″dangling work″ during commit. When a local transactions encounters an ″unresolved action″ the default action is to ″rollback″. You can adjust this action to ″commit″ in an assembly tool. See the assembly tool information center on how to adjust

Attempt to start EJB module fails with ″javax.naming.NameNotFoundException dataSourceName_CMP″exception

This problem can occur because:
- When the DataSource resource was configured, container managed persistence was not selected.

  - To confirm this problem, in the administrative console, browse the properties of the data source given in the NameNotFoundException. On the Configuration panel, look for a check box labeled **Container Managed Persistence**.
  - To correct this problem, select the check box for **Container Managed Persistence**.
- If container managed persistence is selected, it is possible that the CMP DataSource was not bound into the namespace.
  - Look for additional naming warnings or errors in the status bar, and in the hosting application server logs. Check any further naming-exception problems that you find by looking at the topic Application access problems.

Message BBOT0003W is issued

Message BBOT0003W indicates a transaction timeout. The timeout might result in the abnormal termination of the servant where the transaction is running.

- The default timeout value for enterprise bean transactions is 120 seconds. After this time, the transaction times out and the connection closes.
- If the transaction legitimately takes longer than the specified timeout period, on the administrative console:
  1. Go to **Manage Application Servers > server_name**
  2. Select the **Transaction Service properties** page
  3. Increase the **Total transaction lifetime timeout** value
  4. **Save** the configuration

  Note: z/OS will use the value you set for **Total transaction lifetime timeout** as the default transaction timeout setting. If you set a value for this property that is greater than the maximum transaction timeout value, z/OS will use the maximum transaction timeout value as the default.

Symptom:CNTR0001W: A Stateful SessionBean could not be passivated

This error can occur when a Connection object used in the bean is not closed or nulled out.

To confirm this is the problem, look for an exception stack in the logs for the EJB container that hosts the enterprise bean, and looks similar to:

```
StatefulPassi W CNTR0001W:
A Stateful SessionBean could not be passivated: StatefulBeanO
(BeanId(XXX#YYY.jar#ZZZZ),
state = PASSIVATING)
java.io.NotSerializableException: com.ibm.ws.rsadapter.jdbc.WSJdbcConnection
 at java.io.ObjectOutputStream.outputObject((Compiled Code))
 at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java(Compiled Code))
 at java.io.ObjectOutputStream.outputClassFields((Compiled Code))
 at java.io.ObjectOutputStream.defaultWriteObject((Compiled Code))
 at java.io.ObjectOutputStream.outputObject((Compiled Code))
 at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java(Compiled Code))
 at com.ibm.ejs.container.passivator.StatefulPassivator.passivate((Compiled Code))

 at com.ibm.ejs.container.StatefulBeanO.passivate((Compiled Code)
 at com.ibm.ejs.container.activator.StatefulASActivationStrategy.atUnitOfWorkEnd
                      ((Compiled Code))
 at com.ibm.ejs.container.activator.Activator.unitOfWorkEnd((Compiled Code))
 at com.ibm.ejs.container.ContainerAS.afterCompletion((Compiled Code)
```

where *XXX,YYY,ZZZ* is the Bean's name.

To correct this problem, the application must close all connections and set the reference to null for all connections. Typically this activity is done in the `ejbPassivate()` method of the bean. Also, note that the bean must have code to reacquire these connections when the bean is reactivated. Otherwise, there are NullPointerExceptions when the application tries to reuse the connections.

Developing enterprise beans

One of two enterprise bean development scenarios is typically used with the product. The first is command-line using Ant, Make, Maven or similar tools. The second is an IDE-based development and build environment. The steps in this article focus on development without an IDE.

Before you begin

Enterprise JavaBeans (EJB) 2.x beans only: Design a J2EE application and the enterprise beans that it needs.

- Before developing entity beans with container-managed persistence (CMP), read the topic Concurrency control.

EJB 3.0 beans only: Design a Java EE application and the enterprise beans that it needs.

- Before developing entity beans with CMP, read the topic, ″Concurrency control.″ Keep in mind that EJB 3.0 modules do not support entity beans. You must continue to place entity beans in your EJB 2.x-level modules.

About this task

The following is more information about the two basic approaches to selecting tools for developing enterprise beans:

- You can use one of the available IDE tools that automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. The Rational Application Developer product is the recommended IDE. For more information, see the documentation for that product.

  To use the assembly tools with EJB 3.0 modules, you need to add <WAS_HOME>/lib/j2ee.jar to the project's build path to resolve compilation dependencies on the new EJB 3.0 API classes. Code assist works once this is done. If you define a server (see J2EE Perspective), point the server to the product install directory. Before you create the project, the project automatically refers to <WAS_HOME>/lib/j2ee.jar. Be sure to create the server with the setting *Run server with resources on Server*.
- If you have decided to develop enterprise beans without an IDE, you need at least an ASCII text editor. You can also use a Java development tool that does not support enterprise bean development. You can then use tools available in the Java Software Development Kit (SDK) and in this product to assemble, test, and deploy the beans.

  Like the assembly tool, a standard Java EE command-line build environment requires some change to utilize the EJB 3.0 modules. As with previous Java EE application development patterns, you must include the `j2ee.jar` file located in the `<WAS_HOME>/lib/` directory on the compiler classpath. An example of a command-line build environment using Ant is located in the `<WAS_HOME>/samples/src/TechSamp` directory.

The following steps primarily support the second approach, development without an IDE.

1. If necessary, migrate any pre-existing code to the required version of the EJB specification.

   Applications written to the EJB specification versions 1.1, 2.0, and 2.1 can run unchanged in the EJB 3.0 container.
2. Write and compile the components of the enterprise bean.
   - At a minimum, a session bean developed with the EJB 3.0 specification requires a bean class and a business interface.
   - At a minimum, an EJB 1.1 session bean requires a bean class, a home interface, and a remote interface. An EJB 1.1 entity bean requires a bean class, a primary-key class, a home interface, and a remote interface.
   - At a minimum, an EJB 2.x session bean requires a bean class, a home or local home interface, and a remote or local interface. An EJB 2.x entity bean requires a bean class, a primary-key class, a

remote home or local home interface, and a remote or local interface. The types of interfaces go together: If you implement a local interface, you must also define a local home interface.

Note: Optionally, the primary-key class can be *unknown*. See unknown primary-key class for more information.

- A message-driven bean requires only a bean class.

3. For each entity bean, complete work to handle persistence operations.

   For EJB 3.0 modules, consider using the Java Persistence API (JPA) specification to develop plain old Java Object (POJO) persistent entities. Review the topic ″Java Persistence API″ for more information. If you choose to develop entity beans to earlier EJB specifications, follow the steps below:

   - Create a database schema for the entity bean's persistent data.
     – For entity beans with CMP, you must store the bean's persistent data in one of the supported databases. The assembly tool automatically generates SQL code for creating database tables for CMP entity beans. If your CMP beans require complex database mappings, it is recommended that you use Rational Application Developer to generate code for the database tables. For more information on using the assembly tools see the assembly tool information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html
     – For entity beans with bean-managed persistence (BMP), you can create the database and database table by using the database tools or use an existing database and database table.

     For more information on creating databases and database tables, review your database documentation.
   - **(CMP entity beans for EJB 2.x only)**

     Define finder queries with EJB Query Language (EJB QL).

     Define finder queries with EJB Query Language (EJB QL).

     With EJB QL, you define finders in terms of CMP fields and container-managed relationships, as follows:
     – *Public* finders are visible in the bean's home interface. Implemented in the bean class, they return only remote interfaces and collection types.
     – *Private* finders, expressed as SELECT statements, are used only within the bean class. They can return both local and remote interfaces, dependent values, other CMP field types, and collection types.
   - **(CMP entity beans for EJB 1.1 only: an IBM extension)** Create a finder helper interface for each CMP entity bean that contains specialized finder methods (other than the findByPrimaryKey method).

     The following logic is required for each finder method (other than the findByPrimaryKey method) contained in the home interface of an entity bean with CMP:
     – The logic must be defined in a public interface named *Name*`BeanFinderHelper`, where *Name* is the name of the enterprise bean, for example, AccountBeanFinderHelper.
     – The logic must be contained in a String constant named *`findMethodName`*`WhereClause`, where *findMethodName* is the name of the finder method. The String constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method's arguments when that method is called.

What to do next

Assemble the beans in one or more EJB modules

Assemble the beans in one or more EJB 3.0 modules.

Enterprise JavaBeans (EJB) 3.0 specification

This topic describes the Enterprise JavaBeans (EJB) 3.0 specification that is the foundation of the development and application programming model for the EJB 3.0 applications. Read this topic for a brief overview of the EJB 3.0 specification.

The EJB 3.0 specification has justifiably been called the most important upgrade to the Java Platform, Enterprise Edition 5 (Java EE 5) programming model. The EJB 3.0 specification represents simplification and streamlining of the business logic and persistence programming models used in Java EE. The ultimate source of information is the specification, which is available on the Sun Microsystems, Inc., Web site at http://java.sun.com.

While the Java Persistence API (JPA) replacement is called an entity class, it should not be confused with entity enterprise beans. A JPA entity is not an enterprise bean and is not required to run in an EJB container.

The EJB 3.0 specification is organized into three areas:

- EJB core contracts and requirements
- EJB 3.0 simplified application programming interface (API)
- JPA

The EJB core contracts and requirements defines the service provider interfaces (SPIs) between the enterprise bean instance and the enterprise bean container. This part of the specification also includes the APIs between the enterprise bean provider and the enterprise bean container, protocols, component and container contracts, system level issues, infrastructure services that are provided by the container to the bean and other information about development packaging and deployment for session, message-driven and entity beans.

The EJB 3.0 simplified API provides information about simplifying EJB APIs and SPIs that exist from previous EJB specification versions.

The JPA document introduces the Plain Old Java Object (POJO)-style persistent entity development guidelines.

Another good source for EJB 3.0 information is Mastering Enterprise JavaBeans 3.0, Fourth Edition. This edition features chapters on session beans and message-driven beans, EJB and Java EE integration and advanced persistence concepts. Also included is coverage of the JPA and POJO using entities with the EJB programming model.

EJB 3.0 considerations

When using Enterprise JavaBeans (EJB) 3.0 modules, keep in mind the following considerations.

Version 7.0 does not support entity beans in EJB 3.0-level modules

IBM® WebSphere® Application Server Version 7.0 does not support the use of bean managed persistence (BMP) and container managed persistence (CMP) entity beans in EJB 3.0-level modules. BMP entity beans are supported in the Feature Pack for EJB 3.0, although CMP beans are not. EJB entity beans may still be used on V7.0, but they must be packaged in an EJB 2.1 or earlier-level module.

Java Platform, Enterprise Edition (Java EE) applications that are packaged with EJB entity beans in EJB 3.0-level modules fail to install on V7.0.

An EJB Java archive (JAR) file is considered to be an EJB 3.0 module when either of the following are true:

- The EJB JAR file contains configuration data in an ejb-jar.xml file with the following EJB 3.0 header specification:

```
<ejb-jar id="ejb-jar_ID" version="3.0"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
```

- The EJB JAR file contains beans with EJB 3.0-style source annotations that provide configuration data.

You will need to repackage your EJB 3.0 modules using EJB 2.x and earlier modules. Otherwise, the installation of any applications that contain entity beans will fail.

Annotations

Consider if you will use annotations versus deployment descriptors, or both. See the topic ″EJB 3.0 metadata annotations″ for more information about annotations.

EJB module

WebSphere Application Server Version 7.0 supports EJB module Java archive (JAR) files with an ejb-jar.xml deployment descriptor declared at the 1.1, 2.0, 2.1, or 3.0 level, or with no ejb-jar.xml deployment descriptor present. If no deployment descriptor is present, the EJB module is assumed to be at the 3.0 level or greater.

EJB modules that contain EJB 3.0 beans must be declared to be at the EJB 3.0 level. This can be accomplished either by setting the ejb-jar.xml deployment descriptor level to 3.0, or ensuring that the module does not contain an ejb-jar.xml deployment descriptor. If the module level is 2.1 or earlier, no EJB 3.0-specific functions such as annotation scanning or resource injection will be performed.

Entity beans are not supported in EJB 3.0 level modules. You must place any entity beans in EJB modules at the 2.1 or earlier level.

Java EE application client module

The product provides support for Java EE application client modules. Additionally, it supports injection of EJB references into client components if the injection is defined through the @EJB annotation.

Note: EJB 3.0 does not support the injection of an enterprise bean that creates a new enterprise bean of itself. Do not inject an enterprise bean that creates a new enterprise bean of itself.

Defining an ejb-ref reference to an EJB 3.0 business interface from a Java EE client component descriptor

It is possible to define an ejb-ref from an application-client.xml descriptor that points to an EJB 3.0 business interface. EJB 3.0 business interfaces are accessed directly without the use of a home, yet the ejb-ref element in Java EE requires that a home interface type be specified. Therefore, you must include the <home></home> stanza in the ejb-ref definition, but specify a null value as shown in the example below. For the value of the <remote> stanza, specify the EJB 3.0 business interface class name. Finally, when you set the binding value, either during application install or through tooling, specify the location where the EJB 3.0 business interface was bound.

For example, the ejb-ref in your client component's application-client.xml file will look similar to the following:

```
<ejb-ref id="EJBRef_1">
 <ejb-ref-name>java_comp-env_name_of_ref</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home></home>
 <remote>com.ejbs.business.interface.class.name</remote>
</ejb-ref>
```

The corresponding section of the ibm-application-client-bnd.xmi file looks similar to the following. A default EJB binding pattern is used here; the default EJB binding conventions are described in the topic, ″EJB 3.0 applications binding support.″

```
<ejbRefBindings xmi:id="EjbRefBinding_1" jndiName=EJB3App/EJB3Mod.jar/MyBean##com.ejbs.business.interface.class.name">
 <bindingEjbRef href="application-client.xml#EjbRef_1"/>
</ejbRefBindings>
```

EJB 3.0 metadata annotations

Annotations enable you to write metadata for Enterprise JavaBeans (EJB) inside your source code. You can use them instead of extensible markup language (XML) deployment descriptor files. Annotations can also be used *with* descriptor files.

If you installed the Feature Pack for EJB 3.0, the default was to scan annotations during the installation of an EJB 3.0 module. For WebSphere Application Server, Version 7.0, the default is not to scan pre-Java EE 5 modules during the application install or at server startup

To preserve backward compatibility with both the Feature Pack for EJB 3.0 and the Feature Pack for Web Services, you have a choice whether or not to scan legacy Web modules for additional metadata. A server level switch is defined for each feature pack scan behavior. If the default is not appropriate, the switch must be set on each server and administrative server that requires a change in the default. The switches are server custom properties com.ibm.websphere.webservices.UseWSFEP61ScanPolicy={true|false} and com.ibm.websphere.ejb.UseEJB61FEPScanPolicy={true|false}. To define these properties in the administrative console click **Application servers** → **server name** → **Process definition** → **Java Virtual Machine** → **Custom properties**.

The product also provides default values for most of the EJB annotations it uses. In many cases, omitting an annotation implies that you want to use the default value.

For the most part, annotations are found in the *javax.ejb* and *javax.persistence* packages.

Annotation type
ExcludeDefaultInterceptors
ApplicationException
AroundInvoke
EJB
EJBs
ExcludeDefaultInterceptors
ExcludeDefaultInterceptors
Init
Interceptors
Local
LocalHome
MessageDriven
PersistenceUnit
PostActivate
PostConstruct
PreDestroy
PrePassivate
Remote
RemoteHome
Remove
Resource

Annotation type
Stateful
Stateless
Timeout
TransactionAttribute
TransactionManagement

EJB 3.0 interceptors

An interceptor is a method that is automatically called when the business methods of a bean are invoked.

You can define interceptors for session or message-driven beans. Business method interceptors can be defined to apply to all business methods of a bean class or to specific business methods only.

You denote interceptor classes using the Interceptor annotation on the bean class, or in the deployment descriptor using the <interceptor> element. Interceptor methods are marked with the AroundInvoke annotation or with the <around-invoke> element of the deployment descriptor. The interceptor methods always follow the pattern Object <method_name> (InvocationContext c) throws Exception.

Interceptors that apply to all session and message-driven beans in an ejb-jar file, known as default interceptors, are defined in the deployment descriptor using the <interceptor-binding> element. Interceptor methods can also be applied to specific methods rather than default to all methods or to specific beans by using the Interceptors annotation.

You can define any number of interceptors for a bean class. They are invoked in the order that they are specified. Interceptors defined in external classes are processed before interceptors defined within a bean class.

EJB 3.0 interceptors

An interceptor is a method that is automatically called when the business methods of a bean are invoked.

You can define interceptors for session or message-driven beans. Business method interceptors can be defined to apply to all business methods of a bean class or to specific business methods only.

You denote interceptor classes using the Interceptor annotation on the bean class, or in the deployment descriptor using the <interceptor> element. Interceptor methods are marked with the AroundInvoke annotation or with the <around-invoke> element of the deployment descriptor. The interceptor methods always follow the pattern Object <method_name> (InvocationContext c) throws Exception.

Interceptors that apply to all session and message-driven beans in an ejb-jar file, known as default interceptors, are defined in the deployment descriptor using the <interceptor-binding> element. Interceptor methods can also be applied to specific methods rather than default to all methods or to specific beans by using the Interceptors annotation.

You can define any number of interceptors for a bean class. They are invoked in the order that they are specified. Interceptors defined in external classes are processed before interceptors defined within a bean class.

Create stubs command

The createEJBStubs command creates stub classes for remote interfaces of EJB version 3.0 beans packaged in Java archive (JAR) or Enterprise archive (EAR) files. It also provides an option to create a single stub class from an interface class located in a directory or a JAR file. Several command options are provided to package the generated stub classes in different ways. See the Syntax and Examples sections below for more details.

This command is found in the <WAS_HOME>/bin directory as:
- createEJBStubs.bat - Windows platforms
- createEJBStubs.sh - Unix based platforms
- createEjbStubs - iSeries platform

The command searches the input JAR or EAR file, looking for EJB version 3.0 modules that contain beans with remote interfaces. When remote interfaces are found, the corresponding stub classes are generated and packaged according to the command options specified. In the case where the input specified is a single interface class, the tool assumes this class is an EJB version 3.0 remote interface class and generates a remote stub class.

For many client-side scenarios, the WebSphere Application Server Just-In-Time (JIT) deployment feature dynamically generates the RMI-IIOP stub classes that are required for invocation of remote EJB 3.0 business interfaces. However, there are some scenarios where the JIT deploy environment is not available to dynamically generate these classes. In these scenarios, the createEJBStubs command must be used instead to generate and embed the client-side stub class files in your client application. If your client environment is one of the following, use the createEJBStubs command:
- ″Bare″ Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.
- A WebSphere Application Server container (web container, EJB container, or application client container) from a version earlier than version 7, or without the Feature Pack for EJB 3.0 applied.
- Non-WebSphere Application Server environments.

Syntax

createEJBStubs *input_ class_ name* | *input_JAR_name* | *input_EAR_name* [**-help**] [**-newfile [new_file]**] [**-updatefile [update_file]**] [**-quiet**] [**-verbose**] [**-logfile log_file**] [**-appendlog**] [**-cp class_path**] [**-trace**]

createEJBStubs

This is the command to create EJB stub classes for a single interface class file, a JAR file, or an EAR file. When invoked without any arguments, or only –help, the createEJBStubs command displays a list of options that can be specified, and a list of example invocations with detailed explanations.

input_class_name **or** *input_EAR_name* **or** *input_JAR_name*

The first parameter is a required element for the command. It must contain the source class, JAR, or EAR file to process.

This parameter may be the fully qualified name of a single interface class (e.g. com.ibm.myRemoteInterface). Note that the package name segments are separated by “.” characters, no path name proceeds the class name, and the “.class” extension is not included. For this interface class input, you must use the class path option (e.g. –cp my_path, or –cp my_path/my_interfaces.jar) to specify where the interface class will be found. The generated stub class will be placed in the package-defined directory structure, starting with the current directory where the command is invoked.

This parameter may also be a JAR or EAR file. In this case the path must be specified (e.g. my_path/my_Server_App.ear). The generated stub classes will be placed in the same module or

modules with the beans, or in the same module or modules with the remote interface classes, depending on whether the –updatefile option is specified. See below for more details.

-help Provides the command syntax, including a list of options that can be specified, and example invocations with detailed explanations.

-newfile [new_file]
Requests that a new file is generated containing the original files in the input JAR or EAR plus the stub classes. When this option is not specified, the stubs are written back into the original JAR or EAR file. If this option is specified, but the new_file name is not provided, a new file name is constructed by appending the input JAR or EAR file name with ″_withStubs″. This option is not allowed when the first input parameter is an interface class.

-updatefile [update_file]
Requests that a second file (e.g. in addition to the input file) is updated with stub classes. This option also provides a different packaging behavior. The stub classes are packaged in the same module or modules as the remote interface classes. By contrast, when this option is not specified, the stub classes are packaged in the same module or modules with the bean classes. If this option is specified, but the update_file name is not provided only the original JAR or EAR file is updated with stub classes. This option is not allowed when the first input parameter is an interface class.

-quiet Requests the suppression of messages. The -quiet option cannot be specified with either the –verbose or the –trace options. Error messages are still displayed.

-verbose
Requests that additional informational messages be output. The -verbose option cannot be specified with either the -quiet or the –trace options.

-logfile log_file
Requests that messages be printed to a log file in addition to the console. If this option is specified, the log_file name must also be provided.

-appendlog
Requests that messages be appended to an existing log file. If this option is specified, the -logfile option must also be specified.

-cp class_path
Requests that the classloader includes the specified the class path where additional class or jar files are located, which are necessary for the remote interface classes to be loaded. The class path may include multiple segments where each path is separated from a previous path by the default path separator character of the operating system. Each path can specify either a JAR file, or a directory. If this option is specified, the class_path name must also be provided.

-trace Request that detailed trace output be generated. This is intended to collect information for use by IBM service to resolve problems. The trace output is English-only. This option cannot be specified with either the –quiet or the –verbose options.

Examples

createEJBStubs com.ibm.myRemoteInterface -cp my_path

Generate the stub class for one remote interface class and place it in the package-defined directory structure, starting at the current directory. The my_path directory will be used as the class path. If the remote interface class to process is in a JAR file, the –cp my_path/my_interfaces.jar syntax must be used for the class path specification.

createEJBStubs my_path/my_beans.jar -newfile –quiet

Generate the stub classes for all level 3.0 enterprise beans in the my_beans.jar file that have remote interfaces. Both the generated stub classes and the original JAR file contents are packaged into a new JAR file named “my_beans_withStubs.jar” because the optional new_file name parameter is not specified along with the –newfile option. Output messages are suppressed except for error notifications.

createEJBStubs my_path/my_Server_App.ear -logfile myLog.out

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are placed into the original EAR file because the –newfile option is not specified. The stub classes are packaged into the same module or modules as the bean classes because the –updatefile option is not specified. Messages are written to both the myLog.out log file and the command window.

createEJBStubs my_path/my_Server_App.ear -updatefile my_path/my_Client_interfaces.jar

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are placed into both the original EAR file and the my_Client_interfaces.jar file. The stub classes are packaged into the same module or modules as the remote interface classes because the –updatefile option is specified.

createEJBStubs my_path/my_Server_App.ear –updatefile

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are only placed into the original EAR file because the optional update_file name parameter is not provided with the –updatefile option. The stub classes are packaged into the same module or modules as the remote interface classes because the -updatefile option is specified.

EJB 3.0 application bindings overview

Before an application that is installed on an application server can start, all Enterprise JavaBeans (EJB) references and resource references defined in the application must be bound to the actual artifacts (enterprise beans or resources) defined in the application server.

When defining bindings, you specify Java Naming and Directory Interface (JNDI) names for the referenceable and referenced artifacts in an application. The jndiName values specified for artifacts must be qualified lookup names.

You do not need to manually assign JNDI bindings names for each of the interfaces or EJB homes on your enterprise beans in EJB 3.0 modules. If you do not explicitly assign bindings, the EJB container assigns default bindings.

For the EJB 3.0 level, the product provides two distinct namespaces for EJB interfaces, depending on whether the interface is local or remote. The same provision applies to EJB homes, which can be considered a special type of interface. The two namespaces are as follows:

- JVM-scoped ejblocal: namespace
- Global JNDI namespace

Local EJB interfaces and homes must always be bound into a JVM-scoped ejblocal: namespace; they are accessible only from within the same application server process.

In contrast, remote EJB interfaces and homes must always be bound into the globally-scoped WebSphere JNDI namespace; they can be accessed from anywhere, including other server processes and other remote clients. Local interfaces cannot be bound into the globally-scoped JNDI namespace, nor can remote interfaces be bound into the JVM-scoped ejblocal: namespace.

The ejblocal: and globally-scoped JNDI namespaces are completely separate and distinct. For example, an EJB local interface bound at ″ejblocal:AccountHome″ is a not at all the same as a remote interface bound at ″AccountHome″ in the globally-scoped namespace. This helps maintain the distinction between your local and remote interface references. Having a JVM-scoped local namespace also makes it possible for your applications to directly look up or reference local EJB interfaces from anywhere in the JVM server process, including across Java Platform, Enterprise Edition (Java EE) application boundaries.

Assigning default JNDI bindings for EJB business interfaces in the EJB 3.0 container with the Application, Module, and Component names

The WebSphere Application Server V7 EJB container assigns default JNDI bindings for EJB 3.0 business interfaces based on Application Name, Module Name, and Component Name, so it is important to understand how these names are defined. Each of these names is a character string.

Java EE applications are packaged in a standardized format called an Enterprise Application Archive (EAR) file. The EAR is a packed file format similar to a .zip or .tar file format, and can thus be visualized as a collection of logical directories and files packed together into a single physical file. Within each EAR file are one or more Java EE module files, which can include:

- Java Application Archive (JAR) files for EJB modules, Java EE application client modules and utility class modules
- Web Application Archive (WAR) files for Web modules
- Other technology-specific modules such as Resource Application Archive (RAR) files and other types of modules

Within each module file are typically one or more Java EE components. Examples of Java EE components are enterprise beans, servlets, and application client main classes.

Since Java EE modules are packaged within Java EE application archives, and Java EE components are in turn packaged within Java EE modules, the ″nesting path″ of each component can be used to uniquely identify every component within a Java EE application archive, according to its application name, module name, and component name.

Application name

The name of an application is defined by the following (in order of priority):

- The value of the ″Application Name″ specified to the product administrative console, or the ″appname″ parameter supplied to the wsadmin command-line scripting tool, during installation of the application into the product.
- The value of the <display-name> parameter within the META-INF/application.xml deployment descriptor for the application.
- The EAR file name, excluding its ″.ear″ file suffix. For example, an application EAR file named CustomerServiceApp.ear would have an application name of ″CustomerServiceApp″ in this case.

Module name

The name of a module is defined as the Uniform Resource Identifier (URI) of the module file, relative to the root of the EAR file in which it resides. Stated another way, the module name is the module's file name relative to the root of the EAR file, including any ″sub-directories″ in which the module file is nested.

In the following example, the CustomerServiceApp application contains three modules whose names are AccountProcessing.jar, Utility/FinanceUtils.jar, and AppPresentation.war:

```
CustomerServiceApp.ear:     AccountProcessing.jar        com/                 mycompany/
               AccountProcessingServiceBean.class            AccountProcessingService.class     Utility/        FinanceUtils.jar
          META-INF/              ejb-jar.xml         com/              mycompany/
 InterestCalculatorServiceBean.class               InterestCalculatorService.class     AppPresentation.war        META-INF/
      web.xml
```

EJB component name

The name of an EJB component is defined by the following, in order of priority:

- The value of the ejb-name tag associated with the bean in the ejb-jar.xml deployment descriptor, if present.
- The value of the ″name″ parameter, if present, in the @Stateless or @Stateful annotation associated with the bean.
- The name of the bean implementation class, without any package-level qualifier.

Bindings

Review the following bindings that are supported by EJB 3.0:

- Default bindings for business interfaces and homes
- Default binding pattern
- User-defined bindings for EJB business interfaces and homes
- User-defined bindings for resolving references and injection targets
- Default resolution of EJB references and EJB injections: The AutoLink feature
- Naming considerations in clustered environments
- User-defined EJB extension settings
- Legacy (XMI) bindings
- User-specified XML bindings

Default bindings for EJB business interfaces and homes

In WebSphere Application Server V7, it is not necessary for you to explicitly define JNDI binding names for each of the your interfaces or EJB homes within an EJB 3.0 module. If you do not explicitly assign bindings, the product's EJB container assigns default bindings using the rules outlined here. This is somewhat different from the EJB support in product prior to the support of the EJB 3.0 specification.

The container performs two default bindings for each interface (business, remote home, or local home) on each enterprise bean. These two bindings are referred to here as the interface's ″short″ binding and its ″long″ binding. The short binding uses just the package-qualified Java class name of the interface, while the long binding uses the enterprise bean's component ID as an extra qualifier before the package-qualified interface class name, with a hash or number sign (# symbol) between the component ID and the interface class name. You can think of the difference between the two forms as being analogous to a ″short″ TCP/IP hostname (just the machine name) versus a ″long″ hostname (machine name with domain name prepended to it).

For example, an interface's short and long default bindings might be ″com.mycompany.AccountService″ and ″AccountApp/module1.jar/ServiceBean#com.mycompany.AccountService″, respectively.

By default, the component ID for EJB default bindings is formed using the enterprise bean's Application Name, Module Name, and Component Name that are defined above, but you can assign any string you want instead. By defining your own string as the component ID, you can set up a naming convention where the enterprise bean's long-form bindings share a common user-defined portion, yet also have a system-defined portion based on the name of each interface class. It also allows you to make the default EJB binding names independent of how you have packaged the enterprise beans within the application/module hierarchy. Overriding an enterprise bean's default component ID is described in the ″User-defined bindings for EJB business interfaces and homes″ section of this topic.

As mentioned earlier in the section on the JVM-scoped local namespace and the globally-scoped JNDI namespace, all local interfaces and homes must be bound into the ejblocal: namespace, which is accessible only within the same server process (JVM), while remote interfaces and homes must be bound into the globally-scoped namespace, which is accessible from anywhere in the WebSphere product cell. As you would expect, the EJB container follows these rules for the default bindings.

In addition, the ″long″ default bindings for remote interfaces follow recommended Java EE best practices in that they are grouped under an ejb context name. By default, EJB remote home and business interfaces are bound into the root of the application server naming context. However, the application server root context is used for binding more than just EJB interfaces, so to keep this context from getting too cluttered, it is a good practice to group EJB-related bindings into a common ″EJB″ sub-context rather than placing them directly in the server root context. It is similar to why you would use subdirectories on a disk volume rather than putting all the files in the root directory.

The short default bindings for remote interfaces are not bound in the ejb context. The short default bindings are located in the root of the server root context. Even though it is a best practice to group all of the EJB-related bindings under an ejb context, there are other considerations including the following:

- The short default bindings provide a simple, direct way to access an EJB interface. Placing them directly in the server root context and referring to them by just the interface name or the interface name prepended with ejblocal: was in keeping with that goal of simplicity.
- At the same time, placing the long default bindings in the ejb context, or the ejblocal: context in the case of a local interface, kept those bindings out of the server's root context and reduced the clutter there enough to allow having the short bindings in the root context.
- It provides a degree of cross-compatibility with other Java EE application servers that use similar naming conventions.

To summarize, all local default bindings, both short and long, are placed in the ejblocal: server/JVM-scoped namespace, while remote default bindings are placed in the server's root context of the globally-scoped namespace if they are short, or in the <server_root>/ejb context (just below the server's root context) if they are long. Thus, the only default bindings in the server's globally-scoped root context are the short bindings for remote interfaces, which is the best balance between providing a simple, portable usage model and keeping the server's globally-scoped root context from becoming too cluttered.

Default binding pattern

The patterns for each type of binding are displayed in the table. In these patterns, strings written in *<bracketed italics>* represent a value. For example, *<package.qualified.interface>* might be something like com.mycompany.AccountService and *<component-id>* might be something like AccountApp/module1.jar/ServiceBean.

Description	Binding pattern
Short form local interfaces and homes	ejblocal:*<package.qualified.interface>*
Short form remote interfaces and homes	*<package.qualified.interface>*
Long form local interfaces and homes	ejblocal:*<component-id>*#*<package.qualified.interface>*
Long form remote interfaces and homes	ejb/*<component-id>*#*<package.qualified.interface>*

The *component-id* defaults to *<application-name>/<module-jar-name>* /*<ejb-name>* unless it is overridden in the EJB module binding file using the component-id attribute as described in the next section, ″Conflicts in short default binding names when multiple enterprise beans implement the same interface.″

Conflicts in short default binding names when multiple enterprise beans implement the same interface

When more than one enterprise bean that is running in the application server implements a given interface, the short default binding name becomes ambiguous because the short name might refer to any of the Enterprise JavaBeans that implement this interface. In order to avoid this situation, you must either explicitly define a binding for each Enterprise JavaBeans that implements the given interface as described in the next section, or disable short default bindings for applications containing these Enterprise JavaBeans by defining a WebSphere product ″JVM custom property″,

com.ibm.websphere.ejbcontainer.disableShortDefaultBindings. For more information about defining the JVM custom property, see the topic ″Java Virtual machine custom properties.″

To use this JVM custom property, set the property name to com.ibm.websphere.ejbcontainer.disableShortFormBinding and the property value to either * (asterisk} as a wildcard value to disable short form default bindings for all applications in the server, or to a colon-delimited sequence of the Java EE application names for which you want to disable short default bindings, for example, PayablesApp:InventoryApp:AccountServicesApp.

Effect of explicit assignment on default bindings

If you explicitly assign a binding definition for an interface or home, no short or long default bindings are performed for that interface.

Note: This only applies to the specific interfaces for which you assign an explicit binding. Other instances on that enterprise bean, without explicitly-assigned bindings, are bound by using default binding names.

User-defined bindings for EJB business interfaces and homes

For cases where you want to manually assign binding locations rather than using the product default bindings, you can use the EJB module binding file to assign your own binding locations to specific interfaces and homes. You can also use this file to only override the component ID portion of the default bindings on one or more enterprise beans in the module. Overriding the component ID provides a middle ground between allowing the bindings to completely default, versus completely specifying the binding name for each interface.

To specify user-defined bindings information for EJB 3.0 modules, place a file named, ibm-ejb-jar-bnd.xml, in the EJB JAR module's META-INF directory.

Note: The suffix on this file is XM**L**, not XM**I**, as in prior versions of product.
Also, when defining a binding for a local interface, you must preface the name with the string ″ejblocal:″ so it is bound into the JVM-scoped ejblocal: namespace.

The ibm-ejb-jar-bnd.xml file is used for EJB 3.0 modules that run on the product, whereas the ibm-ejb-jar.bnd.xmi file is used for pre-EJB 3.0 modules and for Web modules. The binding file format in ibm-ejb-jar.bnd.xml is different from the XMI file format for the following reasons:

- Bindings and extensions that are declared in the XMI file format depend on the presence of a corresponding ejb-jar.xml deployment descriptor file that explicitly refers to unique ID numbers that are attached to elements in that file. This system is no longer viable for EJB 3.0 modules, where it is no longer a requirement for the module to contain an ejb-jar.xml deployment descriptor.
- The XMI file format was designed to be machine-edited only by the product development tools and system management functions; it was effectively part of the product's internal implementation and the file's structure was never documented externally. This made it impossible for developers to manually edit binding files, or create them as part of a WebSphere-independent build process, in a supported manner.
- Rather than referring to encoded ID numbers in the ejb-jar.xml deployment descriptor, the XML-based binding file refers to EJB components by its EJB name. Each EJB component in a module is guaranteed to have a unique EJB name, either by default or through explicit assignment by the developer, so this provides an unambiguous way to target bindings and extensions.
- The new binding files are XML-based, and an XML Schema Definition (xsd) file is provided to externally document the structure. These .xsd files can be consumed by many common XML file editors to assist in syntactic verification and code completion functions. As a result, it is now possible for developers to produce and edit the binding and extension files independently of the Application Server infrastructure.

The following table lists the ibm-ejb-jar-bnd.xml elements and attributes that are used to assign bindings to EJB interfaces and homes for EJB 3.0 modules in WebSphere Application Server V7.

Element or attribute	How used	Example	Comments
<session>	Declares a group of binding assignments for a session bean.	`<session name= "AccountServiceBean"/>`	Requires name attribute and at least one of the following: simple-binding-name attribute, local-home-binding-name attribute, remote-home-binding-name attribute, or <interface> element.
name	Attribute that identifies the ejb-name of the enterprise bean that a <session>, <message-driven>, or <entity>, or other element applies to.	`<session name= "AccountServiceBean"/>`	The ejb-name value is the name declared in the <ejb-name> element of an ejb-jar.xml deployment descriptor file, the name parameter of a @Session or @MessageDriven annotation, or defaults to the unqualified class name of the EJB implementation class annotated with the @Session or @MessageDriven annotation (if no <ejb-name> value is declared in the XML deployment descriptor and no name parameter is declared on the annotation).

Element or attribute	How used	Example	Comments

component-id	Attribute that overrides the default component ID value for an enterprise bean. The default long-form bindings for this enterprise bean uses the specified component ID instead of *<app_name>/ <module_jar_name>/ <bean_name>*.	`<session name= "AccountServiceBean" component-id= "Dept549/ AccountProcessor"/>`
		
		The above example results in the bean whose ejb-name is AccountServiceBean, having its long-form default local interfaces bound at
		`ejblocal:Department549 /AccountProcessor# <package.qualified. interface>`
		
		Its long-form default remote interfaces are bound at
		`ejblocal:Department549 /AccountProcessor# <package.qualified. interface>`
			Can be used alone, or in combination with the <interface> element, the local-home-binding-name attribute, or the remote-home-binding-name attribute. Interfaces that are not assigned explicit bindings will have default bindings performed using the user-specified component ID value. Interfaces that are assigned explicit bindings are bound using those values.
			
			Since the simple-binding-name attribute is intended to apply to all defined interfaces on a given enterprise bean (leaving no interfaces defaulted), applying a component-id in combination with simple-binding-name is typically not useful.

Element or attribute	How used	Example	Comments

simple-binding-name	A simple mechanism for assigning interface bindings for Enterprise JavaBeans that:
	• Implement a single EJB 3.0 business interface, or
	• Implement a pre-EJB 3.0 style component interface (local, remote or both types) with a companion EJB home.
	The value of the attribute is used as the binding location of the enterprise bean's business interface, or the binding location of the Enterprise JavaBeans local and/or remote homes. The binding is placed in the ejblocal: namespace if the interface or home is local, and placed in the application server's root context of the globally-scoped JNDI namespace if the interface or home is remote.
		`<session name= "AccountServiceBean" simple-binding-name ="ejb/AccountService"/>`
		This example results in the bean whose ejb-name is AccountServiceBean, having its local business interface or home, if any, bound at
		`ejblocal:ejb/ AccountService`
		in the local JVM-scoped EJB namespace, and its remote business interface or home (if any) bound at
		`ejb/AccountService`
		in the application server's root context of the globally-scoped JNDI namespace. It is important to note here that the exact value of the attribute, including, in this specific example, the ″ejb″ subcontext name is used even if the interface is a local interface bound into the ejblocal: namespace. When user-defined bindings are specified, the exact name specified by the attribute is used.)
			Not to be used in combination with local-home-binding-name or remote-home-binding-name attributes, or the <interface> element. Also, should not be used on beans that implement more than one business interface (use the <interface> element in that case instead).
			If this attribute is used on an enterprise bean that implements more than one business interface, or a combination of business interface and local/remote component interface with home, the resulting bindings are disambiguated by appending a hash or number sign (# symbol) to the attribute value, followed by the package-qualified class name of each interface and/or home on the enterprise bean. This condition can be avoided, however, by using the <interface> element to define a binding for each of the business interfaces instead of using simple-binding-name.
			Important: defining a simple-binding-name on a bean that implements more than one business interface is not the same as overriding the default component ID for a bean using <component-id>. Remote interface default bindings defined with a component-id are still grouped under the ejb context (as all remote interface default bindings are), while remote interface bindings disambiguated by the EJB container in response to erroneous use of simple-binding-name on a bean with multiple interfaces are not grouped under the ejb context.Additionally, the inclusion of the package-qualified class name always occurs for long-form default bindings, whereas with

Element or attribute	How used	Example	Comments
local-home-binding-name	Attribute to specify the binding location of an enterprise bean's local home.	`<session name= "AccountServiceBean" local-home-binding -name="ejblocal: AccountService"/>`	Not to be used in combination with the simple-binding-name attribute. Since local homes must always be bound into the JVM-scoped namespace, the value must begin with the ejblocal: prefix.
remote-home-binding-name	Attribute to specify the binding location of an enterprise bean's remote home.	`<session name= "AccountServiceBean" remote-home-binding-name= "ejb/services/ AccountService"/>`	Not to be used in combination with the simple-binding-name attribute. The value cannot begin with the ejblocal: prefix, since remote homes cannot be bound into the ejblocal: namespace.
<interface>	A sub-element of the <session> element that assigns a binding to a specific EJB business interface. In contrast to the simple-binding-name, local-home-binding-name and remote-home-binding-name attributes, both a binding-name parameter and a class parameter are necessary (in fact, this distinction is why a separate XML element is necessary rather than an attribute). The class parameter specifies the package-qualified name of the business interface class to be bound.	`<interface class= "com.ejbs.Inventory Service" binding-name= "ejb/Inventory"/>` (declared as a sub-element inside a <session> element)	Not to be used in combination with the simple-binding-name attribute. Since local interfaces must always be bound into the JVM-scoped namespace, the binding-name value must begin with the ejblocal: prefix when this element is applied to a local interface.
binding-name	Attribute to specify the binding location of a business interface bound with the <interface> element.	`<interface class= "com.ejbs.Inventory Service" binding-name ="ejb/Inventory"/>` (declared as a sub-element inside a <session> element)	Required in combination with the <interface> element (and used on that element only). Since local interfaces must always be bound into the JVM-scoped namespace, the binding-name value must begin with the ejblocal: prefix when applied to a local interface.

Binding file Example 1

The following is a basic ibm-ejb-jar-bnd.xml file containing only the elements and attributes that assign binding names to EJB interfaces. It overrides the component ID used for default bindings on the enterprise bean that is named ″S01″, and assigns explicit bindings to some of the interfaces on the enterprise beans, ″S02″ and ″S03″, in this module.

```
<?xml version="1.0" encoding="UTF-8"?> <ejb-jar-bnd
  xmlns="http://websphere.ibm.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee   http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
 version "1.0">      <session name="S01" component-id="Department549/AccountProcessors"/>      <session name="S02"
 simple-binding-name="ejb/session/S02"/>      <session name="S03">     <interface class="com.ejbs.BankAccountService"
 binding-name="ejblocal:session/BAS"/>      </session>  </ejb-jar-bnd>
```

The binding file results in the following:

1. The session bean with ejb-name ″S01″ is assigned a user-defined component ID, overriding the default component ID (application name/ejb-jar module name/bean name) for all interfaces on that bean. Local interfaces on this bean are bound at

   `ejblocal:Department549/AccountProcessors#<package.qualified.interface.name>`

   while remote interfaces are bound at

   `ejb/Department549/AccountProcessors#<package.qualified.interface.name>`

   .
2. The session bean with ejb-name ″S02″ is assumed to have a single EJB 3.0 business interface. Alternatively, it could have a pre-EJB 3.0 ″component″ interface with local home, remote home, or both local and remote homes. The business interface, or component interface's home(s) are bound at

   `ejblocal:ejb/session/S02`

   if it is local, or

   `ejb/session/S02`

   if it is remote.

   If bean S02 has more than one business interface, or business interface(s) and home, a simple-binding-name is ambiguous. In that case, the container disambiguates the binding assignments by appending #<*package.qualified.interface.name*> to the simple binding name ejb/session/S02 for each of the bean's interfaces.
3. The EJB 3.0 business interface com.ejbs.BankAccountService on the session bean with ejb-name ″S03″ is bound at ejblocal:session/BAS.

All other business interfaces and homes on this bean, if present, are assigned default bindings. The com.ejbs.BankAccountService interface is assumed to be local since it was designated for the ejblocal: namespace in this example; an error would occur if the interface were not local.

The next section expands on this example, introducing elements for resolving the targets of various kinds of reference and injection entries that are declared either in the XML deployment descriptor or through annotations.

User-defined bindings for resolving references and injection targets

The previous section showed you how to assign user-defined binding names for business interfaces and homes. This section covers how to resolve linkage targets for references, injection directives and message-driven bean destinations.

Element or attribute	How used	Example	Comments
<jca-adapter>	Defines the JCA 1.5 adapter activation spec, and a message-destination JNDI location, for delivery of messages to a message-driven bean.	`<jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/>`	Requires *activation-spec-binding-name* attribute. If the corresponding message-drive bean does not identify its message destination by using the <message-destination-link> element, then the *destination-binding-name* attribute is also required. Can optionally include activation-spec-auth-alias attribute.
<ejb-ref>	Resolves the target of an ejb-ref declaration, which is declared through the @EJB annotation or through the ejb-ref in the ejb-jar.xml deployment descriptor, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target enterprise bean in the JVM-scoped ejblocal:, or globally-scoped JNDI namespace.	`<ejb-ref name="com.ejbs.BankAccountServiceBean/s02Ref" binding-name="ejb/session/S02"/>`	Requires the name and binding-name attributes.
<message-driven>	Declares a group of binding assignments for a message-driven bean.	`<message-driven name="EventRecorderBean"> <jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/> </message-driven>`	Requires name attribute and <jca-adapter> sub-element.

Element or attribute	How used	Example	Comments
<message-destination>	Associates the name of a message destination, which is a logical name defined in a Java EE module deployment descriptor, with a specific global JNDI name, which is an actual name in the JNDI namespace. <message-destination-ref> elements in the Java EE module deployment descriptor, or @Resource injection directives that inject message destinations, can then use the <message-destination-line> element to refer to this message-destination by the destination logical name, rather than requiring individual <message-destination-ref> binding entries in the binding file for each defined message-destination-ref.	`<message-destination name="EventProcessing Destination" binding-name="jms/ ServiceQueue"/>`	Requires name and binding-name attributes.
<message-destination-ref>	Resolves the target of a message-destination-ref declaration that is declared through the @Resource annotation or through the message-destination-ref in ejb-jar.xml, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target resource environment in the global JNDI namespace.	`<message-destination-ref name="com.ejbs. BankAccountServiceBean/ serviceQueue" binding-name= "jms/ServiceQueue"/>`	Requires the name and binding-name attributes.
<resource-ref>	Resolves the target of a resource-ref declaration that is declared through the @Resource annotation or through resource-ref in ejb-jar.xml, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target resource in the global JNDI namespace.	`<resource-ref name= "com.ejbs. BankAccountServiceBean/ dataSource" binding-name= "jdbc/Default"/>`	Requires the name and binding-name attributes. Can include the authentication-alias or custom-login-configuration attributes.

Element or attribute	How used	Example	Comments
<resource-env-ref>	Resolves the target of a resource-env-ref declaration that is declared through the @Resource annotation or through resource-env-ref in ejb-jar.xml, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target resource environment in the global JNDI namespace.	<resource-env-ref name="com.ejbs. BankAccountServiceBean/ dataFactory" binding-name="jdbc/ Default"/>	Requires the name and binding-name attributes.
name	Attribute that identifies the naming location, typically within the component-specific java:comp/env namespace, that defines the ″source″ side of a reference/target linkage, such as in ejb-ref, resource-ref, resource-env-ref, message-destination, or message-destination-ref.	<ejb-ref name= "com.ejbs. BankAccountServiceBean/ goodBye" binding-name= "ejb/session/S02"/>	
binding-name	Attribute that identifies the naming location within the ejblocal: or globally-scoped JNDI namespace that defines the ″target″ side of a reference/target linkage, such as in ejb-ref, resource-ref, resource-env-ref, message-destination, or message-destination-ref.	<ejb-ref name= "com.ejbs. BankAccountServiceBean/ goodBye" binding-name= "ejb/session/S02"/>	
activation-spec-binding-name	Attribute that identifies the JNDI location of the activation specification associated with the JCA 1.5 adapter to be used to deliver messages to a message-driven bean.	<jca-adapter activation-spec-binding-name=″jms/ InternalProviderSpec″ destination-binding-name=″jms/ ServiceQueue″/>	This name must match the name of a JCA 1.5 activation specification that you define to WebSphere Application Server.
activation-spec-auth-alias	Optional attribute that identifies the name of a J2C authentication alias used for authentication of connections to the JCA resource adapter. A J2C authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.	<jca-adapter activation-spec-binding-name=″jms/ InternalProviderSpec″ activation-spec-auth-alias=″jms/Service47Alias″ destination-binding-name=″jms/ ServiceQueue″/>	This name must match the name of a J2C authorization alias that you define to WebSphere Application Server.

Element or attribute	How used	Example	Comments
destination-binding-name	Attribute that identifies the JNDI name that the message-driven bean uses to look up its JMS destination in the JNDI name space.	<jca-adapter activation-spec-binding-name=″jms/InternalProviderSpec″ destination-binding-name=″jms/ServiceQueue″/>	This name must match the name of a JMS queue or topic that you define to WebSphere Application Server.
authentication-alias	Optional sub-element of the <resource-ref> binding element. If the resource reference is for a connection factory, then an optional JAAS login configuration can be specified; in this case a simple authentication alias name.	<resource-ref name="com.ejbs.BankAccountServiceBean/dataSource" binding-name="jdbc/Default"> <authentication-alias name="defaultAuth"/> <resource-ref>	This name must match the name of a JAAS authentication alias that you define to WebSphere Application Server.
custom-login-configuration	Optional sub-element of the <resource-ref> binding element. If the resource reference is for a connection factory, then an optional JAAS login configuration can be specified; in this case a set of properties (name/value pairs).	<resource-ref name="com.ejbs.BankAccountServiceBean/dataSource" binding-name="jdbc/Default"> <custom-login-configuration-name="customLogin"> <property name="loginParm1" value="ABC123"/> <property name="loginParm2" value="DEF456"/> </custom-login-configuration> </resource-ref>	This name must match the name of a JAAS login configuration that you define to WebSphere Application Server.

Binding file Example 2

Shown below is an expansion of the basic ibm-ejb-jar-bnd.xml file introduced in Example 1.

```
<?xml version="1.0" encoding="UTF-8"?> <ejb-jar-bnd
  xmlns="http://websphere.ibm.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee   http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
 version "1.0">    <session name="S01" component-id="Department549/AccountProcessors"/>    <session name="S02"
 simple-binding-name="ejb/session/S02"/>    <session name="S03">   <interface class="com.ejbs.BankAccountService"
 binding-name="ejblocal:session/BAS"/>   <ejb-ref name="com.ejbs.BankAccountServiceBean/goodBye"
 binding-name="ejb/session/S02"/>   <resource-ref name="com.ejbs.BankAccountServiceBean/dataSource"
 binding-name="jdbc/Default"/>  </session>    <message-driven-name="MO1">     <jca-adapter
 activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/>   </message-driven>
  <session name="S04" simple-binding-name="ejb/session/S04">   <resource-ref name="ejbs.S04Bean/dataSource"
 binding-name="jdbc/Default">   <authentication-alias name="defaultlogin"/>   </resource-ref>  </session>   <session
 name="S05">   <interface class="com.ejbs.InventoryService" binding-name="ejb/session/S05Inventory"/>   <resource-ref
 name="ejbs.S05Bean/dataSource" binding-name="jdbc/Default">    <custom-login-configuration name="customLogin">    <property
 name="loginParm1" value="ABC123"/>    <property name="loginParm2" value="DEF456"/>      </custom-login-configuration>
   </resource-ref>  </session> </ejb-jar-bnd>
```

This binding results in the following:

1. The business interface and home bindings for the session beans named S01, S02 and S03 are unchanged from the previous example.
2. The session bean whose ejb-name is ″S03″ now includes two reference target resolution bindings:
   - The ejb-ref binding resolves the EJB reference defined at java:comp/env/com.ejbs.BankAccountServiceBean/goodBye, to the JNDI location ejb/session/S02 within the

application server's root JNDI context. The EJB reference could also have been defined by an @EJB injection in the class com.ejbs.BankAccountServiceBean, into an instance variable named ″goodBye.″

Note: ejb/session/S02 is the JNDI location of session bean ″S02″ also defined in this same binding file, which means that the reference points to the session bean whose name is ″S02.″

- The resource-ref binding resolves the resource reference defined at java:comp/env/com.ejbs.BankAccountServiceBean/dataSource, to the JNDI location jdbc/Default. The resource reference could also have been defined by a @Resource injection in the class com.ejbs.BankAccountServiceBean, into an instance variable named ″dataSource.″

3. Bindings are defined for a message-driven bean whose ejb-name is ″M01″. The MDB receives messages from a JMS destination defined to WebSphere Application Server, whose JNDI name is jms/ServiceQueue, using a JCA 1.5 adapter whose JCA 1.5 activation spec has been defined to WebSphere Application Server with the name jms/InternalProviderSpec.
4. The session bean whose ejb-name is ″S04″ is assumed to have a single business interface, which is bound at ejb/session/S04 if remote, or ejblocal:ejb/session/S04 if local. It has a resource-ref with name, java:comp/env/ejbs/S04Bean/dataSource. This can also be the class, ejbs.S04Bean, with an @Resource injection into a variable named, dataSource. This resource-ref resolved to the JNDI location jdbc/Default. The resource-ref refers to a J2C connection and connects to this resource using a simple authentication alias named ″defaultlogin″ that has been defined to WebSphere Application Server.
5. A business interface binding is defined for the interface whose class name is com.ejbs.InventoryService implemented by the session bean whose ejb-name is ″S05″; the interface is assumed to be remote since it is not prefixed with ″ejblocal:″ and will thus be bound at ejb/session/S05Inventory in the server's root JNDI context in the globally-scoped namespace. Any other business interfaces implemented by this bean is assigned default bindings. The bean has a resource-ref with name java:comp/env/ejbs.S05Bean/dataSource (or a @Resource injection in the class ejbs.S05Bean into a variable named ″dataSource″) that is resolved to the JNDI location jdbc/Default. The resource-ref refers to a J2C connection and will connect to this resource using a custom login configuration that includes two name-value pairs.

Bindings file Example 3

This example demonstrates how to define and resolve EJB reference bindings to perform JNDI lookups across application server instances within the same WebSphere Application Server cell. It uses two EJB beans: a called bean that defines an explicit binding using the simple-binding-name attribute, and a calling bean that performs an @EJB injection and uses the ejb-ref element within its associated binding file to resolve the reference so it points at the called bean, which resides in a different application server process.

ibm-ejb-jar-bnd.xml (called bean)

```
<?xml version="1.0" encoding="UTF-8"?> <ejb-jar-bnd
 xmlns="http://websphere.ibm.com/xml/ns/javaee"    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-ej    b-jar-bnd_1_0.xsd"
 version="1.0">        <session name="FacadeBean" simple-binding-name="ejb/session/FacadeBean"/>  </ejb-jar-bnd>
```

This binding file content assumes that the session bean whose ejb-name is ″FacadeBean″ implements a single business interface (and thus the simple-binding-name attribute can be used, as an alternative to the <interface> sub-element). In this case, the FacadeBean implements a single remote business interface, bound at ejb/session/FacadeBean in the server root JNDI context of the application server where the FacadeBean resides.

Code snippet (calling bean)

```
@EJB(name="ejb/FacadeRemoteRef")  FacadeRemote remoteRef;  try {   output =
 remoteRef.orderStatus(input);           } catch (Exception e) {   // Handle exception, etc.  }
```

This code snippet performs an EJB resource injection into the instance variable named ″remoteRef″, which is of type FacadeRemote. The injection overrides the ″name″ parameter, setting the resulting ejb-ref reference name to ejb/FacadeRemoteRef. The code invokes a business method on the injected reference.

ibm-ejb-jar-bnd.xml (calling bean)

```
<?xml version="1.0" encoding="UTF-8"?> <ejb-jar-bnd
 xmlns="http://websphere.ibm.com/xml/ns/javaee"    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
 version="1.0">    <session name="CallingBean">     <ejb-ref name="ejb/FacadeRemoteRef"
 binding-name="cell/nodes/S35NLA1/servers/S35serverA1/ejb/session/FacadeBean"/>   </session> </ejb-bnd-jar>
```

Finally, this binding file resolves the EJB reference with an ejb-ref name of ejb/FacadeRemoteRef to point to the globally-scoped JNDI name of cell/nodes/S35NLA1/servers/S35serverA1/ejb/session/FacadeBean. This globally-scoped JNDI name represents an interface bound at ejb/session/FacadeBean under the server root context of the server named ″S35serverA1″ on the node named ″S35NLA1″ within the WebSphere Application Server cell of the calling bean. To point to a location within a different WebSphere Application Server cell, a CORBAName-style name can be used instead of a standard JNDI name.

Instructions on how to modify the ibm-ejb-jar-bnd.xml file can be found in the topic, Ways to update application files.

The relationship between injections and references

There is a one-to-one correspondence between injection directives and reference declarations - every injection implicitly defines a reference of some type, and conversely, every reference can optionally also define an injection. You can think of an injection annotation as being the mechanism to define references through annotations rather than defining them in the XML deployment descriptor.

By default, an injection defines a reference with a name formed from the package-qualified class name of the component performing the injection, a forward slash (/), then the name of the variable or property being injected into. For example, an injection performed in the class com.ejbs.AccountService, into a variable or property named ″depositService″, results in a reference named java:comp/env/com.ejbs.AccountService/depositService. However, specifying the optional ″name″ parameter on the injection directive overrides this default name and causes the reference to be named according to the value of the ″name″ parameter.

Knowing this rule, it is easy to see how a bindings file can be used not only to resolve targets for references declared in an XML deployment descriptor, but also to resolve targets for references implicitly declared by an annotation injection directive. Simply use the value of the ″name″ parameter on the injection annotation, or the default reference name from the class name and variable/property name if no ″name″ parameter is specified, just as if it were the name of the reference declared in an XML deployment descriptor.

Default resolution of EJB references and EJB injections: The AutoLink feature

AutoLink is a value-add feature of WebSphere Application Server that eliminates the need to explicitly resolve EJB reference targets in certain usage scenarios. In WebSphere Application Server V7, AutoLink is implemented within the boundaries of each WebSphere Application Server process. The AutoLink algorithm works as follows.

When the WebSphere EJB container encounters an EJB reference within a given EJB module, it first checks to see if you have explicitly resolved the target of that reference through inclusion of an entry in the module's binding file. If it finds no explicit resolution of the target in the binding file, the container searches within the referring module for an enterprise bean that implements the interface type you have defined within the reference. If it finds **exactly one** enterprise bean within the module that implements the interface, it uses that enterprise bean as the target for the EJB reference. If the container cannot locate an enterprise bean of that type within the module, it expands the search scope to the application that the

module is part of, and search other modules within that application that are assigned to the same application server as the referring module. Again, if the container finds **exactly one** enterprise bean that implements the target interface, within the application's other modules assigned to the same server as the referring module, it uses that enterprise bean as the reference target.

The scope of AutoLink is limited to the application in which the EJB reference appears, and to the application server on which the referring module is assigned. References to enterprise beans in a different application, enterprise beans in a module assigned to a different application server, or to enterprise beans residing in a module that has been assigned to a WebSphere Application Server cluster, must be explicitly resolved using reference target bindings in the EJB module's `ibm-ejb-jar-bnd.xml` file, or the Web module's `ibm-web-bnd.xmi` file.

It is important to note that AutoLink is only supported for EJB references, not other types of references although it is supported from the EJB container, the Web container, and the application client container. Also, because the scope of the AutoLink function is limited to the server that the referring module is assigned to, or in the case of the Java EE client container, to the server that the client container is configured as its JNDI bootstrap server, it is useful mainly in development environments and other single-server usage scenarios. Even with these present limitations, it can be a significant value during the development experience by removing the need to explicitly resolve EJB references.

Naming considerations in clustered and cross-server environments

The global JNDI naming conventions in the previous sections apply in non-clustered environments and when the lookup target is within the same cluster as the source of the lookup. When a lookup is performed from outside a cluster on a binding that is within a given cluster, the lookup string must be qualified to indicate the name of the cluster in which the target resides, according to the following convention:

```
cell/clusters/<cluster-name>/<name-binding-location>
```

For example, given an EJB interface binding location within the application server root context:

```
ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

If the EJB implementing this interface is assigned to an application server that is a member of a cluster named Cluster47, the lookup string external to that cluster is as follows:

```
cell/clusters/Cluster47/ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

When a lookup is performed across application server processes, the lookup string must be qualified to indicate the name of the node and server in which the target resides, according to the following convention:

```
cell/nodes/<node-name>/servers/<server-name>/<name binding location>
```

Again, given an EJB interface binding location within the application server root context:

```
ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

If the enterprise bean that is implementing this interface is assigned to an application server named Server47A1 that is located on a node named S47NLA1, the cross-server lookup string is as follows:

```
cell/nodes/S47NLA1/servers/Server47A1/ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

User-defined EJB extension settings

For cases where you wish to specify values for WebSphere Application Server EJB Extension settings, you can use an *EJB module extension file* to assign these settings to specific EJB types within that module. You specify extension settings information for EJB 3.0 modules by placing one, or both, of two files into the EJB JAR module's META-INF directory, depending on the type of extension being defined. The names of the two files are `ibm-ejb-jar-ext.xml` and `ibm-ejb-jar-ext-pme.xml`.

Note: The suffix on these files are XM**L**, not XM**I** as in prior versions of WebSphere Application Server.

The ibm-ejb-jar-ext.xml and ibm-ejb-jar-ext-pme.xml files are used for **EJB 3.0** modules running in WebSphere Application Server, whereas the ibm-ejb-jar-ext.xmi and ibm-ejb-jar-ext-pme.xmi files are used for **pre-3.0 EJB** modules. WebSphere Application Server Version 7.0 uses a new XML-based extension file format instead of the previous xmi file format for the following reasons:

1. Bindings and extensions declared in the xmi file format depend on the presence of a corresponding ejb-jar.xml deployment descriptor file, explicitly referring to unique ID numbers attached to elements in that file. This system is no longer viable for EJB 3.0 and later modules, where it is no longer a requirement for the module to contain an ejb-jar.xml deployment descriptor.
2. The xmi file format was designed to be machine-edited only by WebSphere development tools and system management functions; it was effectively part of WebSphere's internal implementation and the file's structure was never documented externally. This made it impossible for developers to manually create or edit binding or extension files, or create them as part of a WebSphere-independent build process, in a supported manner.
3. Rather than referring to encoded ID numbers in ejb-jar.xml, the XML-based extension file format refers to EJB components by their EJB name. Each EJB component in a module is guaranteed to have a unique EJB name, either by default or through explicit assignment by the developer, so this provides an unambiguous way to target bindings and extensions.
4. The new binding and extension file formats are XML-based, and XML Schema Definition (xsd) files are provided to externally document their structure. These .xsd files may be consumed by many common XML file editors to assist in syntactic verification and code completion functions. As a result, it is now possible for developers to produce and edit these binding and extension files independently of WebSphere Application Server infrastructure, using a generic XML editor or scripting system of their choice.

Extensions defined in META-INF/ibm-ejb-jar-ext.xml

The following tables list extension elements and attributes that must be placed in the META-INF/ibm-ejb-jar-ext.xml file. The next section lists elements and attributes that appear in a separate file, META-INF/ibm-ejb-jar-ext-pme.xml.

Element or Attribute	How Used	Example	Remarks
<session>	Declares a group of extension settings for a session bean.	`<session name= "AccountServiceBean"/>`	Requires *name* attribute. In order to have any effect, also include at least one extension setting definition sub-element.
<message-driven>	Declares a group of extension settings for a message-driven bean.	`<message-driven name= "EventProcessorBean"/>`	Requires *name* attribute. In order to have any effect, also include at least one extension setting definition sub-element.

Element or Attribute	How Used	Example	Attribute Default	Remarks
<time-out>	Sub-element to the <session> element that optionally declares the number of seconds between method invocations after which a **stateful** session bean will be no longer available.	`<session name= "ShoppingCartBean"> <time-out value= "600"/></session>`	300 (10 minutes)	Requires *value* attribute, a positive integer. **Only applicable to stateful session beans; must not be used on stateless beans.**

Element or Attribute	How Used	Example	Attribute Default	Remarks
<bean-cache>	Sub-element of <session> element used to declare bean activation/passivation settings for stateful session beans.	`<session name= "ShoppingCartBean> <bean-cache activation-policy= "TRANSACTION"/> </session>`		In order to have any effect, should also include the *activation-policy* attribute.

Element or Attribute	How Used	Example	Attribute Default	Remarks
activation-policy	Attribute of <bean-cache> element that declares the conditions under which the bean instance will be activated and passivated. Applicable to stateful session beans. Allowable values and their meanings are: • **TRANSACTION:** Indicates that the bean activates at the start of a transaction and passivates (and is removed from the active EJB instance cache) at the end of the transaction. • **ONCE:** Indicates that the bean activates when it is first accessed in the server process, and passivates (and is removed from the active EJB instance cache) at the discretion of the container, for example, when the cache becomes full. • **ACTIVITY_SESSION:** Indicates that the bean activates and passivates as follows: 1) On an ActivitySession boundary, if an ActivitySession context is present on activation, 2) On a transaction boundary, if a transaction context (but no ActivitySession context) is present on activation, or otherwise, 3) on an invocation boundary.	`<session name= "ShoppingCartBean> <bean-cache activation-policy= "ONCE"/></session>`	ONCE for stateful session beans.	

Element or Attribute	How Used	Example	Attribute Default	Remarks
<global-transaction>	Sub-element to the <session> and <message-driven> elements that can be used to declare the transaction timeout (in seconds) to be used on transactions started by this specific EJB type (overriding the server setting for global transaction timeout) and also may declare whether this EJB type will propagate global transaction context received through web service atomic transactions, across the heterogeneous Web service environment.	`<session name= "AccountServiceBean" <global-transaction transaction-timeout= "180" send-wsat-context="FALSE"/> </session>`	Server transaction timeout setting for transaction-timeout; FALSE for send-wsat-context	Requires at least one of *transaction-timeout* or *send-wsat-context* attributes.

Element or Attribute	How Used	Example	Attribute Default	Remarks
<local-transaction>	Sub-element to the <session> and <message-driven> elements that can be used to declare settings related to local transactions. Allowed attributes are *boundary*, *resolver*, and *unresolved-action*; these attributes configure, for the component, the behavior of the container's local transaction containment (LTC) environment that the container establishes whenever a global transaction is not present. The meaning of each attribute is as follows: **Boundary** This setting specifies the containment boundary at which all contained resource manager local transactions (RMLTs) must be completed. Possible values are: • **BEAN_METHOD:** This is the default value. If you select this option, RMLTs must be resolved within the same bean method in which they were started. • **ACTIVITY_SESSION:** RMLTs must be resolved within the scope of any ActivitySession in which they were started or, if no ActivitySession context is present, within the same bean method in which they were started. **Resolver** This setting specifies the component responsible for	`<session name>= "AccountServiceBean"> <local-transaction boundary= "BEAN_METHOD" resolver= "APPLICATION" unresolved-action= "ROLLBACK"/> </session>`	`boundary="BEAN_ METHOD"; resolver= "APPLICATION"; unresolved-action= "ROLLBACK"`	Requires at least one of *boundary*, *resolver*, or *unresolved-action* attributes.

Element or Attribute	How Used	Example	Attribute Default	Remarks

Element or Attribute	How Used	Example	Attribute Default	Remarks
<method>	Sub-element to the <method-session-attribute> and <run-as-mode> elements that is used to specify the method name, method signature, or method types to which a given setting will apply. Allowed attributes are *type*, *name*, and *params*. The meaning of each attribute is as follows: **type** • **UNSPECIFIED:** The setting will apply to all methods matching the *name* and/or *params* attributes, regardless of interface type. • **REMOTE:** The setting will apply to remote business interface and remote component interface methods matching the *name* and/or *params* attributes. • **LOCAL:** The setting will apply to remote business interface and remote component interface methods matching the *name* and/or *params* attributes. • **HOME:** The setting will apply to remote home interface methods matching the name and/or params attributes matching the name and/or *params* attributes. • **LOCAL_HOME:** The setting will apply to local home interface methods matching the *name* and/or params attributes. • **SERVICE_ENDPOINT**: The setting will apply to methods on the	<session /name= "AccountServiceBean"> <method-session-attribute type= "REQUIRES_NEW"> <method type= "LOCAL" name= "debitAccount" params="java.lang. String[], int, com.xyz. CustomerInfo"/> </method-session-attribute;> </session>		

Element or Attribute	How Used	Example	Attribute Default	Remarks
<run-as-mode>	Sub-element to the <session> and <message-driven> elements that can be used to declare the security identity that a given EJB method will have while the method is being executed. The identity can be set to use the identity of the caller (mode = CALLER_IDENTITY), the identity of the EJB server (mode = SERVER_IDENTITY), or the identity of a specific security role (mode = SPECIFIED_IDENTITY).	<session name= "AccountServiceBean"> <start-at-app- startvalue="TRUE"/> </session>		Requires *mode* attribute and <*method*> sub-element. If the mode is SPECIFIED_IDENTITY, the <specified-identity sub-element is also required.
<start-at-app-start>	Sub-element to the <session> and <message-driven> elements that can be used to inform the EJB container that specified EJB type shall be initialized at the time the application is first started, rather than the time the EJB type is first used by the application.	<session name= "AccountServiceBean"> <start-at-app- startvalue= "TRUE"/></sesssion>	FALSE (initialize EJB type when EJB is first used by application) for beans other than message-driven beans. Always TRUE for message-driven beans.	Requires *value* attribute

Element or Attribute	How Used	Example	Attribute Default	Remarks
<resource-ref>	Sub-element to the <session> and <message-driven> elements, that may be used to declare additional settings on a Java EE resource reference, such as isolation level to be used on transactions driven through the connection referred to by the reference. Allowable attributes include *isolation-level*. The meaning of each attribute is as follows: **isolation-level** • **TRANSACTION_REPEATABLE_READ:** This isolation level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads. • **TRANSACTION_READ_COMMITTED:** This isolation level prohibits dirty reads, but allows nonrepeatable reads and phantom reads. • **TRANSACTION_READ_UNCOMMITTED:** This isolation level allows reading uncommitted changes (data changed by a different transaction that is still in progress). It also allows dirty reads, nonrepeatable reads, and phantom reads. • **TRANSACTION_SERIALIZABLE:** This isolation level prohibits the following types of reads: 1) Dirty reads, in which a transaction reads a database row containing uncommitted changes from a second transaction, 2) Nonrepeatable reads, in which one transaction reads a row, a second	`<session name= "AccountServiceBean"> <resource-ref name="jdbc/Default" isolation-level= "TRANSACTION_NONE"> </session>`		Requires *name* attribute. In order to have any effect, must also include the isolation-level attribute.

Extensions defined in META-INF/ibm-ejb-jar-ext-pme.xml

The following tables list extension elements and attributes that must be placed in the META-INF/ibm-ejb-jar-ext-pme.xml file.

Element or Attribute	How Used	Example	Attribute Default	Remarks
<internationalization>	Element that may be used to declare the locale that will be used by the EJB type (caller's locale or server's locale).	<internationalization> <application> <ejb name="S01"/> <ejb name="S02"/> </application> <run-as-caller> <method type="LOCAL" name="getFoo" params="int"> <ejb name="C01"/> </method></run-as- caller><run-as- server><method type="LOCAL" name= "getBar" params= "int"><ejb name= "C02"/></method> </run-as-server> <run-as-specified name="North American English"><locale lang="en" country= "US" variant="foo"/> <locale lang="en" country="CA" variant= "bar" /> <time-zone name="GMT"/> <method type="LOCAL" name= "getFoo" params= "int"> <ejb name= "C03"/> </method></run-as- specified><run-as- specified name= "North American French"> <locale lang="fr" country= "US" variant="foo"/> <locale lang="fr" country="US" variant="bar" /> <time-zone name= "GMT" /> <method type="LOCAL" name= "getBar" params= "int"> <ejb name= "C04"/></method> </run-as-specified> </internationalization>		For information on this extension, see http://publib.boulder .ibm.com/infocenter/a diehelp/index.jsp?top ic=/com.ibm.etools.j2 ee.pme.ui.doc/concept s/cin_containerattrib ute.html . Due to the complexity of this function, you may wish to use tooling designed for WebSphere Application Server such as Rational Application Developer (RAD) to produce the desired extension file stanzas, then modify the XML file as desired.

Element or Attribute	How Used	Example	Attribute Default	Remarks
<activity-sessions>	Element that optionally declares the type of activity session management to be used on a designated session bean (BEAN or CONTAINER) and for container-managed activity sessions, the type of activity session behavior to be provided by the container.	<activity-sessions> <container-activity-session name="Foo" type="NOT_SUPPORTED"> <methodtype="HOME" name="findByPrimaryKey" params="int"><ejb name="C01"/> </method> </container-activity-session> <./activity-sessions>		For information on this extension, see http://publib.boulder.ibm.com/infocenter/adiehelp/index.jsp?topic=/com.ibm.etools.j2ee.pme.ui.doc/tasks/tas_depejb2.html . Due to the complexity of this function, you may wish to use tooling designed for WebSphere Application Server such as Rational Application Developer
<app-profiles>	Element that optionally declares application profile settings for one or more EJBs	<app-profiles> <defined-access-intent-policy name="foo"><collection-scope type="SESSION"/> <optimistic-read/> <read-ahead-hint hint="foo.bar.baz"/> </defined-access-intent-policy><run-as-task name="TestEJB1.ejbs.C01LocalHome.createjava.lang.Integer" type="RUN_AS_SPECIFIED_TASK"><task name="/> <method type="LOCAL" name"getFoo" params="int"><ejb name="C01"/> </method> </run-as-task> <ejb-component-extension ejb="C01"> <task name="SomeTask"/></ejb-component-extension> </app-profiles>		Due to the complexity of this function, you may wish to use tooling designed for WebSphere Application Server such as Rational Application Developer (RAD) to produce the desired extension file stanzas, then modify the XML file as desired.

Legacy (XMI) bindings

Existing modules and applications can continue to use the legacy binding support provided in the product, therefore, the existing tools and wizards can be used to specify binding and extension information for applications and modules. Use of the legacy support is limited to EAR files and modules using J2EE 1.4-style XML deployment descriptors.

EJB modules that use a version 3.0 XML deployment descriptor schema or do not have an XML deployment descriptor file must use either defaulted bindings and AutoLink, or user-specified XML binding files.

It is required that CMP entity beans always be packaged in a module with a 2.1 XML deployment descriptor schema version so that existing tools can be used to provide mappings, bindings, and extension support.

User-specified XML bindings

The default bindings for each interface and AutoLink reference resolution for each reference can be overridden by specifying bindings for the EJB module by creating a META-INF/ibm-ejb-jar-bnd.xml file.

The schema files that describe the format are located in the <WAS_HOME>/properties/schemas directory. This form of bindings specification can only be used for modules containing either no XML deployment descriptor or an EJB 3.0 deployment descriptor.

Note: It is not required to specify all bindings. Any binding name or reference that is not defined uses the default bindings and AutoLink support.

Bindings can be specified for the following:

- Session beans using the <session> element.
- Message Driven beans using the <message-driven> element

The only attributes and sub-elements supported for the <session> element are:

- id attribute
- name attribute
- simple-binding-name attribute
- component-id attribute
- ejb-ref element
- resource-ref element and its attributes
- resource-env-ref element and its attributes
- message-destination-ref element and its attributes

The only attributes and sub elements supported for the <message-driven> element are:

- id attribute
- name attribute
- jca-adapter attribute
- ejb-ref element and its attributes
- resource-ref element and its attributes
- resource-env-ref element and its attributes
- message-destination-ref element and its attributes

EJB 3.0 module packaging overview

This topic describes application packaging when you use Enterprise JavaBeans (EJB) 3.0 beans.

Packaging applications that use EJB 3.0 beans is similar to the assembly requirements for Java 2 Platform, Enterprise Edition (J2EE) 1.4 applications: components are packaged into modules, and modules are packaged into application enterprise archive (EAR) files. The components and modules both have describing metadata provided in an Extensible Markup Language (XML) deployment descriptor. The EJB 3.0 specification supports an additional method to describing metadata and for packaging persistence units.

The EAR file is a package file format similar to a .zip or .tar file format. The EAR file can be visualized as a collection of logical directories and files that are packaged together into a simple file. Each EAR file includes one or more Java Platform, Enterprise Edition (Java EE) module files, which can include the following:

- Java application archive (JAR) files for EJB modules. Java EE application client modules and utility class modules.
- Web application archive (WAR) files for Web modules.
- Other technology-specific modules such as resource application archive (RAR) files and other types of modules.

EJB modules without deployment descriptors

You can package EJB modules without a deployment descriptor if you are using EJB 3.0 beans. To do this, you must create a JAR file with metadata in an annotation which is located in the EJB component. EJB 3.0 beans do not need an entry in the ejb-jar.xml file for metadata that you have defined through annotations.

If you installed the Feature Pack for EJB 3.0, the default was to scan annotations during the installation of an EJB 3.0 module. For WebSphere Application Server, Version 7.0, the default is not to scan pre-Java EE 5 modules during the application install or at server startup

To preserve backward compatibility with both the Feature Pack for EJB 3.0 and the Feature Pack for Web Services, you have a choice whether or not to scan legacy Web modules for additional metadata. A server level switch is defined for each feature pack scan behavior. If the default is not appropriate, the switch must be set on each server and administrative server that requires a change in the default. The switches are server custom properties com.ibm.websphere.webservices.UseWSFEP61ScanPolicy={true|false} and com.ibm.websphere.ejb.UseEJB61FEPScanPolicy={true|false}. To define these properties in the administrative console click **Application servers** → **server name** → **Process definition** → **Java Virtual Machine** → **Custom properties**.

EJB modules with deployment descriptors

You can continue to use EJB modules with deployment descriptors. Modules with deployment descriptors can support any EJB specification version level, including EJB 3.0, but generally these descriptors should reflect the implementation requirements of the components in the module.

An EJB module can have an EJB 2.1-, or earlier, style deployment descriptor, or an EJB 3.0-style deployment descriptor.

For EJB 2.1-style deployment descriptors, it is assumed that the deployment descriptor contains the full metadata for the module, and no additional scanning of annotation metadata occurs.

The EJB container annotation scanning is performed on EJB modules that either have no deployment descriptor or have an ejb-jar.xml deployment descriptor at the EJB 3.0 schema level. In other words, the scan finds the annotation and its describing metadata.

Note: You cannot scan for component annotation metadata contained within shared libraries defined using the WebSphere Application Server system management shared library feature.

Persistence units

Persistence units, including the persistence.xml file and the classes associated with it, can be packaged in the module for which they are required. They can also be packaged in the separate utility JAR file that is packaged in the EAR file with its dependent module.

When a separate utility JAR file is packaged, it is necessary for the module that desires it to use the persistence units to declare a dependency on the utility JAR file using the typical MANIFEST. MF Class-Path: declarations. See the example scenario for this packaging method under the section in this topic called ″Session facades used for persistence scenario″

Note: Packaging of persistence units contained within shared libraries defined using the WebSphere Application Server system management shared library feature is not supported at this time.

Application packaging

Modules that utilize the EJB 3.0 specification should be packaged in an EAR file with a J2EE 1.4 deployment descriptor so that legacy application security management can be used.

You can mix EJB 2.x and earlier beans with EJB 3.0 beans in the same application, but you do need to separate EJB 2.x and earlier beans from EJB 3.0 beans so that they are not in the same modules. EJB 3.0 beans are not recognized in modules that contain EJB 2.1-style, or earlier, deployment descriptors.

In the case that the EAR file only contains the JAR and Web archive (WAR) files, and no application.xml file, the product provides a default J2EE 1.4 deployment descriptor that is based on the following defaults that are outlined in the Java EE specification:

- The application name is assumed to be the name of the EAR file, but with the EAR file extension removed.
- Files that are ending in .war are assumed to be Web modules. The context root of the Web module is the name of the file that is relative to the root of the application package, but with the WAR file extension removed.
- Files that are ending in .jar that are not in the /lib directory, and that contain either an ejb-jar.xml file or at least one class that defines a @Stateful or @Stateless annotation, are assumed to be EJB modules.
- Other JAR files that are not in the /lib directory are not assumed to be EJB modules.

If the application archive file contains an application.xml descriptor, processing occurs according to the directives in that descriptor.

AutoLink

AutoLink provides the ability to automatically resolve EJB references to components contained with an EAR file, without having to specify a JNDI binding name. This simplifies application deployment with large numbers of beans and references if they are unique and unambiguous.

Note: AutoLink should not be used for references to components deployed on a cluster.

JPA packaging

It is recommend that persistence units be packaged in separate JAR files to make them more accessible and reusable. These can be tested outside the container, with or without actual database persistence occurring. Persistence units can be included in standalone applications or into EAR files as utility JAR files. Because of the variety of use cases and potential performance issues when scanning large quantities of classes, it is recommended that the persistence unit defines the classes of the persistence units.

Session facades used for persistence scenario

A common pattern is to use session facades for persistence. Using session bean facades to drive JPA is supported. The EntityManager interface is not thread safe, therefore, servlets should never inject @PersistenceContext. Servlets must either use the facade pattern or use an EntityManagerFactory instance to create an EntityManager on each request.

It is recommended that JPA persistence units be defined in a separate JAR file, apart from the session bean facades. Not only is this a best practice that gives greater flexibility in sharing, it also avoids problems mixing JPA and non-JPA annotated classes.

Typically, a JAR file is created to hold the entity classes and the JPA persistence.xml definition and added to the EAR file as a utility JAR file. The EJB 3.0 module adds a dependency on the JAR file by declaring it in the EJB 3.0 module MANIFEST.MF. For example, if an EAR contains a TradeApp.ear, TradeWeb.war, EJB3Trade.jar, and TradeInfo.jar, the EJB3Trade.jar file would have a MANIFEST.MF that looks like the following:

```
Manifest-Version: 1.0
Class-Path: TradeInfo.jar
```

The session facade in the EJB3Trade.jar file refers to JPA entity classes and persistence units in the TradeInfo.jar file. The Web application defined in the TradeWeb.war file can do the same to work with the JPA entity objects as Data Transfer Objects flowing between the Web and EJB container tiers.

Cross-tier and cross version session bean reference scenario

There are several ways to define and use references to EJB 3.0 session beans. For EJB 3.0 session to session, the @EJB injection target can be used. For cross-tier, for example, Web application to EJB 3.0 session, or cross-version, for example, EJB 2.1 session to EJB 3.0 session, an XML deployment descriptor reference can be used to define ejb-refs and ejb-local-refs. There are two variations of these, depending on whether an EJB 3.0 business interface is referred to, or a pre-EJB 3.0 component-style interface that also defines an EJBLocalHome is referred to. Web applications and client applications can also utilize the @EJB annotation if the component being referenced can be resolved using autolink.

For migration scenarios where session beans are being converted from EJB 2.1 beans to EJB 3.0 beans, the pattern is typically to edit the Session bean class, replace the *implements SessionBean* with *implements the business interface*, remove *extends EJBLocalObject* from the local interface and non-business throws clauses, and add the @Stateful @Local @LocalHome(<localhome>.class) or similar annotations. Existing ejb-refs and ejb-local-refs are bound to the new implementation of the session bean.

Note: The default binding name does change.

The previous scenario uses an EJB 2.1-style client pattern with an EJB 3.0-style session bean implementation. For a more current client style, the client-side can be cleaned up to lookup the session bean business interface directly, rather than going through a home interface. In this case, it is not be necessary to define the @LocalHome(<localhome>.class) annotation. You can use a variant definition of ejb-ref and ejb-local-ref to do this. Use a *null* value for the local-home element value and bind the ejb-local-ref to the session bean's ejblocal: binding rather than the home binding. For example:

```
<ejb-local-ref id="EJBLocalRef_1154112538064">
 <description>com.ibm.persistence.ejb3.order.facadecom.ibm.persistence.ejb3.order.facade</description>
 <ejb-ref-name>ejb/OrderEJB</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home></local-home>
 <local>com.ibm.persistence.ejb3.order.facade.OrderProcessor</local>
</ejb-local-ref>
```

The client code also needs to be adjusted to do the appropriate casting for the object being looked up. In this case, the business interface instead of the home interface:

```
try {
 InitialContext ctx = new InitialContext();
 orderProcessor = (OrderProcessor)ctx.lookup("java:comp/env/ejb/OrderEJB");
}
catch(Exception e)  {
 e.printStackTrace(System.out);
 throw new ServletException(e);
}
```

EJB 3.0 deployment overview

Learn about the Enterprise JavaBeans (EJB) 3.0 deployment model, including to *Just-In-Time* (JIT) deployment.

All Java Platform, Enterprise Edition (Java EE) application server products have some form of EJB deployment phase where your application is customized to run in that particular application server implementation. Typically, this is accomplished by an application server-specific deployment tool, which generates code to bridge your EJB interface and implementation code to the application server's EJB container implementation. Some application server products' deploy tools alter the bytecodes of your application classes rather than using code generation, but the end result is similar.

In WebSphere Application Server, the bridging is accomplished by generating code that *wraps* your EJB implementation classes, connecting them to the product EJB container, which in turn allows the EJB container to host your enterprise beans and provide services to them. If one or more of your enterprise beans has defined remote interfaces, additional code is generated to provide the remote function.

Historically, EJB deployment in the WebSphere product has been performed by the *EJBDeploy* tool that is included with product and packaged with WebSphere product-oriented development tools.

The EJBDeploy tool introspects your EJB external interfaces, generates the wrapper code as .java files, then compiles it using the javac compiler to produce .class files, which are then packaged in your EJB module with your application code. The EJBDeploy tool also runs the rmic tool against the remote EJB interfaces in the application, producing additional *stub* and *tie* class files that interact with the Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) Object Request Broker (ORB), providing remote object support. Typically, you run the EJBDeploy tool either when you install the application on the product or sometime before you install the application from the command-line tool or within a development tool.

Just-In-Time deployment

The EJB 3.0 support in WebSphere Application Server adds a new feature called Just-In-Time Deployment. With Just-In-Time Deployment, the EJB container dynamically generates the wrapper, stub, and tie classes in-memory as needed when the application is running. Additionally, the Web container and application client containers dynamically generate the stub class required for remote EJB invocations. Effectively, this means that you do not need to process EJB 3.0 modules, Web modules that invoke EJB 3.0 beans, or client modules that invoke EJB 3.0 beans, through the EJBDeploy tool prior to running them in WebSphere.

createEJBStubs tool

Even though the Just-In-Time Deployment feature, in many cases, dynamically generates the RMI-IIOP stub classes that are required for invocation of remote EJB interfaces, there remain some cases where these stub classes are not dynamically generated. For EJB 3.0 clients not running inside a Web container, EJB container, or client container, that is upgraded to EJB 3.0 level, you must generate the stub classes with the createEJBStubs tool, then make the generated stubs available in the client environment's classpath. Typically you would accomplish this by copying the generated stubs to the location where the client's business interface class resides.

To summarize, the createEJBStubs tool must be used to generate client-side stubs for the following environments:

- ″Bare″ Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.
- WebSphere Application Server container environments prior to Version 7 that do not have the Feature Pack for EJB 3.0 applied.
- Non-WebSphere Application Server environments.

For more information about packaging your EJB module, see the topic, ″EJB 3.0 module packaging overview.″

Developing read-only entity beans

In addition to the existing Enterprise JavaBeans (EJB) caching options, you can develop read-only entity beans.

About this task

You are most likely to want to use it under the following conditions:

- Your application uses data that change relatively infrequently. An example might be a retailing application that uses pricing data that only changes once a week or month.
- Your application can tolerate data that may be stale. The degree of “staleness” that the EJB container allows is configurable by the user.
- The bean is coded in a thread-safe manner, so it can safely be invoked by multiple threads at once.

To use this function, you declare the bean type as *read-only* the same way you currently select the bean caching options, through a selection list within an assembly tool.

To complete this task see the topic, ″Defining bean cache settings for a bean″ in the assembly tool documentation at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html.

Example: Using a read-only entity bean

A usage scenario and example for writing an Enterprise JavaBeans (EJB) application that uses a read-only entity bean.

Usage scenario

A customer has a database of catalog pricing and shipping rate information that is updated daily no later than 10:00 PM local time (22:00 in 24-hour format). They want to write an EJB application that has read-only access to this data. That is, this application never updates the pricing database. Updating is done through some other application.

Example

The customer’s entity bean local interface might be:

```
public interface ItemCatalogData extends EJBLocalObject {

  public int getItemPrice();

  public int getShippingCost(int destinationCode);

}
```

The code in the stateless SessionBean method (assume it’s a TxRequired) that invokes this EntityBean to figure out the total cost including shipping, would look like:

```
.....
// Some transactional steps occur prior to this point, such as removing the item from
 // inventory, etc.
 // Now obtain the price of this item and start to calculate the total cost to the purchaser

 ItemCatalogData theItemData =
    (ItemCatalogData) ItemCatalogDataHome.findByPrimaryKey(theCatalogNumber);

int totalcost = theItemData.getItemPrice();

// ...     some other processing, etc. in the interim
```

```
// ...
// ...

// Add the shipping costs
totalcost = totalcost + theItemData.getShippingCost(theDestinationPostalCode);
```

At application assembly time, the customer sets the EJB caching parameters for this bean as follows:
- ActivateAt = ONCE
- LoadAt = DAILY
- ReloadInterval = 2200

On the first call to the getItemPrice() method after 22:00 each night, the EJB container reloads the pricing information from the database. If the clock strikes 22:00 between the call to getItemPrice() and getShippingCost(), the getShippingCost() method still returns the value it had prior to any changes to the database that might have occurred at 22:00, since the first method invocation in this transaction occurred prior to 22:00. Thus, the item price and shipping cost used remain in sync with each other.

Migrating enterprise bean code to the supported specification

Support for the Enterprise JavaBeans (EJB) 3.0 specification is added for this product.

Before you begin

There should not be migration issues associated with using EJB 3.0 beans. Existing applications should continue to run as-is and compile without error.

Note: The EJB 3.0 specification has deprecated the use of EJB 1.1 style entity beans. While using EJB 3.0 modules in the product has not yet been deprecated, you are encouraged to start migrating to Java Persistence API (JPA) or JDBC.

About this task

Follow these steps as appropriate for your application deployment.

1. Modify enterprise bean code for changes in the specification.
   - You need to migrate the Version 1.1 beans to Version 2.x beans and redploy them on the product. .

     Note: The EJB Version 2.0 specification mandates that prior to the EJB container's running a findBy*Method* query, the state of all enterprise beans enlisted in the current transaction be synchronized with the persistent store. This is done so that the query is performed against current data. If Version 1.1 beans are reassembled into an EJB 2.x-compliant module, the EJB container synchronizes the state of Version 1.1 beans, as well as that of Version 2.x beans. As a result, you might notice some change in application behavior even though the application code for the Version 1.1 beans has not been changed.
2. Reassemble and redeploy all modules to incorporate migrated code.

Migrating enterprise bean code from Version 1.1 to Version 2.1

Enterprise JavaBeans (EJB) Version 2.1-compliant beans can be assembled only in an EJB 2.1-compliant module, although an EJB 2.1-compliant module can contain a mixture of Version 1.x and Version 2.1 beans.

About this task

The EJB Version 2.1 specification mandates that prior to the EJB container starting a findBy*Method* query, the state of all enterprise beans that are enlisted in the current transaction be synchronized with the persistent store. (This action is so the query is performed against current data.) If Version 1.1 beans are reassembled into an EJB 2.1-compliant module, the EJB container synchronizes the state of Version 1.1

beans as well as that of Version 2.1 beans. As a result, you might notice some change in application behavior even though the application code for the Version 1.1 beans has not been changed.

The following information generally applies to any enterprise bean that currently complies with Version 1.1 of the EJB specification. For more information about migrating code for beans produced with the Rational Application Developer tool, see the documentation for that product.

1. In beans with container-managed persistence (CMP) version 1.x, replace each CMP field with abstract get and set methods. In doing so, you must make each bean class abstract.
2. In beans with CMP version 1.x, change all occurrences of `this.`*`field`* `=` *`value`* to `set`*`Field`*`(`*`value`*`)`.
3. In each CMP bean, create abstract get and set methods for the primary key.
4. In beans with CMP version 1.x, create an EJB Query Language statement for each finder method.

   Note: EJB Query Language has the following limitations in Application Developer Version 5:
   - EJB Query Language queries involving beans with keys made up of relationships to other beans appear as invalid and cause errors at deployment time.
   - The IBM EJB Query Language support extends the EJB 2.1 specification in various ways, including relaxing some restrictions, adding support for more DB2 functions, and so on. If portability across various vendor databases or EJB deployment tools is a concern, then care should be taken to write all EJB Query Language queries strictly according to instructions described in Chapter 11 of the EJB 2.1 specification.
5. In finder methods for beans with CMP version 1.x, return java.util.Collection instead of java.util.Enumeration.
6. Update handling of non-application exceptions.
   - To report non-application exceptions, throw javax.ejb.EJBException instead of java.rmi.RemoteException.
   - Modify rollback behavior as needed: In EJB versions 1.1 and 2.1, all non-application exceptions thrown by the bean instance result in the rollback of the transaction in which the instance is running; the instance is discarded. In EJB 1.0, the container does not roll back the transaction or discard the instance if it throws java.rmi.RemoteException.
7. Update rollback behavior as the result of application exceptions.
   - In EJB versions 1.1 and 2.1, an application exception does not cause the EJB container to automatically roll back a transaction.
   - In EJB Version 1.1, the container performs the rollback only if the instance has called setRollbackOnly() on its EJBContext object.
   - In EJB Version 1.0, the container is required to roll back a transaction when an application exception is passed through a transaction boundary started by the container.
8. Update any CMP setting of application-specific default values to be inside ejbCreate (not using global variables, since EJB 1.1 containers set all fields to generic default values before calling ejbCreate, which overwrites any previous application-specific defaults). This approach also works for EJB 1.0 CMPs.

Adjusting exception handling for EJB wrapped applications migrating from version 5 to version 7

Because of a change in the Java APIs for XML based Remote Procedure Call (JAX-RPC) specification, Enterprise JavaBeans (EJB) applications that could be wrapped in WebSphere Application Server Version 5.1 cannot be wrapped in version 6 or 7 unless you modify the code to the exception handling of the base EJB application.

About this task

Essentially, the JAX-RPC version 1.1 specification states:

```
a service specific exception declared in a remote method signature must be a
checked exception. It must extend java.lang.Exception either directly or indirectly
but it must not be a RuntimeException.
```

So it is no longer possible to directly use java.lang.Exception or java.lang.Throwable types. You must modify your applications using service specific exceptions to comply with the specification.

1. Modify your applications that use service specific exceptions. For example, say that your existing EJB uses a service specific exception called UserException. Inside of UserException is a field called *ex* that is type java.lang.Exception. To successfully wrapper your application with Web services in WebSphere Application Server version 7, you must change the UserException class . In this example, you could modify UserException to make the type of *ex* to be *java.lang.String* instead of java.lang.Exception.

```
new UserException class:

package irwwbase;

/**
 * Insert the type's description here.
 * Creation date: (9/25/00 2:25:18 PM)
 * @author: Administrator
 */


public class UserException extends java.lang.Exception {

       private java.lang.String _infostring = null;
       private java.lang.String ex;
/**
 * UserException constructor comment.
 */

public UserException() {
       super();
}
/**
 * UserException constructor comment.
 */
public UserException (String infostring)
{
       _infostring = infostring;
} // ctor
/**
 * Insert the method's description here.
 * Creation date: (11/29/2001 9:25:50 AM)
 * @param msg java.lang.String
 * @param ex java.lang.Exception
 */
public UserException(String msg,String t) {
       super(msg);
       this.setEx(t);

       }
       /**
        * @return
        */
       public java.lang.String get_infostring() {
              return _infostring;
       }

       /**
        * @return
        */
       public java.lang.String getEx() {
              return ex;
       }

       /**
        * @param string
        */
       public void set_infostring(java.lang.String string) {
```

```
            _infostring = string;
        }

        /**
         * @param Exception
         */
        public void setEx(java.lang.String exception) {
            ex = exception;
        }

        public void printStackTrace(java.io.PrintWriter s) {
          System.out.println("the exception is :"+ex);
          }

}
```

2. Modify all of the exception handling in the enterprise beans that use it. You must ensure that your enterprise beans are coded to accept the new exceptions. In this example, the code might look like this:

```
new EJB exception handling:

try {
      if (isDistributed()) itemCMPEntity = itemCMPEntityHome.findByPrimaryKey(ckey);
      else itemCMPEntityLocal = itemCMPEntityLocalHome.findByPrimaryKey(ckey);
  } catch (Exception ex) {
       System.out.println("%%%%% ERROR: getItemInstance - CMPjdbc " + _className);
       ex.printStackTrace();
       throw new UserException("error on itemCMPEntityHome.findByPrimaryKey(ckey)",ex.getMessage());
   }
```

WebSphere extensions to the Enterprise JavaBeans specification

This article outlines extensions to the Enterprise JavaBeans (EJB) specification provided with the product.

Inheritance in enterprise beans

In the Java language, *inheritance* is the creation of a new class from an existing class or a new interface from an existing interface. This product supports two forms of inheritance: standard class inheritance and EJB inheritance.

In standard class inheritance, the home interface, remote interface, or enterprise bean class inherits properties and methods from base classes that are not themselves enterprise bean classes or interfaces.

By contrast in enterprise bean inheritance, an enterprise bean inherits properties, uch as container-managed persistence (CMP) fields and container-managed relationship (CMR) fields, methods, and method-level control descriptor attributes from another enterprise bean.

For more information, see the documentation for the assembly tools at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html.

Optimistic concurrency control for container-managed persistence

This product supports optimistic concurrency control of data access. See “Concurrency control” on page 194 for more information.

Access intents for EJB persistence

This product supports the application of named data-access policies.

Sequence grouping for container-managed persistence

By designating CMP sequence groups for entity beans, you can prevent certain types of database-related exceptions from occurring during the run time of your EJB application. Within each group you specify the order in which the beans update your relational database tables. See “Setting the run time for CMP sequence groups” on page 214 for instructions.

Performance enhancements

Through the lifetime-in-cache settings, this product provides a way for you to improve performance for beans that are only occasionally updated. For more information, see the topic, ″Entity bean assembly settings″ located in the assembly tool documentation at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html.

Some enterprise beans created with the assembly tools can utilize *read-ahead* for loading a bean and its related beans in a single database operation. An entire object graph or any part of the graph can be preloaded by configuring a finder method to use read-ahead.

Assembly and deployment extensions

This product supports IBM extensions of assembly and deployment options.

Enterprise bean development best practices

Use the following guidelines when designing and developing enterprise beans.

- Use a stateless session bean to act as the entry point for business logic.
- Entity beans should use container-managed persistence.
- In an Enterprise JavaBeans (EJB) Version 2.x and later version environments, use local interfaces to improve communication between enterprise beans in the same Java virtual machine.

  Local calls avoid the overhead of RMI/IIOP and use pass-by-reference semantics instead of pass-by-value. For each call, the caller and callee beans share the state of arguments. EJB 2.x and later beans can have both a local and remote interface, but more typically, have one or the other.
- For communicating with remote clients, provide remote and remote home interfaces. For communicating with local clients like servlets, entity beans, and message-driven beans, provide local and local home interfaces.

Batched commands for container managed persistence

From JDBC 2.0 on, *PreparedStatement* objects can maintain a list of commands that can be submitted together as a batch. Instead of multiple database round trips, there is only one database round trip for all the batched persistence requests.

You can enable the use of this feature for EJB container managed persistence (CMP). When you do, the run time defers *ejbStore/ejbCreate/ejbRemove* or the equivalent database persistence requests (insert/update/delete) until they are needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. When the persistence operation finally happens, run time accumulates the database requests and uses JDBC *PreparedStatement* batch operation to make a single JDBC call for multiple rows of the same operation.

The product enables you to make the same settings using assembly tools.

Deferred Create for container managed persistence

For CMP during the *ejbCreate*, the container can create the representation of the entity in the database immediately, or defer it to a later time.

You can turn this option on from the EJB CMP side. When you choose this option, the runtime defers *ejbCreate*, or the equivalent database persistence request, until it is needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. By doing this you can reduce two round trips for the newly created entity (insert and update) to one (insert).

The product enables you to make the same settings using assembly tools. Review the assembly tools information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

Partial column updates for container managed persistence

Previously, the WebSphere Application Server implementation of the Container Managed Persistence (CMP) bean method *ejbStore* stored all of the persistent attributes of the CMP bean to the database, even if only a subset of persistent attribute fields were changed. This needless performance degradation is eliminated in this release of the product.

Note: Entity beans are not supported in EJB 3.0 modules.

For Enterprise JavaBeans (EJB) 2.x CMP entity beans, you can use the *partial update* feature to specify how you want to update the persistent attributes of the CMP bean to the database. This feature is provided as a bean level persistence option, called *PartialOperation*, in the access intent policy configured for the bean. PartialOperation has two possible values:

NONE Partial update is turned off. All of the persistent attributes of the CMP bean are stored to the database. This is the default value.

UPDATE_ONLY
Specifies that updates to the database occur only for the persistent attributes of the CMP bean that are changed.

For information on how to set partial update, see “Setting partial update for container-managed persistent beans” on page 181.

Performance

Performing partial updates increases performance in several ways:

- by reducing query execution time, since only a subset of the columns are in the query. Improvement is higher for tables with many columns and indexes. When the table has many indexes only the indexes affected by the updated columns need to be updated by the backend database.
- by reducing network input and output since there is less data to be transmitted.
- by saving any processing time for non-trivially mapped columns. For example, if a column uses converters, composers, and transformations to partially inject the input record.
- by eliminating unnecessary firing of update triggers. If a CMP bean field is not changed, any trigger depending only on the corresponding column is not fired.

Although partial update improves performancel, it can adversely affect performance as follows:

- If you enable partial update for a bean that your application modifies several different combinations of columns during the same time span, the prepared statement cache maximum for the connection is reached very quickly. As a result, statement handles are evicted from the cache based on least recent usage. This results in statements being prepared repeatedly, decreasing performance for all CMP functions, not just limited to the ejbStore method.
- Partial update query templates cached in the function set increase memory use. The increase is linear relative to the number of fields in the CMP bean for which the partial update access intent option is turned on.
- The PartialOperation persistent option, when used in combination with the Batch Update persistent option, affects the performance of the batch update because each partial query is different. There is an

execution time cost incurred for generating a partial update query string dynamically. Since query fragments are stored for each column, the execution cost to assemble the query fragments is linear, based on the number of CMP bean fields dirtied.
- There are condition checks for each CMP field, for example, to inspect the dirty flags and to execute the preparedStatement setXXX method calls.

Considerations for using partial update

The performance gains you hope to achieve should be weighed against the possible instances where degradation can occur. You can use the following guidelines to help you make the decision.
- Partial update might not benefit an application that only involves a small table with a few columns and simple data types and no update triggers. The cost to assemble the partial query dynamically outweighs the performance gain.
- Partial update is a benefit if there is a complex data type that is not updated often. An example of a complex data type is an employee bean with a “photo” CMP attribute mapped to a BLOB OR VARGRAPHIC, or similar complex backend type, that is typically stored in a different location in the database manager implementation.
- Partial Update might benefit if there are several VARCHAR type columns and only a very few of them are updated.
- It is better not to use the partial operation if the application can randomly be updating different combinations of columns and the number of assignable columns (non-key) is higher than five. This generates different partial queries and fills up the prepared statement cache quickly. But, if the bean does not have too many columns, for example, four or less, and it has complex data types, you might consider turning partial update on, with the option of increasing the statement cache size to ensure an increased number of queries. For information on increasing the statement cache size, refer to Data source settings.
- Partial Update is beneficial when there are update triggers needed on a subset of columns.
- Partial Update is beneficial when the table has many columns and indexes, and only a few indexes are touched by a typical update.

Restrictions

By default, batch update of *update queries* is disabled for all CMP beans for which partial update is enabled. In other words, partial update takes precedence over batch update. Batch update of delete and insert queries is not affected.

Batch update performance is affected when both batch update and partial update persistence options are used on the same bean, because each partial query is different. You can use the JVM property, *-Dcom.ibm.ws.pm.grouppartialupdate=true*, to group the similar partial update queries into a batch update. Grouping partial updates only helps when there are several partial queries with the same shape in a transaction. Otherwise, grouping partial updates has the opposite affect on performance. Because this setting is not on a bean level basis, you should be careful when turning it on. Because this affects all beans that have both partial update and batch update on, you must make sure that batch update of partial queries does increase performance when viewed across all the beans for which both updates are on.

To set the JVM property:

1. Open the server.xml file.
2. Change the value of *-Dcom.ibm.ws.pm.grouppartialupdate=true* to *-Dcom.ibm.ws.pm.grouppartialupdate=false*.

Setting the run time for batched commands with JVM arguments

This article explains how to set the run time for batched commands with JVM arguments.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.

4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.pm.batch=true*.

Setting the run time for deferred create with JVM arguments

For Container Managed Persistence (CMP) to happen during the ejbCreate, the Enterprise JavaBeans (EJB) container can create the representation of the entity in the database immediately, or defer it to a later time.

About this task

When you choose the defer option, the run time defers ejbCreate, or the equivalent database persistence request, until it is needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. By doing this you can reduce two round trips for the newly created entity (insert and update) to one (insert).

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.pm.deferredcreate=true*.

Setting partial update for container-managed persistent beans

For Enterprise JavaBeans (EJB) 2.x CMP entity beans, you can use the *partial update* feature to specify how you want to update the persistent attributes of the CMP bean to the database. This feature is provided as a bean-level persistence option, called *PartialOperation*, in the access intent policy configured for the bean.

About this task

See the topic, ″Partial operation for container managed persistence″ in the assembly tool information center to learn how to complete this task with the assembly tool.

You can access the assembly tool information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

Setting persistence manager cache invalidation

To set persistence manager cache invalidation, follow these steps.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Server Infrastructure area, select **Java and Process Management**.
6. Select **Process Definition**.
7. In the Additional Properties area, select **Java Virtual Machine**.
8. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.ejbpersistence.cacheinvalidation=true*.

Setting the system property to enable remote EJB clients to receive nested or root-cause exceptions

You might want to code your application to perform a given action if a certain kind of exception is the root-cause of a failure and is nested within the exception that you receive. The default behavior in the product might mask a nested or root-cause exception in your application.

About this task

The Enterprise JavaBeans (EJB) container creates a TransactionRolledbackException exception for a remote client when it can create a RemoteException exception instead. With the RemoteException exception, the container does not lose the ability to have root-cause information nested inside the exception.

You can set the following Java virtual machine (JVM) system property to **true** through the administrative console for the product: com.ibm.websphere.ejbcontainer.includeRootExceptionOnRollback This change enables the remote client to receive nested exceptions when a rollback occurs.

Note: This property is applicable only for scenarios where the transaction in which the bean method is running was started by the container for this specific method invocation. All of the other scenarios must result in a TransactionRollBackException exception according to the EJB specification.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Servers > Application servers > server_name**.
4. Under Server infrastructure, select **Java and Process Management > Process Definition**.
5. Under Additional properties, select **Java virtual machine > Custom properties > New**.
6. In the **Name** entry field, type **com.ibm.websphere.ejbcontainer.includeRootExceptionOnRollback**.
7. In the **Value** entry field, type **true**.
8. Select **OK**.

Unknown primary-key class

When writing an entity bean, the minimum requirements usually include a primary-key class. However, in some cases you might choose not to specify the primary-key class for an entity bean with container-managed persistence (CMP).

Perhaps there is no obvious primary key, or you want to allow the deployer to select the primary key fields at deployment time. The primary key type is usually derived from the type used by the database system that stores the entity objects, and you might not know what this key is.

So, the *unknown key type* is actually a type chosen at deployment time, making it changeable each time the bean is deployed. Your client code must deal with this key as type *Object*.

Currently, WebSphere Application Server supports top-down mapping and enables the deployer to choose *String* keys generated at the application server. For an example of how to use this function, see the Samples Gallery.

Configuring a timer service

To configure a timer service, follow these steps.

1. Open the administrative console.
2. Click **Servers** → **Application servers** → **server_name** → **EJB Container settings** → **EJB timer service settings**. The timer service settings panel is displayed.

3. If you want to use the internal, or pre-configured, scheduler instance, click the **Use internal EJB timer service scheduler instance** radio button. If you choose not to change the default settings, this instance is associated with a Apache Derby database. If you choose to customize the pre-configured instance:
   a. To change the data source (you can use any supported database, such as DB2 or Oracle), enter your **Data source JNDI name**.
   b. Enter your chosen **Data source alias**.
   c. Enter your chosen **Table prefix** if you want to have several server processes use the same database, but different tables.
   d. Enter a **Poll interval** value in milliseconds.
   e. If you want more timers to execute concurrently, enter a new value for **Number of timer threads**.

   For more information about the fields, see “EJB timer service settings” on page 185
4. If you want to configure your own scheduler instance instead of using the pre-configured internal one, click the **Use custom scheduler instance** radio button. Some reasons you might want to use your own instance are:
   - to change scheduler service configuration options not available for customization on this panel
   - to keep EJB Timer tasks in the same database tables as your other tasks
   - you are running in a Clustered environment and want to have a single scheduler instance handle all of the EJB Timers for the cluster. This way, an *ejbTimer* Task created on one cluster member can execute on a different cluster member.

   To use your own instance, you must:
   a. Configure a scheduler instance through the Scheduler Service graphical user interface. See “Using schedulers” on page 1659 for information on how to do this.
   b. Select your **Scheduler JNDI name** from the list.
5. Click **Apply**.
6. Click **OK**.

Configuring a timer service for network deployment

Use this task to configure the Enterprise JavaBeans (EJB) timer service to be used across multiple servers.

About this task

This is largely a question of using the same data source. The steps that follow assume that you have already created a database instance (for example, DB2 or Oracle). From there, you must configure the timer service to use that database.

There are two ways to configure the timer service to share the same database across multiple servers. Choose either step 1 or 2.

1. **Configure a scheduler instance for the cluster, then configure the timer service to use that scheduler instance.**
   a. Configure a scheduler instance for the cluster. This creates for you a *custom scheduler instance*. Next you need to configure the timer service to use that custom instance.
   b. Open the administrative console.
   c. Click **Servers** >**Application Servers** > *servername* > **EJB Container Settings** > **EJB timer service settings**. The timer service settings panel appears.
   d. Select the **Use custom scheduler instance** radio button.
   e. Select your **Scheduler JNDI name** from the dropdown list.
   f. Click **Apply**.
   g. Click **OK**.

2. **Configure the timer service default scheduler instance for each server to use the same data source.**
   a. Select the **Use internal EJB timer service scheduler instance** radio button. To customize the pre-configured instance:
   b. To change the data source (you can use any supported database, such as DB2 or Oracle) select your **Data source JNDI name** from the dropdown list. The default database listed cannot be shared, because it is configured to be visible to one server only, and it uses the single server version of Apache Derby, which can only be accessed by one server process at a time.
   c. Enter your chosen **Datasource Alias**.
   d. Enter your chosen **Table Prefix** if you want to have several server processes use the same database, but different tables.
   e. Enter a **Poll Interval** value in milliseconds. For more information about the fields, see "EJB timer service settings" on page 185
   f. Click **Apply**.
   g. Click **OK**.
   h. Change all of your server processes to use the same database you chose from the **Data source JNDI name** dropdown list earlier.

Example: Using the Timer Service

This example shows the implementation of the *ejbTimeout()* method that is called when the scheduled event occurs.

The ejbTimeout method can contain any code that is typically placed in a business method of the bean. Method-level attributes such as *transaction* or *runAs* can be associated with this method by the application assembler. An instance of the Timer object that causes the method to fire is passed in as an argument to ejbTimeout method.

```
import javax.ejb.Timer;
import javax.ejb.TimedObject;
import javax.ejb.TimerService;

public class MyBean implements EntityBean, TimedObject {

  // This method is called by the EJB container upon Timer expiration.
  public void ejbTimeout(Timer theTimer) {

    // Any code typically placed in an EJB method can be placed here.

     String whyWasICalled = (String) theTimer.getInfo():
     System.out.println("I was called because of"+ whyWasICalled);
  } // end of method ejbTimeout
```

A Timer is created that starts the ejbTimeout method in 30 seconds. A simple string object is passed in at Timer creation to identify the Timer.

```
// Instance variable to hold the EJB context.
private EntityContext theEJBContext;

// This method is called by the EJB container upon bean creation.
public void setEntityContext(EntityContext theContext) {

// Save the entity context passed in upon bean creation.
  theEJBContext = theContext;

}

// This business method causes the ejbTimeout method to begin in 30 seconds.
public void fireInThirtySeconds() throws EJBException  {

  TimerService theTimerService = theEJBContext.getTimerService();
```

```
  String aLabel = "30SecondTimeout";
  Timer theTimer = theTimerService.createTimer(30000, aLabel);

} // end of method fireInThirtySeconds

} // end of class MyBean
```

EJB timer service settings

Use this page to configure and manage the Enterprise JavaBeans (EJB) timer service for a specific EJB container.

To view this administrative console page, click**Servers** → **Server types** → **WebSphere application servers** → **server** → **EJB Container Settings** → **EJB timer service settings**.

The two radio buttons that appear on this page offer you choices that are *mutually exclusive*.

Scheduler type:

Use internal EJB timer service scheduler instance:

The product provides an internal scheduler instance for use by the EJB timer service. The internal scheduler instance is pre-configured for basic EJB timer functionality, and provides limited configuration settings for an EJB timer service. Clicking this button specifies that you want to use the internal scheduler instance to manage your tasks. They are persisted to a Cloudscape database associated with the server process. Selecting this choice locks out the *Use Custom Scheduler Instance* option.

This is the default choice.

Use custom scheduler instance:

You can perform a more advanced configuration for the EJB timer service by defining a custom scheduler instance. Scheduler configuration provides more configuration options than the internal EJB timer service pre-configured scheduler instance. You might want to define a custom scheduler instance when running in a clustered environment, allowing all cluster members to run with a single scheduler instance. This enables EJB Timers created on one cluster member to execute on other cluster members. Providing a custom scheduler instance also enables EJB Timers to be maintained in the same database as other scheduled tasks. Selecting this choice locks out the *Use Internal EJB Timer Service Scheduler Instance* option

Data source JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name of the data source where persistent EJB Timers are stored for this EJB container. Any data source available in the name space can be used for EJB Timers. Multiple EJB containers can share a single data source while using different tables by specifying a table prefix.

Data type	String
Default	*jdbc/DefaultEJBTimerDataSource*

Data source alias:

Authentication alias to a user name and password used to access the data source.

Data type	String

Table prefix:

A string prepended to the EJB timer service table names (TASK, TREG, LMGR and LMPR). These tables are created during server start if they do not already exist. See help on the Scheduler Service for information about manually creating these tables. Multiple independent EJB timer services can share the same database if each instance specifies a different prefix string.

Data type	String
Default	*EJBTIMER_*

Poll interval:

The interval at which the EJB timer service daemon polls the database. Each poll operation can be expensive. If the interval is extremely small and there are many scheduled tasks, polling can consume a large portion of system resources. New Timers set to expire sooner than this interval might not execute until the interval ends. If this value is too large, a potentially large number of timer events might be read into memory, because all the timer events occurring in the next poll interval are read in each time.

Data type	Integer
Units	seconds
Default	300
Range	3 -- 1800

Number of timer threads:

The number of threads used to execute concurrent EJB Timer tasks. Setting the number of Timer Threads to zero disables the EJB timer service.

Data type	Integer
Default	1
Range	0 -- 500

Scheduler JNDI name:

This field is only used when the **Use Custom Scheduler Instance** choice is made. It specifies the JNDI name of a custom Scheduler instance to use for managing and persisting EJB Timers. Internal EJB timer service Scheduler Instance configuration information is not applied to the specified Scheduler instance.

Data type	String

Developing enterprise beans for the timer service

About this task

In WebSphere Application Server, the **EJB Timer Service** implements Enterprise JavaBeans (EJB) Timers as a new kind of Scheduler Service task. By default, an internal (or pre-configured) scheduler instance is used to manage those tasks, and they are persisted to a Apache Derby database associated with the server process.

However, you can perform some basic customization to the internal scheduler instance. For information about how to do this customization, see “Configuring a timer service” on page 182.

Creation and cancellation of Timer objects are transactional and persistent. That is, if a Timer object is created within a transaction and that transaction is later rolled back, the Timer object’s creation is rolled back as well. Similar rules apply to the cancellation of a Timer object. Timer objects also survive across application server shutdowns and restarts.

1. Write your enterprise bean to implement the *javax.ejb.TimedObject* interface, including the *ejbTimeout()* method. The bean calls the *EJBContext.getTimerService()* method to get an instance of the **TimerService**object. The bean calls the TimerService method to create a Timer. This Timer is now associated with that bean.
2. After you create it, you can pass the Timer instance to other Java code as a *local* object.

   Note: For WebSphere Application Server Version 6, no assembly tooling supports the Enterprise JavaBeans timedObject. To set the ejbTimeout method transaction attribute you must manually enter the attributes in the deployment descriptor. See “EJB timer service settings” on page 185 for more information.

Clustered environment considerations for timer service

In a single server environment, it is clear which server instance should invoke the *ejbTimeout()* method on a given bean. In a multi-server clustered environment there are two possibilities.

- Separate timer service database per server process or cluster member. This is the default configuration. Only the server instance or cluster member that created the Timer can access the Timer and run the *ejbTimeout()* method. If the server instance is unavailable, the Timer does not run at the specified time, and does not run until the server is restarted. Also, if an enterprise bean calls the *getTimers()* method, only those timers created on the server instance are found. This can cause unexpected behavior if the enterprise bean attempts to cancel all timers associated with it; for example, when the enterprise bean is removed. This configuration is NOT recommended for production level systems.
- Shared or common timer service database for the cluster. Timers can be created and accessed on any server process or cluster member. Timers created in one server process are found by the *getTimers()* method on other server processes in the cluster. When an entity bean is removed, all timers, no matter where created, are cancelled. However, all timers are executed on a single server in the cluster, that is, the *ejbTimeout()* method is run for all timers on a single server. Which server executes the timers varies depending on which server process obtains a lock on the common database tables. If the server executing timers becomes unavailable, then another server or cluster member takes over and begins executing all timers at their scheduled time. This is the recommended configuration for all production level systems.
-

  Note: When using the EJB Timer service in an application using multi-threaded database access, application flow can introduce deadlock problems.
  To avoid this, use the wsPessimisticUpdate access intent. This access intent causes the finder method in your application to run a *select for update* statement instead of a generic select. This in turn prevents the lock escalation deadlock when multiple threads try to escalate their locks to perform an update.

See “Configuring a timer service” on page 182 for information on how to configure the data source (database) to be used for each server process timer service.

Note: Once the data source for the timer service is changed to point to a different database, the server process automatically attempts to create the required tables in that database on the next server start.

If the user Id associated with the start of the server process is not authorized to create database tables in the configured timer service database, then the tables must be created manually. For more information, see Creating scheduler tables using DDL files.

Timer service commands:

Information about Enterprise JavaBeans (EJB) timers is generally specific to the application that creates the timers, and the timers are not visible outside of the creating application. Therefore, management of EJB timers should be performed by the application that contains the enterprise bean and that creates the EJB timer.

However, you can use the following commands during application development. They provide some basic EJB timer management functions. These commands are not available on *client only* installs.

findEJBTimers

This command displays information about existing EJB timers based on specified filter criteria.

The syntax for this command is:

```
findEJBTimers server filter [options]
   filter: -all | -timer | -app [-mod [-bean ]]
           -all
           -timer timer id
           -app   application name
           -mod   module name
           -bean  bean name

  options: -host  host name
           -port  portnumber
           -conntype connector type
           -user  userid
           -password password
           -quiet
           -logfile filename
           -replacelog
           -trace
           -help
```

where :

server the name of the server process where the EJB timers are located

-all find all EJB timers associated with the server process

timer id
EJB Timer ID that uniquely identifies the timer

application name
find all EJB timers associated with the application

module name
find all EJB timers associated with the module

bean name
find all EJB timers associated with the enterprise bean

host name
host name of the server process

portnumber
port of the server process

connector type
type of connection. For example, SOAP, RMI, or NONE.

userid user to use when connecting to the server process

password
password to use when connecting to the server process

quiet disable output

logfile directs output to a file

replacelog
clears the existing log before executing the command

trace enable trace

help provides command-specific help

Note: If the server you specify is configured to use a scheduler instance that is shared by multiple servers, then EJB timers created in any of the server processes might be found.

For an example of the findEJBTimers command, see “FindEJBTimers command example” on page 190.

cancelEJBTimers

This command cancels and removes from persistent storage EJB timers based on the specified filter criteria.

The syntax for this command is:

```
cancelEJBTimers server filter [options]
   filter: -all | -timer | -app [-mod [-bean ]]
           -all
           -timer timer id
           -app   application name
           -mod   module name
           -bean  bean name

  options: -host  host name
           -port  portnumber
           -conntype connector type
           -user  userid
           -password password
           -quiet
           -logfile filename
           -replacelog
           -trace
           -help
```

where :

server the name of the server process where the EJB timers are located

-all find all EJB timers associated with the server process

timer id
EJB Timer ID that uniquely identifies the timer

application name
find all EJB timers associated with the application

module name
find all EJB timers associated with the module

bean name
find all EJB timers associated with the enterprise bean

host name
host name of the server process

portnumber
port of the server process

connector type
type of connection. For example, SOAP, RMI, or NONE.

userid user to use when connecting to the server process

password
password to use when connecting to the server process

quiet disable output

logfile directs output to a file

replacelog
clears the existing log before executing the command

trace enable trace

help provides command-specific help

Note: If the server you specify is configured to use a scheduler instance that is shared by multiple servers, then EJB timers created in any of the server processes might be cancelled.

For an example of the cancelEJBTimers command, see “CancelEJBTimers command example.”

FindEJBTimers command example:

The following examples illustrate how to use the command to find Enterprise JavaBeans (EJB) timers and explain the output statement.

To use the findEJBTimers command to find *all* EJB timers on a server called **server1**:

```
findEJBTimers server1 -all
```

To find all EJB timers on **server1**, associated with the *Increment* bean in the **DefaultApplication**:

```
findEJBTimers server1 -app DefaultApplication.ear -mod Increment.jar -bean Increment
```

When EJB timers matching the filter criteria are found, the output appears similar to this:

```
EJB Timer : 25     Expiration: Mon Feb 09 13:36:47 CST 2004   Repeating
 EJB        : DefaultApplication.ear Increment.jar Increment
 EJB Key: 8
 Info : Increment Counter
EJB Timer : 26     Expiration: Mon Feb 09 13:36:47 CST 2004   Single
 EJB        : DefaultApplication.ear Increment.jar Increment
 EJB Key: 8
 Info : Decrement Counter
2 EJB Timers found
```

In this output:

- The *EJB Timer* is the unique identifier of the timer.
- *Expiration* is the next time the timer is expected to execute.
- *Repeating* or *Single* indicates whether the EJB timer is single action or repeating.
- *EJB Key* is the *toString()* method output of the primary key for the Entity enterprise bean (not present for other EJB types).
- *Info* is the *toString()* method output of the object passed by the application when the EJB timer was created.

Only the first 40 bytes of *toString()* output are displayed for the Primary Key and Timer Info. This information is only useful if the application overrides the *toString()* method for these objects.

Increment in the *DefaultApplication* does not implement the *TimedObject* interface, and so could not actually have associated EJB Timers. Increment is used merely for illustrative purposes in this example.

CancelEJBTimers command example:

The following examples illustrate how to use the command to cancel Enterprise JavaBeans (EJB) timers.

To use the cancelEJBTimer command to cancel all EJB timers on a server called **server1**:

```
cancelEJBTimers server1 -all
```

To cancel all EJB timers on **server1**, associated with the *Increment* bean in the **DefaultApplication**:

```
cancelEJBTimers server1 -app DefaultApplication.ear -mod Increment.jar -bean Increment
```

To cancel a specific EJB timer identified through the FindEJBTimers command or from a system log entry indicating a problem or failure:

```
cancelEJBTimers server1 -id 25
```

Increment in the *DefaultApplication* does not implement the *TimedObject* interface, and so could not actually have associated EJB Timers. Increment is used merely for illustrative purposes in this example.

Web services support in EJB

The product complies with the Java EE and Enterprise JavaBeans (EJB) specifications by enabling you to expose an EJB stateless session bean as a Web service.

The product supports the Java API for XML Web Services (JAX-WS) 2.0 programming model and the EJB 3.0 specification.

EJB 3.0 does not support the @WebService or @WebMethod annotations on EJB 3.0 stateless session beans, which are used to identify the stateless session bean as a JAX-WS implementation, nor does it support injection of Web services references. You can invoke EJB 3.0 beans indirectly by defining a servlet as the JAX-WS implementation and placing code in the servlet which invokes the target EJB 3.0 bean.

You can do this by declaring a link between the desired endpoint name in the Web service deployment descriptor of the EJB module. During deployment and installation of the bean into the Application Server environment, the bean is linked to the specified Web service endpoint.

If you are writing a stateless session bean to implement a preexisting Web Services Description Language (WSDL) interface, you must remember to implement in your bean all of the methods defined on the WSDL interface.

For more information, see “Developing Web services applications with JAX-WS” on page 487.

Defining data sources for entity beans

Before an application that is installed on an application server can start, all enterprise bean (EJB) references and resource references defined in the application must be bound to the actual enterprise beans or resources that are defined in the application server.

Before you begin

Create a data source or JDBC resource and give it a Java Naming and Directory Interface (JNDI) name.

About this task

For more information, see Application bindings.

Before you do this task, it is assumed that the entity beans in your application are container-managed persistence (CMP) enterprise beans.

Note:
The EJB container handles the persistence of the bean attributes in the underlying persistent store. You must specify which data store is used. You do this by binding an EJB module or individual EJB to a data source.

If you bind an EJB *module* to a data source, all beans in that module use the same data source for persistence. If you specify the data source at the bean level, then that data source is used instead.

See the assembly tool information center for the steps on how to complete this task.

Lightweight local operational mode for entity beans

WebSphere Application Server provides a special operational mode called *lightweight local* mode, which can improve the performance of entity bean methods. You can decide which entity beans in your application to run in this mode.

In lightweight local mode, the container streamlines the processing that it performs before and after every method on the local home interface and local business interface of the bean. This streamlining can result in improved performance when entity bean operations are called locally from within an application. Because some processing is skipped when running in lightweight local mode, this mode can be used in certain scenarios only.

Lightweight local mode is patterned somewhat after the Plain Old Java Object (POJO) entity model introduced in the Enterprise JavaBeans (EJB) 3.0 specification. Using lightweight local mode, you can obtain some of the performance advantages of the POJO entity model without having to convert your existing EJB 2.x application code to the new POJO model. You can apply lightweight local mode to both container-managed persistence (CMP) and bean-managed persistence (BMP) entity types that meet the specific criteria.

Note: Entity beans are not supported in EJB 3.0 modules.

When to use the lightweight local mode

Lightweight local mode is designed for entity beans that are created, found, and called using the *Session Facade* pattern. Under this pattern, entity bean local home and local business methods are called from within methods of a stateless session bean or stateful session bean. The session bean methods, which can be called remotely or locally, provide security control and transaction demarcation for the entity beans that are accessed by the session bean.

You can apply lightweight local mode only to an entity bean that meets the following criteria:
- The bean implements an EJB local interface.
- No security authorization is defined on the entity bean local home or local business interface methods.
- No *run-as* security attribute is defined on the local home or local business methods.
- The classes for the calling bean and the called entity bean are loaded by the same Java classloader.
- The entity bean methods do not call the WebSphere Application Server-specific Internationalization Service or Work Area Service.

The first criterion prevents CMP 1.x beans from supporting lightweight local mode, because the 1.x beans cannot have local interfaces.

In addition, lightweight local mode provides its fullest performance benefits only to entity bean methods that do not need to start a global transaction. This condition is true if you ensure that your entity bean also meets the following criteria:
- A global transaction is already in effect when the entity bean home or business method is called. Typically, this transaction is started by the calling session bean.
- The local business interface methods and the local home methods of the entity bean use the following transaction attributes only: REQUIRED, SUPPORTS, or MANDATORY.

If an entity bean method that is running in lightweight local mode must start a global transaction, the bean still functions normally but only a partial performance benefit is realized.

You can mark an entity bean that defines a remote interface or a TimedObject interface, in addition to the local interface, for lightweight local mode. However, the performance benefit is apparent only when the bean is called through its local interface.

Applying lightweight local mode to an entity bean

WebSphere Application Server provides a special operation mode called *lightweight local* mode, which can improve the performance of entity bean methods. You can decide which entity beans in your application to run in this mode.

About this task

You can apply lightweight local mode to specific EntityBean types within your application with the Marker interface technique.

Marker interface technique

About this task

Use the marker interface technique when a group of beans within the application is related through a common inheritance hierarchy, and all the beans in the hierarchy are to be marked. For an application with a large number of beans in a hierarchy, this technique is the most efficient.

To use a marker interface, code your bean implementation class to implement the **com.ibm.websphere.ejbcontainer.LightweightLocal** interface. The bean implementation class does not need to directly implement the interface; any parent class or interface can also implement it. For details, see the **com.ibm.websphere.ejbcontainer** package in the API section of the information center.

Using access intent policies

You can use access intent policies to help the product runtime environment manage various aspects of Enterprise JavaBeans (EJB) persistence.

About this task

You apply access intent policies to EJB Version 2.0 and 2.1 entity beans, and their methods, by using an assembly tool. A set of default access intent policies comes with the assembly tool. The Rational Application Developer product provides supported assembly tools. You can also create your own custom policies. Entity beans are not supported in EJB 3.0 modules.

1. Apply default access intent to CMP entity beans. For more information, see the online help available with the assembly tools.
2. Apply access intent policies to methods of container managed persistence entity beans.
3. Create a custom access intent policy with an assembly tool.
4. Apply access intent policies to bean managed persistence (BMP) entity bean methods by using the AccessIntent API.
5. Apply multiple access intent policies to methods by using application profiling.

Access intent policies

An access intent policy is a named set of properties or access intents that govern data access for Enterprise JavaBeans (EJB) persistence. You can assign policies to an entity bean and to individual methods on an entity bean's home, remote, or local interfaces during assembly. You can set access intents only within EJB Version 2.x-compliant and later modules for entity beans with CMP Version 2.x.

This product supplies a number of access intent policies that specify permutations of read intent and concurrency control; the pessimistic and update policy can be qualified further. The selected policy determines the appropriate isolation level and locking strategy used by the run time environment.

Note: Access intent policies are specifically designed to supplemant the use of isolation level and access intent method-level modifiers found in the extended deployment descriptor for EJB version 1.1 enterprise beans. You cannot specify isolation level and read-only modifiers for EJB version 2.x and later enterprise beans.

Access intent policies configured on an entity basis define the default access intent for that entity. The default access intent controls the entity unless you specify a different access intent policy based on either method-level configuration or application profiling.

Note: Method level access intents were deprecated in Version 6.x.

You can use application profiling or method level access intent policies to control access intent more precisely. Method-level access intent policies are named and defined at the module level. A module can have one or many policies. Policies are assigned, and apply, to individual methods of the declared interfaces of entity beans and their associated home interfaces. A method-based policy is acted upon by the combination of the EJB container and persistence manager when the method causes the entity to load.

For entity beans that are backed by tables with nullable columns, use an optimistic policy with caution. The top down default mapping excludes nullable fields. You can override this when doing a meet-in-the-middle mapping. The fields used in overqualified updates are specified in the ejb-rdb mapping. If nullable columns are selected as overqualified columns, partial update should also be selected.

Note: When using DB2 for z/OS Version 8, nullable OCC columns create no problems. This is true for JDBC and SQLJ deploy options, and partial and full update.

An entity that is configured with a read-only policy that causes a bean to be activated can cause problems if updates are attempted within the same transaction. Those changes are not committed, and the process displays an exception because data integrity might be compromised.

Concurrency control

Concurrency control is the management of contention for data resources. A concurrency control scheme is considered *pessimistic* when it locks a given resource early in the data-access transaction and does not release it until the transaction is closed. A concurrency control scheme is considered *optimistic* when locks are acquired and released over a very short period of time at the end of a transaction.

The objective of optimistic concurrency is to minimize the time that a given resource is unavailable for use by other transactions. This is especially important with long-running transactions, which under a pessimistic scheme would lock up a resource for unacceptably long periods of time.

Under an optimistic scheme, locks are obtained immediately before a read operation and released immediately after. Update locks are obtained immediately before an update operation and held until the end of the transaction.

To enable optimistic concurrency, this product uses an *overqualified update scheme* to test if the underlying data source has been updated by another transaction since the beginning of the current transaction. With this scheme, the columns marked for update and their original values are added explicitly through a WHERE clause in the UPDATE statement so that the statement fails if the underlying column values have been changed. As a result, this scheme can provide column-level concurrency control; pessimistic schemes can control concurrency at the row level only.

Optimistic schemes typically perform this type of test only at the end of a transaction. If the underlying columns have not been updated since the beginning of the transaction, pending updates to container-managed persistence fields are committed and the locks are released. If locks cannot be acquired or if some other transaction has updated the columns since the beginning of the current transaction, the transaction is rolled back: All work performed within the transaction is lost.

Pessimistic and optimistic concurrency schemes require different transaction isolation levels. Enterprise beans that participate in the same transaction and require different concurrency control schemes cannot operate on the same underlying data connection.

Note: Whether to use optimistic concurrency depends on the type of transaction. Transactions with a high penalty for failure might be better managed with a pessimistic scheme. A high-penalty transaction is one for which recovery is risky or resource-intensive. For low-penalty transactions, it is often worth the risk of failure to gain efficiency through the use of an optimistic scheme. In general, optimistic concurrency is more efficient when update collisions are expected to be infrequent; pessimistic concurrency is more efficient when update collisions are expected to occur often.

Read-ahead hints

Read-ahead schemes enable applications to minimize the number of database round trips by retrieving a working set of container-managed persistence (CMP) beans for the transaction within one query. Read-ahead involves activating the requested CMP beans and caching the data for their related beans, which ensures that data is present for the beans that an application most likely needs next. A *read-ahead hint* is a representation of the related beans to read. The hint is associated with the *findByPrimaryKey* method for the requested bean type, which must be an EJB 2.x-compliant CMP entity bean.

A read-ahead hint takes the form of a character string. You do not have to provide the string; the wizard generates it for you based on the container-managed relationships (CMRs) that are defined for the bean. The following example is provided as supplemental information only. Suppose a CMP bean type A has a finder method that returns instances of bean A. A read-ahead hint for this method is specified using the following notation: `RelB.RelC; RelD`

Interpret the preceding notation as follows:
- Bean type A has a CMR with bean types B and D.
- Bean type B has a CMR with bean type C.

For each bean of type A that is retrieved from the database, its directly-related B and D beans and its indirectly-related C beans are also retrieved. The order of the retrieved bean data columns in each row of the result set is the same as the order in the read-ahead hint: an A bean, a B bean (or null), a C bean (or null), a D bean (or null). For hints in which the same relationship is mentioned more than once, for example, `RelB.RelC;RelB.RelE`, the data columns for a bean occur only once in the result set, at the position the bean first occupies in the hint.

The tokens shown in the notation, like *RelB*, must be CMR field names for the relationships, as defined in the deployment descriptor for the bean. In indirect relationships such as `RelB.RelC`, *RelC* is a CMR field name that is defined in the deployment descriptor for bean type B.

A single read-ahead hint cannot refer to the same bean type in more than one relationship. For example, if a Department bean has an *employees* relationship with the Employee bean and also has a *manager* relationship with the Employee bean, the read-ahead hint cannot specify both *employees* and *manager*.

For more information about how to set read-ahead hints, see the documentation for the Rational Application Developer product.

Run-time behaviors of read-ahead hints

When developing your read-ahead hints, consider the following tips and limitations:
- Read-ahead hints on long or complex paths can result in a query that is too complex to be useful. Read-ahead hints on root or leaf inheritance mappings need particular care. Add up the number of tables that potentially comprise a read-ahead preload to gauge the complexity of the join operations that are required. Consider if the resulting statement constitutes a reasonable query on your target database.
- Read-ahead hints do not work in the following cases:

- Preload paths across M:N relationships
- Preload paths across recursive enterprise bean relationships or recursive fk relationships
- When a read-head hint applies to a SELECT FOR UPDATE statement that requires a table join in a database that does not support the combination of those two operations.

  Generally, the persistence manager issues a SELECT FOR UPDATE statement for a bean only if the bean has an access intent that enforces strict locking policies. Strict locking policies require SELECT FOR UPDATE statements for database select queries. If the database table design requires a join operation to fulfill the statement, many databases issue exceptions because these databases do not support table joins with SELECT FOR UPDATE statements. In those cases, WebSphere Application Server does not implement a read-ahead hint. If the database does provide that support, Application Server implements the read-ahead hints that you configure.

  DB2 Universal Database V8.2 supports SELECT FOR UPDATE statements with table joins.

Database deadlocks caused by lock upgrades

To avoid databse deadlocks caused by lock upgrades, you can change the access intent policy for entity beans from the default of wsPessimisticUpdate-WeakestLockAtLoad to wsPessimisticUpdate, or you can use an optimistic locking approach.

When concurrently accessing data, ensure that the application is prepared for database locking that must occur to secure the integrity of the data.

If an entity bean performs a findByPrimaryKey method, which by default obtains a *Read* lock in the database, and the entity bean is updated within the same transaction, a lock upgrade to *Exclusive*.

If this scenario occurs concurrently on multiple threads, a deadlock can happen. This is because multiple read locks can be obtained at the same time but one exclusive lock can only be obtained when the other locks are dropped. Since all transactions are attempting the lock upgrade in this scenario, the one exclusive lock cannot be obtained.

To avoid this problem, you can change the access intent policy for the entity bean from the default of wsPessimisticUpdate-WeakestLockAtLoad method to wsPessimisticUpdate method. This change enables the application to inform the product and the database that the transaction has updated the enterprise bean. The *Update* lock is immediately obtained on the findByPrimaryKey method. This avoids the lock upgrade when the update is performed at a later time.

The preferred technique to define access intent policies is to change the access intent for the entire entity bean. You can change the access intent for the findByPrimaryKey method, but this was deprecated in Version 6. You might want to change the access intent for an individual method if, for example, the entity bean is involved in some transactions that are read only.

An alternative technique is to use an optimistic approach, where the findByPrimaryKey method does not hold a read lock, so there is no lock upgrade. However, this requires that the application is coded for this in order to handle rollbacks. Optimistic locking is intended for applications that do not expect database contention on a regular basis.

To change the access intent policy for an entity bean, you can use the assembly tool to set the bean level, as described in “Applying access intent policies to beans” on page 198.

Access intent assembly settings

Access intent policies contain data-access settings for use by the persistence manager. Default access intent policies are configured on the entity bean.

These settings are applicable only for EJB 2.x and EJB 3.0-compliant entity beans that are packaged in EJB 2.x and EJB 3.0-compliant modules. Connection sharing between beans with bean-managed persistence and those with container-managed persistence is possible if they all use the same access intent policy.

Name:

Specifies a name for a mapping between an access intent policy and one or more methods.

Description:

Contains text that describes the mapping.

Methods - name:

Specifies the name of an enterprise bean method, or the asterisk character (*). The asterisk is used to denote all of the methods of an enterprise bean's remote and home interfaces.

Methods - enterprise bean:

Specifies which enterprise bean contains the methods indicated in the Name setting.

Methods - type:

Used to distinguish between a method with the same signature that is defined in both the home and remote interface. Use `Unspecified` if an access intent policy applies to all methods of the bean.

Data type	String
Range	Valid values are `Home`, `Remote`,`Local`, `LocalHome` or `Unspecified`

Methods - parameters:

Contains a list of fully qualified Java type names of the method parameters. This setting is used to identify a single method among multiple methods with an overloaded method name.

Applied access intent:

Specifies how the container must manage data access for persistence. Configurable both as a default access intent for an entity and as part of a method-level access intent policy.

Data type	String
Default	`wsPessimisticUpdate-WeakestLockAtLoad`. With Oracle, this is the same as wsPessimisticUpdate.
Range	Valid settings are `wsPessimisticUpdate`, `wsPessimisticUpdate-NoCollision`, `wsPessimisticUpdate-Exclusive`, `wsPessimisticUpdate-WeakestLockAtLoad`, `wsPessimisticRead`, `wsOptimisticUpdate`, or `wsOptimisticRead`. Only `wsPessimisticRead` and `wsOptimisticRead` are valid when class-level caching is enabled in the EJB container.

This product supports lazy collections. For each segment of a collection, iterating through the collection (*next()*) does not trigger a remote method call to retrieve the next remote reference. Two policies

(wsPessimisticUpdate and wsPessimisticUpdate-Exclusive) are extremely lazy; the collection increment size is set to 1 to avoid overlocking the application. The other policies have a collection increment size of 25.

If an entity is not configured with an access intent policy, the run-time environment typically uses wsPessimisticUpdate-WeakestLockAtLoad by default. If, however, the **Lifetime in cache** property is set on the bean, the default value of **Applied access intent** is wsOptimisticRead; updates are not permitted.

Additional information about valid settings follows:

Profile name	Concurrency control	Access type	Transaction isolation
wsPessimisticRead (Note 1)	pessimistic	read	For Oracle, read committed. Otherwise, repeatable read
wsPessimisticUpdate (Note 2)	pessimistic	update	For Oracle, read committed. Otherwise, repeatable read
wsPessimisticUpdate-Exclusive (Note 3)	pessimistic	update	serializable
wsPessimisticUpdate-NoCollision (Note 4)	pessimistic	update	read committed
wsPessimisticUpdate-WeakestLockAtLoad (Note 5)	pessimistic	update	Repeatable read
wsOptimisticRead	optimistic	read	read committed
wsOptimisticUpdate (Note 6)	optimistic	update	read committed

Notes:
1. Read locks are held for the duration of the transaction.
2. The generated SELECT FOR UPDATE query grabs locks at the beginning of the transaction.
3. SELECT FOR UPDATE is generated; locks are held for the duration of the transaction.
4. A plain SELECT query is generated. No locks are held, but updates are permitted. Use cautiously. This intent enables execution without concurrency control.
5. Where supported by the backend, the generated SELECT query does not include FOR UPDATE; locks are escalated by the persistent store at storage time if updates were made. Otherwise, the same as wsPessimisticUpdate.
6. Generated overqualified-update query forces failure if CMP column values have changed since the beginning of the transaction.

   Be sure to review the rules for forming overqualified-update query predicates. Certain column types (for example, BLOB) are ineligible for inclusion in the overqualified-update query predicate and might affect your design.

Applying access intent policies to beans

You can apply an access intent policy to an application's entity beans through the assembly tool.

About this task

Container-managed persistence (CMP) developers can use *access intent* to provide hints on how the application server run time should manage the details of persistence without having to explicitly manage any of the persistence logic from within their application.

Using the access intent service is also an option for programmers who develop bean-managed persistence (BMP) entity beans. Because the only meaningful difference between BMP and CMP components is the mechanism that provides the persistence logic, BMP beans leverage access intent hints in the same manner as the EJB container manages accent intent for CMP beans. This ability becomes especially

important when BMP entities and CMP entities want to share connections. BMP beans configured with the same concurrency as the CMP beans and implemented to the same isolation level mapping as the CMP can share connections.

Developers can apply access intent policies to BMP entity beans as well as to CMP entity beans. It is expected that BMP developers use only those access intent attributes that are important to a particular BMP bean. The access intent service interface is bound into the *java:comp namespace* for each particular BMP bean. The access intent policy retrieved from the access intent service is current from the time that the *ejbLoad* process is called until the time that the *ejbStore* process completes its invocation.

Note: This is the preferred technique to define access intent policies. Method-level access intent is deprecated in Version 6.0.

1. Start an assembly tool.
2. Optional: Open the Java EE perspective to work with Java EE projects. Click **Window > Open Perspective > Other > Java EE**.
3. Optional: Open the Project Explorer view. Click **Window > Show View > Project Explorer**. Another helpful view is the Navigator view (**Window > Show View > Navigator**).
4. Create a new application EAR file or edit an existing one.

   For example, to change attributes of an existing application, use the import wizard to import an EAR file. To start the import wizard:

   a. Select **File > Import > EAR file > Next**
   b. Select the EAR file.
   c. Create a WebSphere Application Server v6.0 type of Server Runtime. Select **New** to open the New Server Runtime Wizard and follow the instructions.
   d. In the *Target server* field, select *WebSphere Application Server v6.0* type of Server Runtime.
   e. Select **Finish**
5. In the Project Explorer view of the J2EE perspective, right-click **Deployment Descriptor: EJB Module Name** under the EJB module for the bean instance, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the EJB project is displayed in the property pane.
6. Select the **Access** tab.
7. In the **Access Intent for Entities 2.x (Bean Level)** panel, select the name of the bean.
8. On the right side of the **Access Intent for Entities 2.x (Method Level)** panel, select **Add**. The **Add Access Intent** panel displays.
9. In the **Access intent name** field, select wsPessimisticUpdate from the drop-down list.
10. Optional: Enter a **Description** to help you remember what this policy does.
11. Optional: Change the **Persistence Option** setting
12. Click **Finish**. The access intent policy for the entity bean is shown in the **Access Intent for Entities 2.x (Bean Level)** panel

Configuring read-read consistency checking with an assembly tool

Read-read consistency checking only applies to LifeTimeInCache beans whose data is read from another transaction. For the Access Intents that are for *repeatable read* (RR), this means the product checks that the data is consistent with that in the data store, and ensures that no one updates it after the checking.

About this task

For the access intents that are for *read committed* (RC), this means the product checks that the data is consistent at the point of checking, it does **not** guarantee that the data does not change after the checking. This makes the behavior of the LifeTimeInCache bean the same as non-LifeTimeInCache beans.

To perform this checking, you need to configure CMP entity beans with read-read consistency checking. You can do this using an assembly tool. To learn how to complete this task see the topic, ″Adding bean-level access intent for entity beans 2.x″ in the assembly tool documentation at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

Example: Read-read consistency checking

Read-read consistency checking only applies to LifeTimeInCache beans whose data is read from another transaction.

Usage scenario

For the access intents that are for *repeatable read* (RR), this means the product checks that the data is consistent with that in the data store and ensures that no one updates it after the checking. For the Access Intents that are for *read committed* (RC), this means the product checks that the data is consistent at the point of checking, but it does **not** guarantee that the data does not change after the checking. This makes the behavior of the LifeTimeInCache bean the same as non-LifeTimeInCache beans.

You have three options for setting consistency checking, as shown in the following scenarios concerning the calculation of interest in ″Ann's″ bank account. In each case, the data store is shared by this Enterprise JavaBeans (EJB) container managed persistence (CMP) application to calculate the interest and other applications, such as EJB bean managed persistence (BMP) , Java Database Connectivity (JDBC), or legacy applications. Also in each case, the EJB account is configured as a *long-lifetime* bean.

NONE

- The server is started.
- User 1 in Transaction 1 calls Account.findByPrimaryKey(″10001″), account data for Ann is read from the database, with a balance of $100.
- Ann's record is cached by the persistence manager (PM) on the server.
- User 2 writes a JDBC call and changes the balance to $120.
- User 3 in Transaction 2 calls Account.findByPrimaryKey() for account ″10001″, Ann's data is read from cache, with a balance of $100.
- Calculate Ann's interest, but the result might not be correct because of the data integrity issue.

Read-read checking AT_TRAN_BEGIN

- The server is started.
- User 1 in Transaction 1 calls Account.findByPrimaryKey(″10001″), account data for Ann is read from the database, with a balance of $100.
- Ann's record is cached by the persistence manager (PM) on the server.
- User 2 writes a JDBC call and changes the balance to $120.
- User 3 in Transaction 2 calls Account.findByPrimaryKey() for account ″10001″, Ann's data is read from cache, with a balance of $100.
- PM performs read-read check on Ann's account and finds that the balance of 100 is changed. It issues a database query to retrieve balance of $120, and Ann's data in the cache is refreshed.
- Calculate Ann's interest, proceed with the transaction because data integrity is protected.

Read-read checking AT_TRAN_END

- The server is started.
- User 1 in Transaction 1 calls Account.findByPrimaryKey(″10001″), account data for Ann is read from the database, with a balance of $100.
- Ann's record is cached by the persistence manager (PM) on the server.
- User 2 writes a JDBC call and changes the balance to $120.

- User 3 in Transaction 2 calls Account.findByPrimaryKey() for account ″10001″, Ann's data is read from database, with balance of $100.
- Calculate Ann's interest.
- During end of transaction 2, PM performs read-read check on Ann's account and finds that the balance of 100 is changed.
- PM rolls back the transaction and invalidates the cache. The transaction fails and again data integrity is protected.

Access intent service

Access intent is a WebSphere Application Server runtime service that enables you to precisely manage an application's persistence.

The access intent service defines a set of declarative annotations used by the Enterprise JavaBeans (EJB) container and its agents to make performance optimizations for entity bean access. These annotations are organized into sets called *access intent policies*.

Access intent policies contain a set of annotations considered as hints by the EJB container and its agents. Most access intent policies are hints representing high-level abstractions that can be mapped to a specific back end resource manager. It is the responsibility of the EJB persistence machinery to ensure the necessary concurrency control, connection, and cache management when carrying out the persistence details. The EJB persistence manager can use access intent hints to make better performance decisions when carrying out its assigned task. A smaller number of access intents are hints to the EJB container, influencing the management of EJB collections.

Typically, you configure *bean level* access intent for your applications. You can also apply access intent policies to beans within the scope of *application profiles*. Consequently, you can configure beans with multiple and opposing access intent policies. The application profiling documentation explains in more detail how to configure an application to apply a particular access intent policy to a bean for one request, then apply another access intent policy to the same bean for a different request.

Support for applying access intent policies at the method level is deprecated in WebSphere Application Server Version 6.0. In this practice of configuring access intent, you apply a policy to methods within the scope of an EJB module so that the policy becomes the default access intent for all requests upon those methods.

Access intent design considerations

Note: Refrain from over-tuning an application. You can introduce errors by incorrectly using the access intent service. For example, misuse of the wsPessimisticUpdate-NoCollision policy can result in lost updates; inappropriately setting the collection increment value can introduce performance issues; and problem determination is more difficult when an application is configured with multiple access intent policies.

Note: Clarity and simplicity should be your guiding principles when using the access intent service. This is even more important when applying access intent polices within the scope of application profiles.

Even though access intent policies can be configured on any method of an entity bean, some attributes of a policy can only be leveraged by the runtime environment under certain conditions. For example, concurrency and access intent are only used for CMP entity beans when the ejbLoad method is driven to open a connection and read data from a given resource; that data is cached and used to drive the proper queries during invocation of the ejbStore method. Read-ahead hints are only used during the execution of a finder for a bean. The collection increment and resource manager prefetch increment are only used on multi-object finders. Configuring policies on methods that do not use the policy is not an error. Only certain attributes of any policy are used, even when the policy is appropriately applied to a method. However,

configuring policies unnecessarily throughout an application obscures the design of the application and complicates the maintenance of the application.

Access intent with BMP entity beans

Access intent's declarative functionality provides great power to you as a CMP entity bean developer. You can provide hints on how the product should manage the details of persistence without having to explicitly manage any of the persistence logic in the application. There are situations, however, in which you might need to develop BMP entity beans. Since the only meaningful difference between BMP and CMP components is who provides the persistence logic, BMP entity beans should be able to leverage access intent hints just as the product does on behalf of CMP entity beans. BMP entity beans that use the access intent service participate in application profiling; that is, the value of the access intent attributes can differ from request to request, allowing the BMP entity bean to seamlessly modify its persistence strategy.

You can apply access intent policies to BMP entity bean methods as well as CMP entity bean methods. Because access intent hints are not contractual in nature, there is no obligation for a BMP entity bean to exploit them. BMP entity beans are expected to use only those access intent attributes that are important to that particular bean.

The current access intent policy is bound into the java:comp namespace for a particular BMP entity bean. That policy is current only for the duration of the method call during which the access intent policy was retrieved. In a typical scenario, you would cache the access type during invocation of the ejbLoad method so that appropriate actions can be taken during invocation of the ejbStore method.

Access intent best practices

When applying access intent policies to EJB methods, consider the following issues.

- **Start by configuring the default access intent policy for an entity.** After your application is built and started, you can tune certain access paths in your application using application profiling or method-level access intent.
- **Don't mix access types.** Avoid using both pessimistic and optimistic policies in the same transaction. For most databases, pessimistic and optimistic policies use different isolation levels. This can result in multiple database connections, which prevents you from taking advantage of the performance benefits possible through connection sharing.
- **Take care when applying the `wsPessimisticUpdate-NoCollision` policy.** This policy does not ensure data integrity. No database locks are held, so concurrent transactions can overwrite each other's updates. Use this policy only if you can be sure that only one transaction attempts to update persistent store at any time.

For further information on Java Persistence API (JPA) Access intent, see the topic on JPA Access intent.

Applying access intent policies to methods

You apply an access intent policy to a method, or set of methods, in an application's entity beans through the assembly tool.

About this task

Note: Method-level access intent is deprecated in Version 6.0.

1. Start an assembly tool.
2. Optional: Open the Java EE perspective to work with Java EE projects. Click **Window > Open Perspective > Other > Java EE**.
3. Optional: Open the Project Explorer view. Click **Window > Show View > Project Explorer**. Another helpful view is the Navigator view (**Window > Show View > Navigator**).
4. Create a new application EAR file or edit an existing one.

For example, to change attributes of an existing application, use the import wizard to import an EAR file. To start the import wizard:

a. Select **File > Import > EAR file > Next**

b. Select the EAR file.

c. Create a WebSphere Application Server v6.0 type of Server Runtime. Select **New** to open the New Server Runtime Wizard and follow the instructions.

d. In the *Target server* field, select *WebSphere Application Server v6.0* type of Server Runtime.

e. Select **Finish**

5. In the Project Explorer view of the J2EE perspective, right-click the **Deployment Descriptor: EJB Module Name** under the EJB module for the bean instance, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the EJB project is displayed in the property pane.
6. Select the **Access** tab.
7. On the right side of the **Access Intent for Entities 2.x (Method Level)** panel, select **Add**. The **Add Access Intent** panel displays.
8. Specify the **Name** for your new intent policy.
9. Select the **Access intent name** from the drop-down list.
10. Enter a **Description** to help you remember what this policy does.
11. Optional: Select **Read Ahead Hint**. A single access intent read ahead hint might not refer to the same bean type in more than one relationship. For example, if a **Department** enterprise bean has a relationship *employees* with the **Employee** enterprise bean, and also has a relationship *manager* with the **Employee** enterprise bean, then a read ahead hint cannot specify both *employees* and *manager*.
12. Click **Next**. The next **Add Access Intent** panel displays, with optional attributes.
13. Optional: Decide whether or not to overwrite these optional access intent attributes. Click on those you want to change.
14. Click **Next**. The next **Add Access Intent** panel, with a list of Enterprise Beans, displays.
15. Select one or more Enterprise Beans from the list.

    Note: If you selected **Read Ahead Hint** in an earlier step, you can only select **ONE** bean at this step.
16. Click **Next**. The next **Add Access Intent** panel, with a list of methods, displays.
17. Select the methods you want to use.
18. If you *DID NOT* select **Read Ahead Hint** in an earlier step, click **Finish**. If you *DID* select the Read Ahead Hint option, you can click **Next** to specify your Read Ahead Hint for the specified bean. The next **Add Access Intent** panel, with a list of EJB preload paths, displays.
19. Edit the EJB preload path by selecting relationship roles from the **Relationship roles:** window.
20. Click **Finish**. A new entry is created in the **Access Intent for Entities 2.x (Method Level)** panel

Using the AccessIntent API

This task describes how to programmatically retrieve and call the AccessIntent API during the execution of bean managed persistence (BMP) entity bean methods.

1. Look up the access intent service from the namespace. For example:

```
InitialContext ic = new InitialContext();
AccessIntentService aiService = ic.lookup("java:comp/websphere/AppProfile/AccessIntentService");
```

2. From a method of the remote or local component interface of the BMP, get the current AccessIntent object using the javax.ejb.EntityContext. This is passed to the BMP when the container calls the setEntityContext method. Assume the EntityContext was stored in a variable named myEntityCtx. For example:

```
AccessIntent ai = aiService.getAccessIntent (myEntityCtx);
```

3. Use the get() methods of AccessIntent interface to obtain the desired information. For example:

```
int concurrency = ai.getConcurrencyControl();
int accessType = ai.getAccessType();
if ( (concurrency == AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC)
   && (accessType == AccessIntent.ACCESS_TYPE_UPDATE) ) {
     int exclusive = ai.getPessimisticUpdateLockHint();
     // . . .
}
// . . .
```

For a detailed example of the use of the AccessIntent API, see “Example: Using IBM extended APIs to share connections between CMP beans and BMP beans” on page 977.

Results

Note: The access intent object reference retrieved from the java:comp lookup is current for the duration of the method in which the reference was looked up. Depending on how you configured the application profile, subsequent calls of the same method might not retrieve the same access intent reference. You can only look up the object reference during the call of a BMP entity bean's method; the reference does not exist during a request on a container managed persistence (CMP) entity bean. Therefore, access intent object references should not be cached beyond, or used outside of, the scope of the execution of any given BMP method.

AccessIntent interface

The AccessIntent interface is available to bean-managed persistence (BMP) entity beans.

A BMP entity bean can get and use an instance of the AccessIntent interface. For more information see “Using the AccessIntent API” on page 203.

AccessIntent interface

```
package com.ibm.websphere.appprofile.accessintent;

/**
* This interface defines the essential access intents
* available at run time.
*/
public interface AccessIntent {

/**
* Returns the concurrency control intent, which indicates
* the application prefers either pessimistic or optimistic
* concurrency control when accessing the current component
* in the context of the current transaction.
*/
public int getConcurrencyControl();
public final int CONCURRENCY_CONTROL_PESSIMISTIC = 1;
public final int CONCURRENCY_CONTROL_OPTIMISTIC = 2;

/**
* Returns access type intent, which indicates the application
* intends either update or read access of the current component
* in the context of the current transaction.
*/
public int getAccessType();
public final int ACCESS_TYPE_UPDATE= 1;
public final int ACCESS_TYPE_READ = 2;

/**
* Returns an integer value that indicates that the run time should
* assume that there will be no collision on retrieved rows.
*/
public int getPessimisticUpdateLockHint();
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NOCOLLISION = 1;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_WEAKEST_LOCK_AT_LOAD = 2;
```

```
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NONE = 3;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_EXCLUSIVE = 4;

/*
* Returns an integer value that indicates that the run time should
* assume that there will be collisions on retrieved rows.
*/
public int getPessimisticUpdateLockHint();
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NOCOLLISION = 1;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_WEAKEST_LOCK_AT_LOAD = 2;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NONE = 3;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_EXCLUSIVE = 4;


/**
* Returns the collection access intent, which indicates the
* application intends to access the objects returned by the
* currently executing finder in either serial or random fashion.
*/
public int getCollectionAccess();
public final int COLLECTION_ACCESS_RANDOM = 1;
public final int COLLECTION_ACCESS_SERIAL = 2;

/**
* Returns the collection scope, which indicates the maximum
* lifespan of a lazy collection.
*/
public int getCollectionScope();
public final int COLLECTION_SCOPE_TRANSACTION = 1;
public final int COLLECTION_SCOPE_ACTIVITYSESSION = 2;
public final int COLLECTION_SCOPE_TIMEOUT = 3;

/**
* Returns the timeout value in seconds when collectionScope is Timeout.
*/
public int getCollectionTimeout();

/**
* Returns the number of elements the application requests be contained
* in each segment of the element collection returned by the currently
* executing finder.
*/
public int getCollectionIncrement();

/**
* Returns the ReadAheadHint requested by the application for the currently
* executing finder.
*/
public ReadAheadHint getReadAheadHint();

/**
* Returns the number of elements the application requests be contained in
* each segment of a query made on a database.
*/
public int getResourceManagerPreFetchIncrement();

}
```

Access intent exceptions

The exceptions thrown in response to the application of access intent policies are listed.

com.ibm.ws.ejbpersistence.utilpm.PersistenceManagerException

If the method that drives the ejbLoad() method is configured to be read-only but updates are then made within the transaction that loaded the bean's state, an exception is thrown during invocation of the ejbStore() method, and the transaction is rolled back. Likewise, the ejbRemove() method cannot succeed in a transaction that is set as read-only. If an update hint is applied to methods of

entity beans with bean-managed persistence, the same behavior and exception results. The forwarded exception object contains the message string `PMGR1103E: update instance level read only bean beanName`

This exception is also thrown if the applied access intent policy cannot be honored because a finder, ejbSelect, or container-managed relationship (CMR) accessor method returns an inherently read-only result. The forwarded exception object contains the message string `PMGR1001: No such DataAccessSpec - methodName`

The most common occurrence of this error is when a custom finder that contains a read-only EJB Query Language (EJB QL) statement is called with an applied access intent of `wsPessimisticUpdate` or `wsPessimisticUpdate-Exclusive`. These policies require the use of a USE AND KEEP UPDATE LOCKS clause on the SQL SELECT statement to be executed, but a read-only query cannot support USE AND KEEP UPDATE LOCKS. Other examples of read-only queries include joins; the use of ORDER BY, GROUP BY, and DISTINCT keywords.

To eliminate the exception, edit the EJB query so that it does not return an inherently read-only result or change the access intent policy being applied.

- If an update access is required, change the applied access intent setting to `wsPessimisticUpdate-WeakestLockAtLoad` or `wsOptimisticUpdate`.
- If update access is not truly required, use `wsPessimisticRead` or `wsOptimisticRead`.
- If connection sharing between entity beans is required, use `wsPessimisticUpdate-WeakestLockAtLoad` or `wsPessimisticRead`.

com.ibm.websphere.ejb.container.CollectionCannotBeFurtherAccessed

If a lazy collection is driven after it is no longer in scope, and beyond what has already been locally buffered, a CollectionCannotBeFurtherAccessed exception is thrown.

com.ibm.ws.exception.RuntimeWarning

If an application is configured incorrectly, a run-time warning exception is thrown as the application starts; startup is ended. You can validate an application's configuration by choosing the verify function. Some examples of misconfiguration include:

- A method configured with two different access intent policies
- A method configured with an undefined access intent policy

Access intent troubleshooting tips

The following frequently asked questions involving access intent are answered.

Oracle database fails when no access policies are applied

No access intent policies have been applied and the application runs with a DB2 database, but it fails with an Oracle database with the following message:

```
com.ibm.ws.ejbpersistence.utilpm.PersistenceManagerException: PMGR1001E: No such DataAccessSpec
:FindAllCustomers. The backend datastore does not support the SQLStatement needed by this
AccessIntent: (pessimistic update-weakestLockAtLoad)(collections: transaction/25) (resource
manager prefetch: 0) (AccessIntentImpl@d23690a).
```

If you have not configured access intent, all of your data is accessed under the default access intent policy (`wsPessimisticUpdate-WeakestLockAtLoad`). On DB2 the weakest lock is share. On Oracle databases, however, the weakest lock is update; this means that the SQL query must contain a FOR UPDATE clause. To avoid this problem, try to apply an access intent policy that supports optimistic concurrency.

Calling a finder method displays an InconsistentAccessIntentException at run time

This can occur when you use method-level access intent policies to apply more control over how a bean instance is loaded. This exception indicates that the entity bean was previously loaded in the same transaction. This could happen if you called a multifinder method that returned the bean instance with

access intent policy X applied; you are now trying to load the second bean again by calling its findByPrimaryKey method with access intent Y applied. Both methods must have the same access intent policy applied.

Likewise, if the entity was loaded once in the transaction using an access intent policy configured on a finder, you might have called a container-managed relationship (CMR) accessor method that returned the entity bean configured to load using that entity's default access intent.

To avoid this problem, ensure that your code does not load the same bean instance twice within the same transaction with different access intent policies applied. Avoid the use of method-level access intent unless absolutely necessary.

An InconsistentAccessIntentException displays in a container-managed relationship with two beans

There are two beans in a container-managed relationship. The findByPrimaryKey method is called on the first bean and then the getBean2 method is called and a CMR accessor method is called, on the returned instance and an InconsistentAccessIntentException displays.

You are probably using read-ahead. When you loaded the first bean, you caused the second bean to be loaded under the access intent policy applied to the finder method for the first bean. However, you have configured your CMR accessor method from the first bean to the second with a different access intent policy. CMR accessor methods are really finder methods in disguise; the run-time environment behaves as if you were trying to change the access intent for an instance you have already read from persistent store.

To avoid this problem, beans configured in a read-ahead hint are all driven to load with the same access intent policy as the bean to which the read-ahead hint is applied.

A bean with a one-to-many relationship to a second bean displays an UpdateCannotProceedWithIntegrityException error when an instance of the second bean is added to the first bean's collection

A bean with a one-to-many relationship to a second bean displays an UpdateCannotProceedWithIntegrityException error when an instance of the second bean is added to the first bean's collection. The first bean has a pessimistic-update intent policy applied.

The second bean probably has a read intent policy applied. When you add the second bean to the first bean's collection, you are not updating the first bean's state, you are implicitly modifying the second bean's state. The second bean contains a foreign key to the first bean, which is modified.

To avoid this problem, ensure that both ends of the relationship have an update intent policy applied if you expect to change the relationship at run time.

Assembling EJB modules

An enterprise bean is a Java component that can be combined with other resources to create Java Platform Enterprise Edition (Java EE) applications. This topic describes assembling Enterprise JavaBeans (EJB) modules based on the EJB specifications.

Before you begin

This topic assumes that you have created and unit tested an enterprise bean file that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Assemble an EJB module to contain enterprise beans and related code artifacts. Group Web components, client code, and resource adapter code in separate modules. After assembling an EJB module, you can install it as a standalone application or combine it with other modules into an enterprise application.

Use an assembly tool to assemble an EJB module in any of the following ways:
- Import an existing EJB module (EJB JAR file).
- Create a new EJB module.
- Copy code artifacts, such as entity beans, from one EJB module into a new EJB module.

For information on assembling EJB modules, refer to the online documentation or the information center for your assembly tool. The Rational Application Developer product provides supported assembly tools.

- "Assembling EJB 2.x modules"
- "Assembling EJB 3.0 modules" on page 209

What to do next

After you finish assembling your EJB module, you are ready to deploy your module.

Assembling EJB 2.x modules

An enterprise bean is a Java component that can be combined with other resources to create Java EE applications. This topic describes assembling Enterprise JavaBeans (EJB) modules based on the EJB 2.x and earlier specifications.

Before you begin

This topic assumes that you have created and unit tested an enterprise bean (EJB file) that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Assemble an EJB module to contain enterprise beans and related code artifacts. Group Web components, client code, and resource adapter code in separate modules. After assembling an EJB module, you can install it as a standalone application or combine it with other modules into an enterprise application.

Use an assembly tool to assemble an EJB module in any of the following ways:
- Import an existing EJB module (EJB JAR file).
- Create a new EJB module.
- Copy code artifacts (such as entity beans) from one EJB module into a new EJB module.

For information on assembling EJB modules, refer to the online documentation or the information center for your assembly tool at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html.

Results

An EJB module is migrated or created, reflecting the Java EE structure that specifies the location of enterprise bean content files, class files, class paths, the deployment descriptor, and supporting metadata. For more information on the location of the content see the assembly tool information center.

What to do next

After you finish assembling your EJB module, you are ready to deploy your module.

Assembling EJB 3.0 modules

An enterprise bean is a managed Java component that can be combined with other resources to create Java Enterprise Edition (Java EE) applications.

Before you begin

This topic assumes that you have created and unit tested an enterprise bean (EJB file) that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Assemble an EJB 3.0 module to contain enterprise beans and related code artifacts. Group Web components, client code, and resource adapter code in separate modules. After assembling an EJB module, you can install it as a standalone application or combine it with other modules into an enterprise application.

To learn about how to assemble an EJB 3.0 module with an assembly tool see the assembly tool information center.

Rational Application Developer can be extended with additional plug-ins to provide development support specifically for Java Persistence API (JPA).

See the Eclipse open source project, Dali, for a plug-in that provides this extension. See the related links in this topic for the Dali JPA tools Web site.

Note: Issues and problems using this plug-in need to resolved through the Eclipse open source community.

What to do next

After you finish assembling your EJB module, you are ready to deploy your module.

Defining container transactions for EJB modules

Container transaction properties specify how an Enterprise JavaBeans (EJB) container is to manage transaction scopes for the enterprise bean's method invocations.

About this task

A transaction attribute is mapped to one or more methods. Some container transaction settings are not available for all enterprise beans. Also, some methods are not available for particular transaction settings and beans. These rules have been implemented in the Add Container Transaction wizard based on the EJB specification.

To complete this task see the topic, Defining container transactions for EJB modules, in the assembly tool information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

Related concepts

“EJB containers” on page 125
An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean’s operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

Related tasks

“Assembling EJB modules” on page 207
An enterprise bean is a Java component that can be combined with other resources to create Java Platform Enterprise Edition (Java EE) applications. This topic describes assembling Enterprise JavaBeans (EJB) modules based on the EJB specifications.

Defining container transactions for EJB modules

Container transaction properties specify how an Enterprise JavaBeans (EJB) container is to manage transaction scopes for the enterprise bean’s method invocations.

About this task

A transaction attribute is mapped to one or more methods. Some container transaction settings are not available for all enterprise beans. Also, some methods are not available for particular transaction settings and beans. These rules have been implemented in the Add Container Transaction wizard based on the EJB specification.

To complete this task see the topic, Defining container transactions for EJB modules, in the assembly tool information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

Related concepts

“EJB containers” on page 125
An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean’s operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

Related tasks

“Assembling EJB modules” on page 207
An enterprise bean is a Java component that can be combined with other resources to create Java Platform Enterprise Edition (Java EE) applications. This topic describes assembling Enterprise JavaBeans (EJB) modules based on the EJB specifications.

References

References are logical names used to locate external resources for enterprise applications. References are defined in the application’s deployment descriptor file. At deployment, the references are bound to the physical location (global JNDI name) of the resource in the target operational environment.

This product supports the following types of references:
- An EJB reference is a logical name used to locate the home interface of an enterprise bean.
- A resource reference is a logical name used to locate a connection factory object.

These objects define connections to external resources such as databases and messaging systems. The container makes references available in a JNDI naming subcontext. By convention, references are organized as follows:
- EJB references are made available in the java:comp/env/ejb subcontext.
- Resource references are made available as follows:
  - JDBC data source references are declared in the java:comp/env/jdbc subcontext.
  - JMS connection factories are declared in the java:comp/env/jms subcontext.

– Mail connection factories are declared in the java:comp/env/mail subcontext.
– URL connection factories are declared in the java:comp/env/url subcontext.

EJB references

Use this page to view and modify the Enterprise JavaBeans (EJB) references to the enterprise beans. References are logical names used to locate external resources for enterprise applications. References are defined in the application's deployment descriptor file. At deployment, the references are bound to the physical location (global Java Naming and Directory Interface (JNDI) name) of the resource in the target operational environment.

If your application defines EJB references, for **Map EJB references to beans**, specify JNDI names for enterprise beans that represent the logical names that are specified in EJB references. Each EJB reference defined in the application must be bound to an EJB file before clicking Finish in the Summary panel.

If the EJB reference is from an EJB 3.0, Web 2.4, or Web 2.5 module, the JNDI name is optional. If the **Allow EJB reference targets to resolved automatically** option is enabled, the JNDI name is optional for all modules. The runtime provides a container default or automatically resolves the EJB reference if a binding is not provided.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → **application_name** → **EJB references**.

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Module

Specifies the name of the Enterprise JavaBeans module used by your application.

EJB

Specifies the name of an enterprise bean that is contained by the module.

URI

Specifies location of the module relative to the root of the application EAR file.

Resource Reference

Specifies the name of the EJB reference that is used in the enterprise bean, if applicable, and declared in the deployment descriptor of the application module.

Class

Specifies the name of a Java class associated with this enterprise bean.

Target Resource JNDI Name

Specifies the JNDI name of the enterprise bean.

This is a data entry field. To modify the JNDI name bound to this bean, type the new name in this field, then select **OK**.

Data type String

EJB JNDI names for beans

Use this page to view and modify the Java Naming and Directory Interface (JNDI) names of non-message-driven enterprise beans in your application or module.

If your application uses EJB 2.1 and earlier modules, on the Provide JNDI names for beans panel, specify a JNDI name for each enterprise bean in every EJB 2.1 and earlier module. You must specify a JNDI name for every EJB 2.1 and earlier enterprise bean defined in the application. For example, for the EJB module MyBean.jar, specify MyBean.

The JNDI name for an EJB module can be used for both EJB 3.0 modules and pre-EJB 3.0 modules. For a pre-EJB 3.0 module, you need to provide a JNDI name for the bean. For an EJB 3.0 module, you have three options

- Provide no JNDI names at all
- Select the radio button to provide a JNDI name for the bean, or
- Select the radio button to provide local or remote home JNDI names.

If no JNDI name is provided, the runtime provides a default value. If JNDI name for the bean is provided, you cannot provide any JNDI name for business interface in the Provide JNDI names for business interfaces panel.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → **application** → **EJB JNDI names**.

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

EJB module

Specifies the name of the Enterprise JavaBeans module used by your application.

EJB

Specifies the name of an enterprise bean that is contained by the module.

URI

The Uniform Resource Identifier (URI) specifies the location of the module archive relative to the root of the application EAR.

JNDI name

Specifies the Java Naming and Directory Interface (JNDI) name of the enterprise bean.

This is a data entry field. To modify the JNDI name bound to this bean, type the new name in this field, then select **OK**.

Data type String

Bind EJB business

Use this administrative console panel to specify Java Naming and Directory (JNDI) name bindings for each enterprise bean with a business interface in an EJB module. Each enterprise bean with a business interface in an EJB module must be bound to a JNDI name. For any business interface that does not provide a JNDI name, or if its bean does not provide a JNDI name, a default binding name is provided. If its bean provides a JNDI name, the default JNDI name for the business interface is provided on top of its bean JNDI name by appending the package-qualifed class name of the interface.

If you specify the JNDI name for a bean in the Provide JNDI names for beans panel, do not specify any business interface JNDI name in this panel for the same bean . If you do not specify the JNDI name for a bean in the Provide JNDI names for beans panel, you can optionally specify a business interface JNDI name. If you do not specify a business interface JNDI name, the runtime provides a container default.

To view this panel in the application installation wizard, click **Applications** → **New applications** → **New Enterprise Application** → **application_path** → **Next** → **Detailed - Show all installation options and parameters** → **Next** → **Step: Bind EJB business**.

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

EJB module
Specifies the EJB module that contains the enterprise beans that bind to the JNDI name.

EJB
Specifies the enterprise bean that binds to the JNDI name.

URI
The Uniform Resource Identifier (URI) specifies the location of the module archive relative to the root of the application EAR.

Business interface
Specifies the enterprise bean business interface in an EJB module.

JNDI name
Specifies the JNDI name associated with the enterprise bean business interface in an EJB module.

Sequence grouping for container-managed persistence

After assembling an Enterprise JavaBeans (EJB) module that contains container-managed persistence (CMP) beans, you can prevent certain types of database-related exceptions from occurring during application run time. Using *sequence grouping*, you can specify the order in which entity beans update relational database tables.

Note: Entity beans are not supported in EJB 3.0 modules.

Eliminate exceptions resulting from referential integrity (RI) violations

Sequence grouping is particularly useful for preventing violations of database *referential integrity* (RI). A database RI policy prescribes rules for how data is written to and deleted from the database tables to maintain relational consistency. Run-time requirements for managing bean persistence, however, can cause an EJB application to violate RI rules, which can cause database exceptions. These run-time requirements mandate that:

- Entity bean create and remove operations correlate to the database immediately upon method invocation.
- Entity bean changes are cached by the EJB container until either a finder method is called, or the transaction ends.

Consequently, the order in which entity beans update the database is unpredictable. That randomness translates into high risk of the application violating database RI. Although caching the operations for batch processing overrides these run-time requirements, it does not guarantee a bean persistence sequence that follows any given RI policy.

The only way to guarantee a persistence sequence that honors database RI is to designate the sequence, which you do in the EJB deployment descriptor editor of the assembly tool. Through the sequence grouping feature, you assign beans to CMP groups. Within each group you specify the order in which the persistence manager inserts bean data into the database to accomplish updates without violating RI.

See the “Setting the run time for CMP sequence groups” topic for detailed instructions on designating sequence groups. Consult your database administrator about the RI policy with which you need to synchronize.

Minimize exception risk for optimistic concurrency control schemes

Sequence grouping can also reduce the risk of transaction rollback exceptions for entity beans that are configured for optimistic concurrency control. In these concurrency control schemes, database locks are held for minimal amounts of time so that a maximum number of transactions consistently have access to the data. The relatively unrestricted state of the database can lead to transaction rollback exceptions for two common reasons:

- When concurrent transactions attempt to lock the same table row, database deadlock occurs.
- Transactions can occur in an order that violates application logic.

Use the sequence grouping feature to order bean persistence so that these scenarios are less likely to occur.

Setting the run time for CMP sequence groups

By designating container managed persistence (CMP) sequence groups for entity beans, you can prevent certain types of database-related exceptions from occurring during the run time of your Enterprise JavaBeans (EJB) application. Within each group you need to specify the order in which the beans update your relational database tables.

Before you begin

When you define a sequence group, you designate it as one of two types:

- RI_INSERT, for setting a bean persistence sequence to prevent database referential integrity (RI) violations
- UPDATE_LOCK, for setting a bean persistence sequence to minimize exceptions resulting from optimistic concurrency control

About this task

Both types of sequence groups must be created after you have assembled the beans into an EJB module, prior to installing your application on the product. If you need to edit sequence groups, you must uninstall the application, make your changes using the following steps as a guide, and then reinstall your application.

Note: If you already selected or plan to use top-down mapping for mapping your enterprise beans to back end data, you do not need to create a sequence group with an RI_INSERT type. The product does not generate an RI policy for the database schema that it creates when you select top-down mapping.

To learn how to complete this task see the assembly tool information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

What to do next

You are now ready to deploy your EJB module or combine it with other modules into a J2EE application.

Deploying EJB modules

When you deploy an Enterprise JavaBeans (EJB) module, you install that module on a server that has been configured to support deployed modules.

Before you begin

Assemble one or more EJB modules, assemble one or more Web modules, and assemble them into a J2EE application.

Assemble one or more EJB 3.0 modules, assemble one or more Web modules, and assemble them into a J2EE application.

For an overview about the changes to the EJB deployment model for EJB 3.0, see the topic ″EJB 3.0 deployment overview.″

1. Prepare the deployment environment.
2. Update the configuration for each EJB module as needed for the deployment environment.
3. Deploy the application.

What to do next

If you specify that the EJBDeploy tool be run during application installation and the installation fails with a NameNotFoundException message, ensure that the input Java archive (JAR) or enterprise archive (EAR) file does not contain source files. Either remove the source files or include all dependent classes and resource files on the class path. If there are source files in the input JAR or EAR file, the EJB deployment tools runs a rebuild before generating the deployment code.

If the module deploys successfully, test and debug the module.

EJB 3.0 deployment overview

Learn about the Enterprise JavaBeans (EJB) 3.0 deployment model, including to *Just-In-Time* (JIT) deployment.

All Java Platform, Enterprise Edition (Java EE) application server products have some form of EJB deployment phase where your application is customized to run in that particular application server implementation. Typically, this is accomplished by an application server-specific deployment tool, which generates code to bridge your EJB interface and implementation code to the application server's EJB container implementation. Some application server products' deploy tools alter the bytecodes of your application classes rather than using code generation, but the end result is similar.

In WebSphere Application Server, the bridging is accomplished by generating code that *wraps* your EJB implementation classes, connecting them to the product EJB container, which in turn allows the EJB container to host your enterprise beans and provide services to them. If one or more of your enterprise beans has defined remote interfaces, additional code is generated to provide the remote function.

Historically, EJB deployment in the WebSphere product has been performed by the *EJBDeploy* tool that is included with product and packaged with WebSphere product-oriented development tools.

The EJBDeploy tool introspects your EJB external interfaces, generates the wrapper code as .java files, then compiles it using the javac compiler to produce .class files, which are then packaged in your EJB module with your application code. The EJBDeploy tool also runs the rmic tool against the remote EJB interfaces in the application, producing additional *stub* and *tie* class files that interact with the Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) Object Request Broker (ORB), providing

remote object support. Typically, you run the EJBDeploy tool either when you install the application on the product or sometime before you install the application from the command-line tool or within a development tool.

Just-In-Time deployment

The EJB 3.0 support in WebSphere Application Server adds a new feature called Just-In-Time Deployment. With Just-In-Time Deployment, the EJB container dynamically generates the wrapper, stub, and tie classes in-memory as needed when the application is running. Additionally, the Web container and application client containers dynamically generate the stub class required for remote EJB invocations. Effectively, this means that you do not need to process EJB 3.0 modules, Web modules that invoke EJB 3.0 beans, or client modules that invoke EJB 3.0 beans, through the EJBDeploy tool prior to running them in WebSphere.

createEJBStubs tool

Even though the Just-In-Time Deployment feature, in many cases, dynamically generates the RMI-IIOP stub classes that are required for invocation of remote EJB interfaces, there remain some cases where these stub classes are not dynamically generated. For EJB 3.0 clients not running inside a Web container, EJB container, or client container, that is upgraded to EJB 3.0 level, you must generate the stub classes with the createEJBStubs tool, then make the generated stubs available in the client environment's classpath. Typically you would accomplish this by copying the generated stubs to the location where the client's business interface class resides.

To summarize, the createEJBStubs tool must be used to generate client-side stubs for the following environments:

- ″Bare″ Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.
- WebSphere Application Server container environments prior to Version 7 that do not have the Feature Pack for EJB 3.0 applied.
- Non-WebSphere Application Server environments.

For more information about packaging your EJB module, see the topic, ″EJB 3.0 module packaging overview.″

EJBDEPLOY relationships – troubleshooting tips

Use this information to troubleshoot information for EJBDEPLOY problems.

DB2 for z/OS Version 7.x

Problems might exist when EJBDeploy creates a data relationship in DB2 for z/OS Version 7.x. EJBDeploy creates a table with a composite of the two primary keys of the EJBs that are related to each other. If the composite keys are larger than 254 characters, DB2 for z/OS V7.x does not accept this relationship and the following error can occur:

```
DSNT408I SQLCODE = -613, ERROR:  THE PRIMARY KEY OR A UNIQUE CONSTRAINT
IS TOO LONG OR HAS TOO MANY COLUMNS
DSNT418I SQLSTATE   = 54008 SQLSTATE RETURN CODE
```

This problem can be seen when the primary keys that are created for the two related beans have primary keys that are strings. This results in the composite being made up of 2 varchar(250) primary keys for a total of 500, which is greater than 254 maximum in DB2 for z/OS version 7.x.

Things to consider when utilizing top-down mappings to ensure you do not experience this problem:

- Top-down mappings are a guideline and must be reviewed with the DBA.

- Schemas that are created top-down by EJBDeploy are designed only for testing, and as a guideline for the actual schema required. The use of the meet-in-the-middle mapping does not present this problem.
- The composite key constraint problem is not experienced when using DB2 V8, which has 2K maximum key lengths.

EJBDEPLOY_JVM_OPTIONS

Set the `EJBDEPLOY_JVM_OPTIONS` property to override Java virtual machine (JVM) options that are passed to the code that deploys Enterprise JavaBeans (EJBs) (ejbdeploy.sh). Set this property in one of the following locations: *deploymentmanager*/bin/setupCmdLine.sh or *appServerHome*/bin/setupCmdLine.sh

For example, the following command increases the heap size of the JVM for ejbdeploy.sh:

```
export EJBDEPLOY_JVM_OPTIONS="-Xms128m -Xmx512m"
```

The converter that is defined for the primary key is not invoked on its foreign key value

The mapping for primary key fields to database columns may use a converter to transform the key values. If a container-managed persistence (CMP) bean uses a converter to map its primary key, and that bean has a relationship where the bean at the other end holds a foreign key, the mapping for the foreign key will not use the converter.

The following errors might occur, indicating that the converter defined for the primary key is not invoked on its foreign key value. During the run of the ejbDeploy command , you receive the following message:

```
No type mapping defined for Java datatype1 to Database datatype2
```

During run time, the application does not find the CMP bean at the other end of the relationship.

To work around this limitation, define your own foreign key in the database table, and create a mapping that uses the same converter as defined for the primary key on the enterprise beans at the other end of its relationship.

EJB module settings

Use this page to configure and manage a specific deployed EJB module.

Note: You cannot start or stop an individual EJB module for modification. You must start or stop the appropriate application entirely.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → **application_name** → **Manage Modules** → **module_name**.

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

URI

Specifies location of the module relative to the root of the application EAR file. The URI must match the URI of a ModuleRef URI in the deployment descriptor of the deployed application (EAR).

Alternate deployment descriptor

Specifies an alternate deployment descriptor for the module as defined in the application deployment descriptor according to the J2EE specification.

Starting weight

Specifies the order in which modules are started when the server starts. The module with the lowest starting weight is started first.

Data type	Integer
Default	5000
Range	Greater than 0

Task overview: Storing and retrieving persistent data with the Java Persistence API (JPA)

The Java Persistence API (JPA) for the application server defines the management of persistence and object/relational mapping within Java Enterprise Edition (Java EE) and Java Standard Edition (Java SE) environments.

About this task

The Java Persistence API (JPA) represents a simplification of the persistence programming model. JPA functions within the Java EE specification for Enterprise Java Beans (EJB) 3.0 requirements, managing persistence and object/relational mapping. The JPA specification defines the object/relational mapping within its own guidelines instead of relying on vendor-specific mapping implementations. These features make applications that use JPA easier to implement and manage.

In a nutshell, JPA combines the best features from previous persistence mechanisms such as Java Database Connectivity (JDBC) APIs, Object Relational Mapping (ORM) frameworks, and Java Data Objects (JDO). Creating entities under JPA is as simple as Plain Old Java Objects (POJOs). JPA supports the features provided by JDBC without requiring the knowledge of the specific programming models defined by the various JDBC implementations. Like object-relational software and object databases, JPA allows the use of advanced object-oriented concepts such as inheritance. JPA avoids vendor lock-in because it does not rely on a strict specification like JDO and EJB 2.x entities.

The JPA implementation does not mandate that you migrate existing applications. Existing EJB 2.x Container Managed Persistence applications continue to execute without changes. JPA may not be ideal for every application, however, for many applications it provides a better alternative to other persistence implementations.

Use the topics listed below for detailed information about aspects of JPA:

- Learn about **persistence and JPA**
- Learn about the differences in **Apache OpenJPA and JPA for WebSphere Application Server**
- Find information on **developing JPA applications for a Java EE environment** or **developing JPA applications for a Java SE environment**
- A guide to **configuring persistence providers**
- Set up a datasource by **configuring JDBC for use with JPA for WebSphere Application Server**
- Learn about **Configuring caching to improve performance**
- Monitor your applications by **logging your application's behavior with JPA for WebSphere Application Server**
- Troubleshoot JPA problems: **Troubleshooting JPA**
- Learn about **JPA Access Intent**

What to do next

Product support for JPA

The implementation of Java Persistence API for the application server can be used on all platforms that are supported for the application server, including iSeries® and z/OS. Java Persistence API for the application server functions with all databases supported in WebSphere Application Server. In addition to these, Java Persistence API for WebSphere Application Server can support databases that are supported by the OpenJPA implementation of JPA.

Note: Databases supported by OpenJPA but not supported by the product have not been tested extensively by IBM and might contain unknown compatibility issues. For a list of supported databases, refer to the OpenJPA user guide.

Additional information

For additional information about OpenJPA, see the OpenJPA User Guide. For information about Java Persistence API specifications, see the link listed below. The information resides on both IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

- For information on the Java Persistence API specification, consult: http://java.sun.com/javaee/technologies/persistence.jsp.

Related tasks

Configuring Persistence Provider support in the application server
Persistence providers are implementations of the Java Persistence API (JPA) specification and can be deployed in the Java EE compliant application server that supports JPA persistence.

Associating persistence units and data sources
Java Persistence API (JPA) applications allow you to specify the underlying data source used by the persistence provider to access the database.

Configuring OpenJPA caching to improve performance
The OpenJPA implementation allows users the option of storing frequently used data in the memory to improve performance. OpenJPA provides concurrent data and concurrent query caches that allow applications to save persistent object data and query results in memory to share among threads and for use in future queries.

Troubleshooting Java Persistence API (JPA) applications
Use this information to find various known problems with JPA applications.

Related information

../../../../com.ibm.websphere.wim.doc.en/supporteddatabases.html

Java Persistence API (JPA) Architecture

Data persistence, the ability to maintain data between application executions, is vital to enterprise applications because the required access to relational databases. Applications that are developed for this environment must manage persistence themselves or make use of third-party solutions to handle database updates and retrievals with persistence. The Java Persistence API (JPA) provides a mechanism for managing persistence and object-relational mapping and functions for the EJB 3.0 specifications.

The JPA specification defines the object-relational mapping internally, rather than relying on vendor-specific mapping implementations. JPA is based on the Java programming model that applies to Java EE environments, but JPA can function within a Java SE environment for testing application functions.

JPA represents a simplification of the persistence programming model. The JPA specification explicitly defines the object-relational mapping, rather than relying on vendor-specific mapping implementations. JPA standardizes the important task of object-relational mapping by using annotations or XML to map objects into one or more tables of a database. To further simplify the persistence programming model:

- The EntityManager API can persist, update, retrieve, or remove objects from a database

- The EntityManager API and object-relational mapping meta-data handle most of the database operations without requiring you to write JDBC or SQL code to maintain persistence
- JPA provides a query language, extending the independent EJB querying language (also known as JPQL), that you can use to retrieve objects without writing SQL queries specific to the database you are working with.

JPA is designed to operate both inside and outside of a Java Enterprise Edition (Java EE) container. When you run JPA inside a container, the applications can use the container to manage the persistence context. If there is no container to manage JPA, the application must handle the persistence context management itself. Applications that are designed for container-managed persistence do not require as much code implementation to handle persistence, but these applications cannot be used outside of a container. Applications that manage their own persistence can function in a container environment or a Java SE environment.

Elements of a JPA Persistence Provider

Java EE containers that support the EJB 3.0 programming model must support a JPA implementation, also called a persistence provider. A JPA persistence provider uses the following elements to allow for easier persistence management in an EJB 3.0 environment:

- **Persistence unit:** consists of the declarative meta-data that describes the relationship of entity class objects to a relational database. The EntityManagerFactory uses this data to create a persistence context that can be accessed through the EntityManager.
- **EntityManagerFactory:** used to create an EntityManager for database interactions. The application server containers typically supply this function, but the EntityManagerFactory is required if you are using JPA application-managed persistence.
- **Persistence context:** defines the set of active instances that the application is manipulating currently. The persistence context can be created manually or through injection.
- **EntityManager:** the resource manager that maintains the active collection of entity objects that are being used by the application. The EntityManager handles the database interaction and meta-data for object-relational mappings. An instance of an EntityManager represents a persistence context. An application in a container can obtain the EntityManager through injection into the application or by looking it up in the Java component name-space. If the application manages its persistence, the EntityManager is obtained from the EntityManagerFactory.

  Note: Injection of the EntityManager is only supported for the following artifacts:
  - EJB 3.0 session beans
  - EJB 3.0 message-driven beans
  - Servlets, Servlet Filters, and Listeners
  - JSP tag handlers which implement interfaces javax.servlet. jsp.tagext.Tag and javax.servlet.jsp.tagext.SimpleTag
  - Java Server Faces (JSF) managed beans
  - the main class of the application client.
- **Entity objects:** a simple Java class that represents a row in a database table in its simplest form. Entities objects can be concrete classes or abstract classes. They maintain states by using properties or fields.

For more information about persistence, see the section on Java Persistence API Architecture and the section on Persistence in the Apache OpenJPA User's Guide. For more information and examples on specific elements of persistence, refer to the sections on the EntityManagerFactory, and the EntityManager in the Apache OpenJPA User's Guide.

JPA for WebSphere Application Server

JPA for WebSphere Application Server is built on the Apache OpenJPA open source project.

Apache OpenJPA and JPA for WebSphere Application Server

Apache OpenJPA is a compliant implementation of the Sun Microsystems JPA specification. Using OpenJPA as a base implementation, WebSphere Application Server employs extensions to provide additional features and utilities for WebSphere Application Server customers. Because JPA for WebSphere Application Server is built from OpenJPA, all OpenJPA functionality, extensions and configurations are unaffected by the WebSphere Application Server extensions. Users running OpenJPA applications do not need to make any changes to use their applications in WebSphere Application Server.

JPA for WebSphere Application Server provides more than compatibility with OpenJPA. JPA for WebSphere Application Server contains a set of tools for application development and deployment. Other features of JPA for WebSphere Application Server include support for XML mapping, JPA Access Intent, enhanced tracing capabilities, command scripts and translated message files. The provider of JPA for WebSphere Application Server is `com.ibm.websphere.persistence.PersistenceProviderImpl`.

The properties for OpenJPA can be defined in one of two ways. You can either specify the property in the `persistence.xml` file or by using a Java virtual machine (JVM) command line argument on either client or server. See the following examples:

- Specify the OpenJPA property in the `persistence.xml` file.

```
<properties>
 <property name="openjpa.jdbc.SchemaFactory" value="native(ForeignKeys=true)" />
</properties>
```

- Specify the OpenJPA property using a JVM command line argument on the client or server.

```
-Dopenjpa.jdbc.SchemaFactory="native(ForeignKeys=true)"
```

For more information on OpenJPA properties, see Chapter 2 on Configuration and the Part 3 Reference sections in the OpenJPA users guide.

The Extension Properties of JPA for WebSphere Application Server may be specified with the openjpa or wsjpa prefix. You can mix the openjpa and wsjpa prefixes as you wish for a common set of properties. Exceptions to the rule are wsjpa specific configuration properties, which should use the wsjpa prefix only. In the event that a JPA for WebSphere Application Server specific property is used with the openjpa prefix, a warning message will be logged indicating that the offending property will be treated as a wsjpa property. The reverse does not hold true for the openjpa prefix. In that case, the offending property will merely be ignored.

Developing and packaging JPA applications for a Java EE environment

Containers in the application server can provide many of the necessary functions for the Java Persistence API (JPA) in a Java Enterprise Edition (Java EE) environment. The application server also provides JPA tools to assist you with developing applications in a Java EE environment.

About this task

JPA applications require different configuration techniques from applications that use container-managed persistence (CMP) or bean-managed persistence (BMP). They do not follow the typical deployment techniques that are associated with applications that implement CMP or BMP. In JPA applications, you must define a persistence unit and configure the appropriate properties to ensure that the applications can run in a Java EE environment.

The container supports all necessary injections to ensure that applications run in the Java EE environment. For example, the container can inject the @PersistenceUnit and @PersistenceContext for your applications.

1. Generate your entities classes. Depending upon your development model, you might use some or all of the JPA tools:
   - **Top-down mapping:** You start from scratch with the entity definitions and the object-relational mappings, and then you derive the database schemas from that data. If you use this approach, you are most likely concerned with creating the architecture of your object model and then writing your entity classes. These entity classes would eventually drive the creation of your database model. If you are using a top-down mapping of the object model to the relational model, develop the entity classes and then use OpenJPA functionality to generate the database tables that are based on the entity classes. The wsmapping tool would help with this approach.
   - **Bottom-up mapping:** You start with your data model, which are the database schemas, and then you work upwards to your entity classes. The wsreversemapping tool would help with this approach.
   - **Meet in the middle mapping:** probably the most common development model. You have a combination of the data model and the object model partially complete. Depending on the goals and requirements, you will need to negotiate the relationships to resolve any differences. Both the wsmapping tool and the wsreversemapping tool would help with this approach.

   The JPA solution for the application server provides several tools that help with developing JPA applications. Combining these tools with IBM Rational Application Developer provides a solid development environment for either Java EE or Java SE applications. Rational Application Developer includes GUI tools to insert annotations, a customized `persistence.xml` file editor, a database explorer, and other features. Another alternative is the Eclipse Dali project. More information on Rational Application Developer or the Eclipse Dali plugin can be found at their respective web sites.
2. Enhance the entity classes using the JPA enhancer tool, wsenhancer, for the application server. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. The enhancer post-processes the bytecode that is generated by the Java compiler and adds the fields and methods that are necessary to implement the persistence features. To use the wsenhancer tool, type the following at a command prompt:

   ```
   ${app_server_root}/bin/wsenhancer.sh [parameters][arguments]
   ```

   Although JPA for the application server and OpenJPA can automatically enhance the entities at run time, you will obtain better performance if you can enhance your entities when you build the application. The application will not attempt to enhance entities that are already enhanced.
3. Optional: If you are not using the development model for bottom-up mapping, generate or update your database tables automatically or by using the wsmapping tool.
   - By default, the object-relational mapping does not occur automatically, but you can configure the application server to provide that mapping with the openjpa.jdbc.SynchronizeMappings property. This property can accelerate development by automatically ensuring that the database tables match the object model. To enable automatic mapping, include the following line in the `persistence.xml` file:

     ```
     <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
     ```

     Note: To enable automatic object-relational mapping at run time, all of your persistent classes must be listed in the Java .class file, mapping file, and Java archive (JAR) file elements in XML format.
   - To manually update or generate your database tables, run the JPA mapping tool for the application server from the command line to create the tables in the database. For example:

     ```
     ${app_server_root}/bin/wsmapping.sh [parameters][arguments]
     ```
4. Optional: If you are using DB2 and want to use static SQL, run the wsdb2gen command. In order to use the wsdb2gen tool, pureQuery runtime must be installed. The wsdb2gen tool creates *persistence_unit_name*.pdqxml file under the same META-INF directory where your `persistence.xml` file is located. If you have multiple persistence units, wsdb2gen command must be run for each persistence unit. To use the wsdb2gen tool, type the following at a command prompt:

   ```
   ${app_server_root}/bin/wsdb2gen.sh [parameters]
   ```
5. Configure these properties.

a. Specify the configuration options for your database. The application server manages access to data sources. You can configure the data sources, connection pooling, and JTA transaction service in the administrative console. If you have a specific data source for your application, configure the data source before you install your JPA application.
   1) Configure your data sources through the administrative console. See the topic on configuring the JDBC provider and data source for more information.
   2) Specify the Java Naming and Directory Interface (JNDI) names for the <jta-data-source> and <non-jta-data-source> elements. If you use the component name space method for data source retrieval, ensure that your application defines these resource references so that you can use these JNDI names to access the data source. This configuration provides more flexibility if you need to alter the configuration for the data source. For more information on using the JNDI interface, refer to the topic on developing applications that use JNDI. For example, the `persistence.xml` file would have a entry like the following:

      ```
      <jta-data-source>java:comp/env/jdbc/FooBarDataSourceJNDI</jta-data-source>
      ```

b. If you are using pureQuery, configure your data source to use pureQuery. Ensure the `pdq.jar` and `pdqmgmt.jar` files are included on the JDBC provider class path. The JPA provider implementation must be the IBM implementation of JPA provider of `com.ibm.websphere.persistence.PersistenceProviderImpl`. The OpenJPA persistence provider does not provide support for pureQuery. For more information, refer to the topic configuring a data source to use pureQuery.

6. Package the application. There are several packaging options for an application that uses JPA in a Java EE environment. Choose the packaging option that best suits the JPA usage and configuration within the modules of your application. These are some of the most common packaging options. For a definitive list of packaging options, see the Java Persistence API specification.

   Note: If you are using pureQuery, add the *persistence_unit_name*`.pdqxml` files created previous or the or *persistence_unit_name*`.pdqxml` files created in the above Step 4 to the JPA application JAR file. The files are located in same META-INF directory where your `persistence.xml` file is located.

   - For a standalone EJB module or a standalone application client module, package the EJB and application client modules in a standard JAR file. Ensure that you package the application with these conditions:
     – The JAR file must contain your EJB class files or the Java class files for the application client.
     – The META-INF directory of the archive must include your `persistence.xml` file.
     – If your application uses mapping files, orm.xml, or a custom mapping file, the JAR file must contain those files as well. If the location of the orm.xml file is not specified in the persistence unit, the default location is the META-INF directory of the JAR file.
   - For a standalone web module, package the application in a standard Web Application archive (WAR). Ensure that you package the application with these conditions:
     – The WAR file must contain your web application class files. The web application class files must be included in the WEB-INF/classes directory or in a JAR file that is located in the WEB-INF/lib directory of the WAR file.
     – Your `persistence.xml` file must be included in the WEB-INF/classes/META-INF directory or in the META-INF directory of a JAR file that is included in your WEB-INF/lib directory of your WAR file.
     – If your application uses mapping files, orm.xml, or a custom mapping file, the WAR file must also contain those files. Mapping files can reside in the WEB-INF/classes directory or in a JAR file that is contained within the WEB-INF/lib directory of the WAR file. Use the <mapping-file> element of the `persistence.xml` file to specify the location of mapping files. For example:

       ```
       <mapping-file>META-INF/JPAorm.xml</mapping-file>
       ```

- For enterprise application that contains one or more modules, package the application in a standard Enterprise Application archive (EAR). An enterprise application can contain one ore more EJB module, web module, or application client module. Ensure that you package the application with these conditions:
  - If multiple modules use the same persistence unit, you can create a persistence archive and package the persistence archive within your EAR file.
  - Include your entity classes, any necessary supporting classes, your `persistence.xml` file, and additional mapping files in the persistence archive file. Follow the packaging rules for EJB and application client modules for the location of your `persistence.xml` file and mapping files.
  - Each module that uses the persistence archive must have a class path entry in its META-INF/MANIFEST.MF file. Here is an example manifest file:

    ```
    Manifest-Version: 1.0
    Class-Path: MyJPAEntities.jar
    ```

  - If your modules use separate persistence units and share entity classes, you can package the entity classes in a persistence archive and specify different `persistence.xml` file and mapping files for each module. If the modules do not share entity classes or a persistence configuration, package each module as a standalone EJB module, a standalone application client module, or a standalone web archive and then package them in the EAR file.

Example

This is a sample `persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="TheWildZooPU" transaction-type="JTA">
        <jta-data-source>jdbc/FooBarDataSourceJNDI</jta-data-source>
        <!-- additional Mapping file, in addition to orm.xml>
        <mapping-file>META-INF/JPAorm.xml</mapping-file>

        <class>com.company.bean.jpa.PersistebleObjectImpl</class>
        <class>com.company.bean.jpa.Animal</class>
        <class>com.company.bean.jpa.Dog</class>
        <class>com.company.bean.jpa.Cat</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
           <property name="openjpa.ConnectionFactoryProperties" value="PrettyPrint=true, PrettyPrintLineLength=72"/>
           <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
        </properties>

     </persistence-unit>
</persistence>
```

What to do next

For more information on any of the commands, classes or other OpenJPA information discussed here, refer to the Apache OpenJPA User's Guide.

Related tasks

"Mapping persistent properties to XML columns" on page 242
If your database supports Extensible Markup Language (XML) column types, you can use mapping tools to manage XML objects. The Java Persistence API (JPA) specification does not contain support for mapping XML columns to Java objects. You have the choice of mapping XML columns to a Java string or a Java byte array field. These mapping techniques make using the XML objects as strings or byte arrays difficult. JPA for the application server allows you to simplify the management of XML objects by using a third-party solution for mapping management.

Associating persistence units and data sources
Java Persistence API (JPA) applications allow you to specify the underlying data source used by the persistence provider to access the database.

Configuring a JDBC provider and data source
For access to relational databases, applications use the JDBC drivers and data sources that you configure for the application server.

"Developing applications that use JNDI" on page 1414
References to enterprise bean (EJB) homes and other artifacts such as data sources are bound to the WebSphere Application Server name space. These objects can be obtained through Java Naming and Directory Interface (JNDI). Before you can perform any JNDI operations, you need to get an initial context. You can use the initial context to look up objects bound to the name space.

Task overview: Data Studio pureQuery
Data Studio pureQuery provides Java Persistence API (JPA) users an alternative way to access a DB2 database. PureQuery supports static Structured Query Language (SQL).

Configuring to use pureQuery in a Java EE environment
Use this task to configure the application data source Java Database Connectivity (JDBC) provider to use pureQuery to access DB2.

Configuring pureQuery to use multiple package collections
Set up a pureQuery Java Persistence API (JPA) application to use multiple DB2 package collections.

Related reference

wsjpaversion command
Use this command line tool to find out information about the installed version of Java Persistence API (JPA) for WebSphere Application Server.

wsappid command

The Java Persistence API (JPA) specification allows an entity's primary key to be made up of more than one column. In this case, the primary key is referred to as a ″composite″ or ″compound″ primary key. You need to provide an ID class, which is specified by the @IdClass annotation, in order to manage a composite primary key. Use the identity tool for JPA to generate an ID class for entities that use composite primary keys.

Syntax

```
Before running the command, you must have a copy of persistence.xml on the classpath,
or specify it as a properties file through the -p [path_to_persistence.xml]
argument. Issue the command from the bin subdirectory of the app_install_root directory.
```

The command syntax is as follows:

```
wsappid.sh [parameters][arguments]
```

Parameters

The wsappid tool accepts the standard set of command-line arguments that are defined by the configuration framework along with the following:

- **-directory/-d** *<output_directory>*: The path to the output directory. If the directory does not match the generated output ID class package, the package structure will be created beneath the directory. If this

parameter is not specified, the wsappid tool will attempt to find the directory of the .java file for the class that is capable of persistence, and the wsappid tool uses the current directory if a .java file is not found.
- **-ignoreErrors/-i** <true/t | false/f>: If this parameter is set to false, an exception is thrown if the tool is run on any class that does not use the application identity or is not the base class in the inheritance hierarchy.
- **-token/-t** *<token>*: The token that is used to separate the values of stringed primary keys in the string form of the object ID. This option can be used only if there are multiple primary key fields. The default is ″**::**″.
- **-name/-n** *<id_class_name>*: The name of the identity class to generate. If this option is specified, the wsappid tool must be run on exactly one class. If the class meta-data already names an ID class for the object, this option will be ignored. If the name is not fully qualified, the package of the persistence class is appended to form the fully qualified name.
- **suffix** *<id_class_suffix>*: A string with which to suffix each persistent class name to form the identity class name. This option is overridden by the **-name/-n** parameter or by any object ID class that is specified in the meta-data.

Each additional argument to the wsappid tool must be one of the following:
- The full name of a persistent class.
- The `.java` name for a persistent class.
- The `.class` file of a persistent class.

Usage

The identity tool used with JPA for application server simplifies the task of creating an identity class for entities that use composite IDs. A composite ID refers to an identity that uses more than one field as its primary key. The entity class must be compiled, and primary keys need to be identified in the entity class. Run the wsappid tool from the command line in the *app_install_root*/bin/ directory. When you run this command, a new class representing the composite ID of the entity is generated. Messages and errors are logged to the console as specified.

Examples

Consider the following entity :

```
@Entity
public class Employee {

@Id
private int division;

@Id private int id;
// . . .
}
```

Before the entity can be used we need an ID class. For this example, assume that the entity is found in the src/main/java directory.

To generate an ID class for the Magazine entity run:

```
wsappid.sh -s Id src/main/java/Employee.java -d src/main/java
```

A new class, EmployeeId.java, will be generated in the src/main/java directory.

Additional information

You can refer to the Application identity tool in persistence classes in the Apache OpenJPA User's Guide.

wsenhancer command

The entity enhancer tool for Java Persistence API (JPA) applications in the application server inserts bytecode into an entity class file that allows the JPA provider to manage the state of an entity.

JPA with the application server requires that all entity classes be enhanced if you want to manage their state. In a container-managed environment, automated enhancement is provided by the containers. In a Java SE environment, though, there are no containers to manage persistence and you might use this command frequently before packaging application files for testing. After you have created the JPA entities, you can run the wsenhancer tool to inject bytecode into the entities before packaging the JAR file into the EAR file for the application.

Syntax

Before running the command, you must have a copy of persistence.xml on the classpath, or specify it as a properties file through the -p [*path_to_persistence.xml*] argument. Issue the command from the bin subdirectory of the *app_install_root* directory.

The command syntax is as follows:

```
wsenhancer.sh [parameters][arguments]
```

Parameters

The enhancer accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-directory/-d** *<output directory>*: specifies the path to the output directory. If the directory does not match the enhanced class's package, the package structure will be created beneath the directory. By default, the enhancer overwrites the original .class file.
- **-enforcePropertyRestrictions/-epr** <true/t | false/f>: specifies whether to generate an exception when it appears that a property access entity is not obeying the restrictions that are placed on property access. The default is set to false.
- **-addDefaultConstructor/-adc** <true/t | false/f>: specifies that all of the persistent classes define a no-argument constructor. This flag informs the enhancer if it should add a protected no-arg constructor to any persistent classes in which the constructor is not already present.
- **-tmpClassLoader/-tcl** <true/t | false/f>: specifies whether the enhancer should load persistent classes with a temporary class loader. This function allows other code to load the enhanced version of the class afterwards within the same Java Virtual Machine (JVM). The default is set to true.

  Note: If you are encountering class loading problems when running the enhancer, you can set this flag to false as a debugging step.
- For class name, specify one of the following:
  - The full name of a class.
  - The `.java` name for a class.
  - The `.class` file of a class.

If you do not supply any arguments to the enhancer, it will run on the classes in your persistent class list.

Usage

In order to use the wsenhancer tool you need entities defined to JPA specifications, and the entities need to be compiled. Run the wsenhancer tool against the entities before packaging them into a JAR file. If the entities are already packaged, you must extract the entity class files, run the enhancer, and recreate the JAR file.

To enhance your entities:

- Verify that your entities are in the class path, if they are not, add them.
- Run the wsenhancer command. It is found in ${app_server_root}/bin directory.

Messages and errors are logged to the console as specified in the log settings. After invoking the wsenhancer command, your files are enhanced.

Examples

To enhance all entities on the classpath:

```
$ cd build
/home/user/myproject/build $ ${app_server_root}/bin/wsenhancer.sh
```

All entities in myproject will be enhanced.

To enhance a specific entity when you have the source files:

```
$ cd build
/home/user/myproject/build $ ${app_server_root}/bin/wsenhancer.sh Magazine.java
```

To enhance a specific entity when you have the compiled class files:

```
$ export CLASSPATH=target/classes
$ ${app_server_root}/bin/wsenhancer.sh /bin/wsenhancer.sh target/classes/jpa/example/MyEntity.class
```

The entity, Magazine.java, located in myproject will be enhanced.

Additional information

For more information about enhancement tools, refer to the section on persistent classes in the Apache OpenJPA reference documentation.

wsmapping command

The wsmapping tool is used to provide top-down mapping of the entity object model to the database relational model. You can use the wsmapping tool to create database tables.

You can use the wsmapping tool to create database tables.

Syntax

Before running the command, you must have a copy of persistence.xml on the classpath, or specify it as a properties file through the -p [*path_to_persistence.xml*] argument. Issue the command from the bin subdirectory of the *app_install_root* directory.

The command syntax is as follows:

```
wsmapping.sh [options][arguments]
```

Parameters

The mapping tool accepts the standard set of command-line arguments defined by the configuration framework with the following options:

- **-schemaAction/-sa** <add | refresh | drop | build | reflect | retain | createDB | import | export | none>: The action to execute against the schema. These options correspond to the actions of the schema tool. **Add** is the default action if none is specified. Actions can be composed in a list separated by commas.

  Note: The wsmapping tool accepts the -action/-a flag to specify the action to take on individual classes. Unless you are running wsmapping on all of your persistent types at once, or dropping a mapping, you need to use the default **add** action or the **build** action. Otherwise, you might inadvertently drop schema components that are used by classes that you are not currently running the tool against.
- **-schemaFile/-sf** <true/t | false/f>: This option can be used to write the planned schema to an XML document rather than modify the database. The XML document can then be modified, manupulated and committed to the database with the schema tool.

- **-sqlFile/-sql** <stdout | output file>: This option can be used to write the planned schema modifications to an SQL script rather than modify the database. Combine this parameter with a **schemaAction** of build to generate a script that recreates the schema for the current mappings, even if the schema already exists.
- **-dropTables/-dt** <true/t | false/f>: When this option is set to true, schema drops tables that appear to be unused during **retain** and **refresh** actions. The default is true.
- **-dropSequences/-dsq** <true/t | false/f>: If this option is set to true, schema drops sequences that are unused during **retain** and **refresh** actions. The default is true.
- **-openjpatables/-ot** <true/t | false/f>: When reflecting the schema, this parameter determines whether to reflect on tables and sequences with names that start with OPENJPA_. Certain OpenJPA components can use these tables and sequences, such as the table schema factory. When using other actions, the openjpaTables parameter controls whether these tables can be dropped or not. The default setting is false.
- **-ignoreErrors/-i** <true/t | false/f>: If set to false, an exception is displayed if the tool encounters any database errors. The default is set to false.
- **-schemas/-s** *<schema list>*: Denotes a list of schema and table names the OpenJPA should access when running the wsschema tool. This is the equivalent to setting the openjpa.jdbc.Schemas property to run once. This parameter corresponds to the **-schemas/-s** parameter in the wsschema tool. This option is ignored if **-readSchema/-rs** is not set to true.
- **-readSchema/-rs** <true/t | false/f>: Set this option to true to read the entire existing schema when the mapping tool runs. Reading the existing schema ensures that OpenJPA does not generate any mappings that use the table, index, primary key or foreign key names that conflict with existing names.

  Note: Depending on the particular JDBC driver, selecting the **-readSchema/-rs** function can slow down the process for large schemas.
- **-primaryKeys/-pk** <true/t | false/f>: This flag determines if the primary keys can be manipulated on existing tables. The default is true.
- **-foreignKeys/fk** <true/t | false/f>: This flag determines if foreign keys can be manipulated on existing tables. The default is true. This means that to add any new foreign key to a class that has already been mapped, you must explicitly set this parameter flag to true.
- **-indexes/-ix** <true/t | false/f>: This flag determines if indexes can be manipulated on existing tables. The default is true. This means that to add any new indexes to a class that has already been mapped, you must explicitly set this parameter flag to true.
- **-sequences/-sq** <true/t | false/f>: This flag determines if sequences can be manipulated. The default is true.
- **-meta/-m** <true/t | false/f>: This flag determines whether or not a mapping applies to metadata rather than, or in addition to, standard mappings.
- The wsmapping tool accepts the **-action/-a** flag to specify the action to take on each class. Multiple actions can be composed in a list, separated by commas. The available actions are:
  - **buildSchema**: This is the default action. The **buildSchema** action makes the database schema match your existing mappings. If the provided mappings conflict with the class definitions, OpenJPA fails with an informative exception.
  - **validate**: Ensure that the mappings for the given classes are valid and that they match the schema of the database. No mappings of tables are changed as a result of this action. An exception is occurs if any mappings are invalid.

Each additional argument to the wsmapping tool must be one of the following:
- The full name of a persistent class.
- The `.java` name for a persistent class.
- The `.class` file of a persistent class.

If you do not supply any arguments to the wsmapping tool, it runs on the classes in the persistent classes list.

Usage

Before running the wsmapping tool, you need to configure the datasource information, including the URL, user, and password. It is required that the wsenhancer tool is run before the wsmapping tool to insert bytecode into the entity classes. Also, the compiled class files for your entities should be on the classpath (assume entity class files can be found in target/classes) , for example:

```
export CLASSPATH=${CLASSPATH}:target/classes

wsmapping.sh ...
```

To create tables, run the wsmapping command from the ${WAS_HOME}/bin directory. When completed, the database tables are created or updated. Messages and errors are logged to the console as specified by log settings.

wsmapping.sh . . . On Windows :

Note: By specifying the buildSchema parameter to the openjpa.jdbc.SynchronizeMappings property, the mapping tool provides the default mapping that matches with the database schema automatically. You are not required to run this mapping tool if the default mapping satisfies the necessary database schema.

Examples

To create the database tables needed for the Magazine.java file:

```
${WAS_HOME}/bin/wsmapping.sh Magazine.java
```

To drop the tables for Magazine.java:

```
C:\> %WAS_HOME%/bin/wsmapping.sh -sa dropDB Magazine.java
```

To validate the mappings for all classes on the classpath:

```
C:\> %WAS_HOME%/bin/wsmapping.sh -a validate
```

Additional information

Consult chapter 7, Mapping in the OpenJPA reference documentation for more information and examples.

wsreversemapping command

The wsreversemapping tool generates persistent class definitions and metadata from a database schema.

Syntax

Before running the command, you must have a copy of persistence.xml on the classpath, or specify it as a properties file through the -p [*path_to_persistence.xml*] argument. Issue the command from the bin subdirectory of the *app_install_root* directory.

The command syntax is as follows:

```
wsreversemapping.sh [parameters][arguments]
```

Parameters

The wsreversemapping tool accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-schemas/-s** *<schema and table names>*: A list of schema and table names, separated by commas, to run the reversmapping tool on if no XML schema file is supplied. Each element of the list must follow the naming conventions for the openjpa.jdbc.Schemas property. If this parameter flag is omitted, it defaults to the value of the **Schemas** property. If the **Schemas** property is not defined, then all schemas will be reverse mapped.

- **-package/-p** *<package name>*: The package name of the generated classes. If no package name is given, the generated code will not contain package declarations.
- **-directory/-d** *<output directory>*: All generated code and metadata will be written to the directory at this path. If the path does not match the package of a class, the package structure will be created beneath this directory. This parameter defaults to the current directory.
- **-useSchemaName/-sn** <true/t | false/f>: Set this parameter flag to true to include the schema as well as the table name in the name of each generated class. This can be useful when dealing with multiple schemas that have tables with identical names.
- **-useForeignKeyName/-fkn** <true/t | false/f>: Set this parameter flag to true if you want the field names for relations to be based on the database foreign key name. By default, relation field names are derived from the name of the related class.
- **-nullableAsObject/-no** <true/t | false/f>: By default, all non-foreign key columns are mapped to primitives. Set this parameter flag to true to generate primitive wrapper fields instead for columns that allow null values.
- **-blobAsObject/-bo** <true/t | false/f>: By default, all binary columns are mapped to the byte[] fields. Set this parameter flag to true to map them to Object fields instead.

  Note: When mapped this way, the column is presumed to contain a serialized Java object.
- **-primaryKeyOnJoin/pkj** <true/t | false/f>: The standard reverse mapping tool behavior is to map all tables with primary keys to persistent classes. If your schema has primary keys on many join tables as well, set this flag to true to avoid creating classes for those tables.
- **-inverseRelations/-ir** <true/t | false/f>: Set this parameter flag to false to prevent the creation of inverse one-to-many/one-to-one relations for every many-to-one/one-to-one relation detected.
- **-useDatastoreIdentity/-ds** <true/t | false/f>: Set to true to use datastore identity for tables that have single numeric primary key columns. The tool typically uses application identity for all generated classes.
- **-useBuiltinIdentityClass/-bic** <true/t | false/f>: Set this parameter flag to false to prevent the reversemapping tool from using built-in application identity classes when possible. This will force the tool to create custom application identity classes even when there is only one primary key column.
- **-innerIdentityClasses/-inn** <true/t | false/f>: Set this parameter flag to true to have any generated application identity classed be created as static inner classes within the persistent classes. The default setting is false.
- **-identityClassSuffix/-is** *<suffix>*: Suffix to append to the class names to form application identity class names, or for inner identity classes, the inner class name. The default suffix is `Id`.
- **-typeMap/-typ** <type mapping>: A string that specifies the default Java classes to generate for each SQL type that is seen in the schema. The format is SQLTYPE1=JavaClass1, SQLTYPE2=JavaClass2. The SQL type name first looks for a customization that is based on SQLTYPE(SIZE,PRECISION), then SQLTYPE(SIZE), and then SQLTYPE. If a column with type CHAR is found, it will first look for the CHAR(50,0) type name specification, then it will look for the CHAR(50), and finally it will look for the CHAR. For example, to generate a char array for every char column whose size is exactly 50 characters, and to generate a short for every type name of INTEGER, you might specify: CHAR(50)=char[],INTEGER=short.

  Note: Various databases report different type names differently, one database type may not work for another database. Enable TRACE level logging on the MetaData channel to track which type names JPA for WebSphere Application Server is examining.
- **-customizerClass/-cc** *<class name>*: The full class name of an org.apache.openjpa.jdbc.meta.ReverseCustomizer customization plugin. If you do not specify a reverse customizer of your own, the system defaults to a PropertiesReverseCustomizer. This customizer allows you to specify simple customization options in the properties file given with the -customizerProperties flag.
  - **-customizerProperties/-cp***<properties file or resource>*: The path or resource name of a properties file to pass to the reverse customizer on initialization.
  - **-customizer/-c** *<property name> <property value>*: The given property name will be matched with the corresponding Java bean property in the specified reverse customizer, and set to the given value.

Usage

The wsreversemapping tool is used to perform reverse (bottom-up) mappings of database tables to entity source files. This is useful if developers want to generate Java files from a database for use in other JPA applications. To run this tool:

- You need to have database tables and your database connection configured.
- Run the wsreversemapping tool from the command line in the $ {WAS_HOME}/bin directory.
- The tool will generate .java files for every class along with a XML descriptor file orm.xml

The generated Java files from the wsreversemapping tool might require some editing before they can be used in an application. Also, generated files will not contain annotations. Annotations can be added manually if desired. Messages and errors are logged to the console as specified by the configuration.

Examples

Generate entities based on the information saved in the schema.xml file. Schema.xml was created by running the schema tool. The Java files are created in the **src** directory and use the package com.xyz:

```
${WAS_HOME}/bin/wsreversemapping.sh -pkg com.xyz -d ./src schema.xml
```

Generate entities based on information in a DB2 database. Entities are created in the **src** directory, and use the package com.reversemapped:

```
C:\> %WAS_HOME%/bin/wsreversemapping.bat -sa dropDB Magazine.javapkg com.reversemapped -d src
-connectionDriverName=com.ibm.db2.jcc.DB2Driver -connectionURL=jdbc:db2:localhost:50000/TEST
-connectionUser=db2User -connectionPassword=db2Password
```

Additional information

For more information, consult the mapping section in the Apache OpenJPA User's Guide.

wsschema command

The schema tool can be used to view the database schema in XML form or match an XML schema to an existing database.

Developers may find that they need the wsschema tool for its powerful functions. The wsschema tool can reflect on the current database schema, optionally translating it into an XML representation for further manipulation. Also, the schema tool can take an XML schema definition, calculate the differences between the XML and the existing database schema, and apply the necessary changes to make the databases correspond to the XML schema. The XML format used by the schema tool is abstract from the differences in SQL dialects used by different vendors. The tool also automatically adapts its SQL to meet foreign dependencies, thus the schema tool is useful as a general way to manipulate the schemas.

Syntax

The command syntax is as follows:

```
wsschema.sh [parameters][arguments]
```

Issue the command from the bin subdirectory of the *app_install_root* directory.

Parameters

The wsschema tool accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-ignoreErrors/-i** <true/t | false/f>: If set to false, an exception will be thrown if the tool encounters any database errors. The default is set to false.
- **-file/-f** *<stdout | output file>*: Use this option to write a SQL script for the planned schema modifications, rather than committing them to the database. When this is used in conjunction with the export or reflect

actions, the named file will be used to write the exported schema XML. If the file names a resource in the CLASSPATH, data will be written to that resource. Use stdout to write to standard output. The default setting is stdout.

- **-openjpatables/-ot** <true/t | false/f>: When reflecting the schema, this parameter determines whether to reflect on tables and sequences whose names start with OPENJPA_. Certain OpenJPA components may use such tables and sequences, such as the table schema factory. When using other actions, openjpaTables controls whether these tables can be dropped or not. The default setting is false.
- **-dropTables/-dt** <true/t | false/f>: When this option is set to true, schema drops tables that appear to be unused during retain and refresh actions. The default is true.
- **-dropSequences/-dsq** <true/t | false/f>: If this option is set to true, schema drops sequences that appear to be unused during retain and refresh actions. The default is true.
- **-sequences/-sq** <true/t | false/f>: This flag determines if sequences may be manipulated. The default is true.
- **-indexes/-ix** <true/t | false/f>: This flag determines if indexes may be manipulated on existing tables. The default is true.
- **-primaryKeys/-pk** <true/t | false/f>: This flag determines if primary keys may be manipulated on existing tables. The default is true.
- **-foreignKeys/fk** <true/t | false/f>: This flag determines if foreign keys may be manipulated on existing tables. The default is true.
- **-record/-r** <true/t | false/f>: This flag permits or prevents writing schema changes made by the schema tool to the current schema factory. Select true to permit writing schema changes or false to prevent writing schema changes. The default is set to true.
- **-schemas/-s** <*schema list*>: Denotes a list of schema and table names the OpenJPA should access when running the schema tool. This is the equivalent to setting the openjpa.jdbc.Schemas property to run once.

Note: The schema tool accepts the **-action/-a** flag. Multiple actions may be composed in a list, separated by commas. The available actions are:

- **add**: This is the default action if no other actions are specified. It updated the schema with the given XML documents by adding tables, columns, indexes, or other components. This action never drops any schema components.
- **retain**: This action keeps all schema components in the given XML definition but drops the rest from the database. This action never adds any schema components.
- **drop**: Drops all schema components in the schema XML. This action will drop tables only if they would have 0 columns after dropping all columns listed in the XML.
- **refresh**: This action is the equivalent of the **retain** and the **add** functions.
- **build**: Generates SQL to build a schema matching the one in the supplied XML file. Unlike the**add** action, this option does not take into account the fact that part of the schema defined in the XML file may already exist in the database. This action is typically used in conjunction with the **-file/-f** parameter flag to write a SQL script. This script may be used later to recreate the schema in the XML.
- **reflect**: Generates a XML representation of the current database schema.
- **createDB**: This action generates SQL to recreate the current database. This action is typically used in conjunction with the **-file/-f** parameter flag to write a SQL script that can be used to recreate the current schema on a new database.
- **dropDB**: Generates SQL to drop the current database. Like the **createDB** action this may be used with the **-file/-f** parameter flag to script a database drop rather than manually perform it.
- **import**: Imports the given XML schema definition into the current schema factory.

  Note: This action will do nothing if the schema factory does not store a record of the schema.
- **export**: Exports the current schema factory's stored schema definition to a XML file.

  Note: This may produce an empty file if the schema factory does not store a record of the schema.
- **deleteTableContents**: This action executes SQL to delete all rows from all tables that OpenJPA finds.

Usage

The wsschema tool is used to obtain a XML file that describes the schema of your database. To generate a XML schema file:

- You need to have database tables and your database connection configured.
- Run the wsschema tool from the command line in the $ {WAS_HOME}/bin directory.
- The tool will generate a XML file that describes the database schema.

Messages and errors are logged to the console as specified by the configuration.

Examples

Add the necessary schema components to the database to match the given XML document without dropping any data:

```
$ wsschema.sh targetSchema.xml
```

Repeat the same action as the previous example, this time not changing the database but instead writing any planned changes to a SQL script:

```
 wsschema.sh -f script.sql targetSchema.xml
```

Write an SQL script that will recreate the current database:

```
$ wsschema.sh -a createDB -f script.sql
```

Refresh the schema and delete all the contents of all the tables that OpenJPA knows about:

```
$ wsschema.bat -a refresh,deleteTableContents
```

Drop the current database:

```
$ wsschema.sh -a dropDB
```

Write a XML representation of the current schema to the file *schema.xml*:

```
$ wsschema.sh -a reflect -f schema.xml
```

Additional information

For more information, refer to chapter 4 JDBC, in the OpenJPA reference documentation.

wsdb2gen command

The command allows users to utilize the pureQuery feature in Java Persistence API (JPA) applications.

Syntax

The command syntax is as follows:

```
wsdb2gen.sh [parameters]
```

Before running the command, your `persistence.xml` file must be in the META-INF directory and the META-INF directory must be in the class path.

Parameters

- **-help** : This parameter displays the help information.
- **-pu** : The name of the persistence unit defined in `persistence.xml` file.
- **-collection** : The collection-id which is assigned to package names. The default is `NULLID`.
- **-url** : The url of the target database. This is used to validate the generated SQL. A url must be specified either in the `persistence.xml` file or as a command option. If both are specified, the url specified in the command option will be used.
- **-user** : The user id

- **-pw** : The corresponding password to connect to target database. If this parameter is not specified, the value found in the `persistence.xml` file will be used.
- **-package** : If this parameter is specified, then the -**package** parameter takes the string value package name and a single DB2 package with the specified name is generated. If the -**package** parameter is not specified, then one package is generated for each entity class. The name consists of seven or fewer letters. For each entity class, the first seven characters of the entity class is used for the package name. If the first seven characters are not unique, then the package name is changed to create a unique name.

Usage

The `persistence.xml` file must be included in the application Java archive (JAR) file and is also used as input in the DB2 bind to create the DB2 package. The wsdb2gen command requires a connection to a database in order to validate generated SQL. The database does not have to be the same as the run time database, but it should be at the same version and release level.

Ensure the following JAR files are on the class path:

- pdq.jar
- pdqmgmt.jar
- db2jcc.jar
- db2jcc_licence_cu.jar.

If the database URL specifies a DB2 for zOS database, then the following JAR file must also be on the class path: db2jcc_licence_cisuz.jar

Examples

```
wsdb2gen.sh  -pu payroll -collection prod1 -url jdbc:db2://myhostname:50000/proddb  -user produser -pw secret
```

ANT Task WsJpaDB2GenTask

The ANT task `WsJpaDB2GenTask` provides an alternative to the wsdb2gen command.

The `WsJpaDB2GenTask` ANT task utility allows users to utilize the pureQuery feature in Java Persistence API (JPA) applications. Instead of executing the wsdb2gen from the command line, you can use the example code in your ANT build XML file to use the `WsJpaDB2GenTask` in your build process.

```
Both the PDQ runtime Java archive (JAR) files, pdq.jar and pdqmgmt.jar, must be specified using the ANT -lib option.
```

Example

The example listed below could be run with the ANT command using the following:

```
<!-- invoke this build using the ANT command
ant jar -noclasspath -lib c:/was7/lib/j2ee.jar
-lib c:/was7/plugins/com.ibm.ws.jpa.jar
-lib c:/sqllib/java/db2jcc.jar
-lib c:/sqllib/java/db2jcc_license_cu.jar
-lib c:/sqllib/java/pdq.jar
-lib c:/sqllib/java/pdqmgmt.jar
-->
```

When calling the ANT command, the JAR files for pureQuery, JPA, and the JDBC driver must be on the library list.

```
<?xml version="1.0"?>


<project name="sample" default="jar">

<taskdef name="enhancer" classname="org.apache.openjpa.ant.PCEnhancerTask" />
```

```
<taskdef name="wsdb2gen" classname="com.ibm.websphere.persistence.pdq.ant.WsJpaDB2GenTask" />

<target name="clean" description="remove intermediate files">
<delete dir="classes"/>
<delete dir="enhanced" />
<delete>
<fileset dir="." includes="META-INF/*.pdqxml" />
<fileset dir="." includes="sample.jar" />
</delete>
</target>

<target name="compile"
description="compile the Java source code to class files">
<mkdir dir="classes"/>
<javac srcdir="." destdir="classes">
<classpath>
<pathelement location="c:/was7/lib/j2ee.jar"/>
<pathelement location="c:/was7/plugins/com.ibm.ws.jpa.jar" />
</classpath>
</javac>
</target>

<target name="enhance" depends="compile" >
<mkdir dir="enhanced" />
<enhancer directory="./enhanced" >
<config propertiesFile="META-INF/persistence.xml" />
<classpath>
<pathelement location="." />
<pathelement location="classes" />
</classpath>
</enhancer>
</target>


<target name="wsdb2gen" depends="enhance" >
<wsdb2gen pu="MyAntTest" url="jdbc:db2://localhost:50000/demodb" user="user1" pw="secret" >
<classpath>
<pathelement location="."/>
<pathelement location="enhanced" />
</classpath>
</wsdb2gen>
</target>


<target name="jar" depends="wsdb2gen"
description="create a Jar file for the application">
<jar destfile="sample.jar">
<fileset dir="classes" includes="**/*.class"/>
<fileset dir="." includes="META-INF/*.xml" />
</jar>
</target>
</project>
```

Developing and packaging JPA applications for a Java SE environment

Prepare and package persistence applications to test outside of the application server container in a Java SE environment.

About this task

JPA applications require different configuration techniques from applications that use container-managed persistence (CMP) or bean-managed persistence (BMP). They do not follow the typical deployment techniques that are associated with applications that implement CMP or BMP. In JPA applications, you must define a persistence unit and configure the appropriate properties in the `persistence.xml` file to ensure that the applications can run in a Java SE environment.

There are some considerations for running JPA applications in a Java SE environment:

- Resource injection is not available. You must configure these services specifically or programmatically.
- The life cycle of the EntityManagerFactory and EntityManager are managed by the application. Applications control the creation, manipulation, and deletion of these constructs programmatically.

For this task, you will need to specify the com.ibm.ws.jpa.thinclient_7.0.0.jar standalone Java archive (JAR) file in your class path. This standalone JAR file is available from the client and server install images. The location of this file on the client install image is ${*app_client_root*}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar.
The location of this file on the server install image is ${*app_server_root*}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar

1. Generate your entities classes. Depending upon your development model, you might use some or all of the JPA tools:
   - **Top-down mapping:** You start from scratch with the entity definitions and the object-relational mappings, and then you derive the database schemas from that data. If you use this approach, you are most likely concerned with creating the architecture of your object model and then writing your entity classes. These entity classes would eventually drive the creation of your database model. If you are using a top-down mapping of the object model to the relational model, develop the entity classes and then use OpenJPA functionality to generate the database tables that are based on the entity classes. The wsmapping tool would help with this approach.
   - **Bottom-up mapping:** You start with your data model, which are the database schemas, and then you work upwards to your entity classes. The wsreversemapping tool would help with this approach.
   - **Meet in the middle mapping:** probably the most common development model. You have a combination of the data model and the object model partially complete. Depending on the goals and requirements, you will need to negotiate the relationships to resolve any differences. Both the wsmapping tool and the wsreversemapping tool would help with this approach.

   The JPA solution for the application server provides several tools that help with developing JPA applications. Combining these tools with IBM Rational Application Developer provides a solid development environment for either Java EE or Java SE applications. Rational Application Developer includes GUI tools to insert annotations, a customized `persistence.xml` file editor, a database explorer, and other features. Another alternative is the Eclipse Dali project. More information on Rational Application Developer or the Eclipse Dali plugin can be found at their respective web sites.
2. Optional: Enhance the entity classes using the JPA enhancer tool, or specify the Java agent to perform dynamic enhancement at run time.
   - Use the wsenhancer tool. The enhancer post-processes the bytecode that is generated by the Java compiler and adds the fields and methods that are necessary to implement the persistence features. For example:

     ```
     ${app_client_root}/bin/wsenhancer.sh [parameters][arguments]
     ```
   - You can specify the Java agent mechanism to perform the dynamic enhancement at run time. For example, type the following at the command prompt:

     ```
     java -javaagent:${app_client_root}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar com.xyz.Main
     ```
3. Optional: If you are not using the development model for bottom-up mapping, generate or update your database tables automatically or by using the wsmapping tool.
   - By default, the object-relational mapping does not occur automatically, but you can configure the application server to provide that mapping with the openjpa.jdbc.SynchronizeMappings property. This property can accelerate development by automatically ensuring that the database tables match the object model. To enable automatic mapping, include the following line in the `persistence.xml` file:

     ```
     <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
     ```

     Note: To enable automatic object-relational mapping at run time, all of your persistent classes must be listed in the Java .class file, mapping file, and Java archive (JAR) file elements in XML format.

- To manually update or generate your database tables, run the JPA mapping tool for the application server from the command line to create the tables in the database. For example:

  ${*app_server_root*}/bin/wsmapping.sh [parameters][arguments]

4. Optional: If you are using DB2 and want to use static SQL, run the wsdb2gen command. In order to use the wsdb2gen tool, pureQuery runtime must be installed. The wsdb2gen tool creates *persistence_unit_name*.pdqxml file under the same META-INF directory where your persistence.xml file is located. If you have multiple persistence units, wsdb2gen command must be run for each persistence unit. To use the wsdb2gen tool, type the following at a command prompt:

   ${*app_server_root*}/bin/wsdb2gen.sh [parameters]

5. Optional: If you are using application-managed identity, generate an application-managed identity class with the wsappid tool. When you use an application-managed identity, one or more of the fields must be an identity field. Use an identity class if your entity has multiple identity fields and at least one of the fields is related to another entity. The application-managed identity tool generates Java code that uses the identity class for any persistent type that implements application-managed identity. For example, type the following at a prompt:

   ${*app_client_root*}/bin/wsappid.sh [parameters][arguments]

6. Configure the properties of the persistence unit in the persistence.xml file that will be used in the JPA application.
   a. Specify your data source. Use the openjpa.Connection property to obtain a connection to the database. When you run a JPA application in a Java SE environment, a JTA data source will be treated as a data source that is not JTA compliant.
   b. Select com.ibm.websphere.persistence.PersistenceProviderImpl as the persistence provider.

      Note: Include the persistence provider in the classpath if you run the JPA application as a standalone application.

   c. Specify your database configuration options. If you are using pureQuery, configure your data source to use pureQuery, ensure the pdq.jar and pdqmgmt.jar files are included on the JDBC provider classpath. For more information, see topic ″Configuring a data source to use pureQuery″. Indicate the database type and method of connection in the persistence.xml file.

      ```
      <property name="openjpa.ConnectionDriverName" value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="openjpa.ConnectionURL" value="jdbc:derby:target/database/jpa-test-database;create=true"/>
      ```

   d. Specify the transaction type to RESOURCE_LOCAL. For example, the following entry should be in the persistence.xml file:

      ```
      <persistence-unit name="persistence_unit" transaction-type="RESOURCE_LOCAL">
      ```

   e. Include the location of the object relationship mapping file, orm.xml, and any additional mapping files. For example, the following entry should be in the persistence.xml file:

      ```
      <mapping-file>META-INF/JPAorm.xml</mapping-file>
      ```

   f. Add any vendor specific properties to the persistence unit.
7. Package the application.

   Note: Package the persistence units in separate JAR files to make them more accessible and reusable. If you package the persistence units this way, they can be tested outside the container both with and without the occurrence of database persistence. The persistence units can be included in standalone applications or they can be packaged into EAR files as persistence archive files. If you package the persistence unit into a persistence archive file, all of the application components must be able to access the persistence archive. The application that uses the persistence units must declare a dependency on the persistence archive using the MANIFEST.MF Class-Path: declaration.

   Note: If you are using pureQuery, add the *persistence_unit_name*.pdqxml files created previous or the or *persistence_unit_name*.pdqxml files created in the above Step 4 to the JPA application JAR

file. The files are located in same META-INF directory where your `persistence.xml` file is located.

To package the application:

```
jar -cvf ${jar_Name} ${entity_Path}
```

where ${*jar_Name*} represents the name of the JAR file to create, and ${*entityPath*} represents the root location where the entities reside, which is where you compiled them.

8. When you run your standalone application, specify the com.ibm.ws.jpa.thinclient_7.0.0.jar standalone JAR file in your class path when executing your application. For example, use the following Java call to run the com.xyz.Main standalone application:

```
java -classpath ${app_client_root}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar other_jar_file.jar com.xyz.Main
```

Example

The following is a sample `persistence.xml` file for the Java SE Environment:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="TheWildZooPU" transaction-type="RESOURCE_LOCAL">

        <!-- additional Mapping file, in addition to orm.xml>
        <mapping-file>META-INF/JPAorm.xml</mapping-file>

        <class>com.company.bean.jpa.PersistebleObjectImpl</class>
        <class>com.company.bean.jpa.Animal</class>
        <class>com.company.bean.jpa.Dog</class>
        <class>com.company.bean.jpa.Cat</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
            <property name="openjpa.ConnectionDriverName"
                      value="org.apache.derby.jdbc.EmbeddedDriver" />
            <property name="openjpa.ConnectionURL"
                      value="jdbc:derby:target/database/jpa-test-database;create=true" />
            <property name="openjpa.Log"
                      value="DefaultLevel=INFO,SQL=TRACE,File=./dist/jpaEnhancerLog.log,Runtime=INFO,Tool=INFO" />
            <property name="openjpa.ConnectionFactoryProperties"
                      value="PrettyPrint=true, PrettyPrintLineLength=72" />
            <property name="openjpa.jdbc.SynchronizeMappings"
                      value="buildSchema(ForeignKeys=true)" />
            <property name="openjpa.ConnectionUserName"
                      value="user" />
            <property name="openjpa.ConnectionPassword"
                      value="password"/>
        </properties>

     </persistence-unit>

</persistence>
```

What to do next

For more information on any of the commands, classes or other OpenJPA information discussed here, refer to the Apache OpenJPA User's Guide.

Related tasks

“Task overview: Storing and retrieving persistent data with the Java Persistence API (JPA)” on page 218
The Java Persistence API (JPA) for the application server defines the management of persistence and object/relational mapping within Java Enterprise Edition (Java EE) and Java Standard Edition (Java SE) environments.

Associating persistence units and data sources
Java Persistence API (JPA) applications allow you to specify the underlying data source used by the persistence provider to access the database.

Task overview: Data Studio pureQuery
Data Studio pureQuery provides Java Persistence API (JPA) users an alternative way to access a DB2 database. PureQuery supports static Structured Query Language (SQL).

Configuring to use pureQuery in a Java EE environment
Use this task to configure the application data source Java Database Connectivity (JDBC) provider to use pureQuery to access DB2.

Configuring pureQuery to use multiple package collections
Set up a pureQuery Java Persistence API (JPA) application to use multiple DB2 package collections.

Related reference

“wsappid command” on page 225
The Java Persistence API (JPA) specification allows an entity’s primary key to be made up of more than one column. In this case, the primary key is referred to as a ″composite″ or ″compound″ primary key. You need to provide an ID class, which is specified by the @IdClass annotation, in order to manage a composite primary key. Use the identity tool for JPA to generate an ID class for entities that use composite primary keys.

“wsenhancer command” on page 226
The entity enhancer tool for Java Persistence API (JPA) applications in the application server inserts bytecode into an entity class file that allows the JPA provider to manage the state of an entity.

wsjpaversion command
Use this command line tool to find out information about the installed version of Java Persistence API (JPA) for WebSphere Application Server.

“wsmapping command” on page 228
The wsmapping tool is used to provide top-down mapping of the entity object model to the database relational model. You can use the wsmapping tool to create database tables.

“wsreversemapping command” on page 230
The wsreversemapping tool generates persistent class definitions and metadata from a database schema.

“wsschema command” on page 232
The schema tool can be used to view the database schema in XML form or match an XML schema to an existing database.

Enabling SQL statement batching

SQL statement batching can improve the performance of your application server. Java Persistence API (JPA) for WebSphere Application Server uses the Java Database Connectivity (JDBC) addBatch and executeBatch APIs to batch statements.

About this task

By default, statement batching is enabled for DB2 and Oracle databases. To enable SQL statement batching and to set the batch limit for JPA applications, you need to configure the persistence.xml file. The following steps review how to enable and disable statement batching, as well as set the batch limit:

1. Define the UpdateManager property in the persistence.xml file. For example:

```
<property name="openjpa.jdbc.UpdateManager"
value="com.ibm.ws.persistence.jdbc.kernel.OperationOrderUpdateManager(batchLimit=100)"/>
```

Note: The example shows that the SQL statement batch limit is set to 100.

Note: If you are using a DB2 or an Oracle database, by default the SQL statement batching is enabled and set to batchLimit=100. However, if you are using DB2 or Oracle, you are not required to specify this property in the `persistence.xml` file.

2. If you need to disable SQL statement batching, set the batchLimit value to 0 (zero) or remove the property. However, if you are using a DB2 or an Oracle database, you must specify the DBDictionary property, database, and set the defaultBatchLimit to 0 (zero). For example:

```
<property name="openjpa.jdbc.DBDictionary" value="db2(defaultBatchLimit=0)"/>
```

Results

You have now updated the `persistence.xml` file to enable or disable statement batching and set the batch limit.

Database generated version ID

Java Persistence API (JPA) for WebSphere Application Server has extended OpenJPA to work with database generated version IDs. These generated version fields (timestamp or token) can be used to efficiently detect changes to a given row.

Trigger based version ID generation is supported for all databases that WebSphere Application Server supports. Support is based on two Version Strategies in JPA for WebSphere Application Server.

- @VersionStrategy(″com.ibm.websphere.persistence.RowChangeTimestampStrategy″), if the entity version field type is Timestamp, and
- @VersionStrategy(″com.ibm.websphere.persistence.RowChangeVersionStrategy″), if the entity version field type is Long

Database generated version ID example

In this example, the Entity class is defined with the new Version Strategy annotation. The Entity has a surrogate version column.

```
@Entity(name="Item")
@VersionColumn(name="versionField")
@VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy")
public class Item implements Serializable
{
 @Id
 private int id2;

 private String name;

 private double price;

   @OneToOne
   private Owner master;
}
```

The create table statement for this would be:

```
CREATE TABLE ITEM
(ID2  INT  NOT NULL,
 NAME  VARCHAR(50) ,
 PRICE   DOUBLE,
 OWNER_ID INT,
             VERSIONFIELD GENERATED ALWAYS FOR EACH ROW ON
              UPDATE AS ROW CHANGE TIMESTAMP
         PRIMARYKEY(ID2));
```

During any updates to Item (insert or update) the VersionColumn value will be updated in the database. After the update, the value for VersionColumn is retrieved from the database and updated in the in memory object. Thereby the objects in the data cache reflect the correct version value. Here the Entity is using the @VersionColumn which produces a Surrogate Version Id rather than defining an explicit field in the entity .

The Entity could also use @Version annotation to define an explicit version field. The explicit version field could be of type Long or Timestamp corresponding to the @VersionStrategy. During any updates to Item (insert or update) the Version value will be updated in the database. After the update the value for Version would be retrieved from the database and updated in the in memory object. Thereby the objects in the data cache would reflect the right version value.

This is an example where the Entity has a version field defined, and the type Timestamp matches the RowChangeTimestampStrategy in the @VersionStrategy (if the version field type is using type long, then the RowChangeVersionStrategy should be annotated to match) :

```
@Entity(name="Item")
@VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy")
public class Item implements Serializable

{
 @Id
 private int id2;

 private String name;

 private double price;

   @Version
   private Timestamp versionField;

   @OneToOne
   private Owner master;
}
```

For z/OS DB2 V9 and Linux, Unix, and Windows DB2 V9.5, the generated database column must be of type timestamp but we support both the RowChangeTimestampStrategy and the RowChangeVersionStrategy. MS SqlServer only supports a non timestamp generated version ID that goes with the RowChangeVersionStrategy . To use the RowChangeTimestampStrategy, you must use a trigger on a timestamp field in the database. For other databases you can use triggers to simulate database version generation and use either strategy.

Mapping persistent properties to XML columns

If your database supports Extensible Markup Language (XML) column types, you can use mapping tools to manage XML objects. The Java Persistence API (JPA) specification does not contain support for mapping XML columns to Java objects. You have the choice of mapping XML columns to a Java string or a Java byte array field. These mapping techniques make using the XML objects as strings or byte arrays difficult. JPA for the application server allows you to simplify the management of XML objects by using a third-party solution for mapping management.

About this task

DB2, Oracle, and SQLServer databases support XML column types, XPath queries, and indices over these columns.

Persistent properties to XML mapping

An embedded class with XML column support needs to use XML marshalling to write the data to the XML column and unmarshalling to retrieve the data from the XML column. The path expressions and predicates over the embedded class are converted to XML predicates, XPATH expressions, or XQuery expressions and are written to the database.

WebSphere Application Server allows JPA applications to use a third-party tool for XML mapping. This is done through the extension points for custom field mappings. The third-party mapping tool uses the extension points by providing a custom value handler for the persistent fields that are mapped to the XML columns. In OpenJPA, this value handler is named org.apache.openjpa.xmlmapping.XmlValueHandler and this handler requires the @Strategy annotation on the Java field that is mapped to the XML column.

1. Annotate the entity property using the XML value handler strategy. The mapping of persistent properties to XML columns requires the @Strategy and the @Persistent annotation.

   ```
   @Persistent
   @Strategy("org.apache.openjpa.xmlmapping.XmlValueHandler")
   ```

   The XML value handler for the persistent property is set to org.apache.openjpa.xmlmapping.XmlValueHandler.
2. Change the default for fetch type if it is necessary. For example:

   ```
   @Persistence(fetch=FetchType.LAZY)
   ```

   The fetch type is now LAZY. If a value for the fetch type is not entered, the default is set to EAGER.
3. Annotate your embedded classes with the binding annotations for Java API for XML Binding (JAXB). These bindings can be created from an XML schema by using the Java Architecture for XML Binding Compiler (XJC).
4. Make sure that the class that maps to the root of the XML document is annotated with @XmlRootElement, in addition to the other annotations.
5. Compile your Java sources.
6. Run the enhancer tool on the entities. Refer to the topic on the entity enhancer tool for more information.

Example

For example, shipAddress, a property of Order Entity, is mapped to XML column shipaddr:

```
@Entity
public class Order {
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
  int oid;
 @Persistent
 @Strategy("org.apache.openjpa.xmlmapping.XmlValueHandler")
      @Column(name="shipaddr")
      Address shipAddress;
...
```

The OpenJPA mapping tool generates a SHIPADDR column with XML type in the table definition for ORDER table.

Related tasks

“Developing and packaging JPA applications for a Java EE environment” on page 221
Containers in the application server can provide many of the necessary functions for the Java Persistence API (JPA) in a Java Enterprise Edition (Java EE) environment. The application server also provides JPA tools to assist you with developing applications in a Java EE environment.

“Developing and packaging JPA applications for a Java SE environment” on page 236
Prepare and package persistence applications to test outside of the application server container in a Java SE environment.

Related reference

“wsenhancer command” on page 226
The entity enhancer tool for Java Persistence API (JPA) applications in the application server inserts bytecode into an entity class file that allows the JPA provider to manage the state of an entity.

JPA Access Intent

Java Persistence API (JPA) Access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the JDBC isolation level and whether read, update or exclusive locks are acquired when retrieving data.

About this task

Note: For a JPA persistence provider on the application server, the application can specify isolation and ReadLockMode based on a TaskName. The TaskName provides a better control over applying these characteristics. The application defines a set of entity types and their corresponding access intent for each TaskName defined in a persistence unit.

Note:

- Access intent is available for application in the Java EE server environment
- Access intent is applicable to non-query entity manager interface methods. Query should use query hint interface to set its isolation and read lock values.
- Access intent is only available for DB2 databases.
- Access intent is in effect only when pessimistic lock manager is used. Add the following to the persistence unit’s property list. <property name=″openjpa.LockManager″ value=″pessimistic″/>

The following table compares the Enterprise JavaBeans (EJB) 2.x entity bean access intent with the JPA access intent properties:

Table 7. Access intent Properties and Descriptions

WebSphere EJB 2.x entity bean access intent	JPA access intent	Description
optimistic	isolation: Read Committed	Data is read but no lock is held. Version id is used on update to insure data integrity. Other transactions can read and update data.
	lockManager: Optimistic	
	query Hint: ReadLockMode: READ	
pessimistic read	isolation: Repeatable Read	Data is read with shared locks. Other transactions attempting to update data are blocked.
	lockManager: Optimistic	
	query Hint: ReadLockMode: READ	
pessimistic update	isolation: Repeatable Read	Data is retrieved with update or exclusive lock. Other writes are blocked until commit. This access intent can be used to serialize update access to data when there are multiple writers.
	lockManager: Pessimistic	
	query Hint: ReadLockMode: WRITE	

Table 7. Access intent Properties and Descriptions (continued)

WebSphere EJB 2.x entity bean access intent	JPA access intent	Description
pessimistic exclusive	isolation: Serializable	Data is retrieved with update or exclusive lock. Other writes are blocked until commit. This access intent can be used to serialize update access to data when there are multiple writers.
	lockManager: Pessimistic	
	query Hint: ReadLockMode:WRITE	

A TaskName is set on a transaction context by one of the following:

- TaskName is automatically set via the EJB container when a transaction begins using WebSphere local transaction (EJB unspecifid transaction), JTA global transaction in a Container-Managed Transaction (CMT) or user-initiated global transaction in a Bean-Managed Transaction (BMT)
- TaskName is manually set in an application using the TaskNameAccessor API provided for JPA.

The advantage of using task names is that the access intent can be specified on a request scope rather than specifying it in the `persistence.xml` file, which has an application scope across all entities. Often a query is contained in a method or component which is used in many different transaction contexts. Some of these contexts may require repeatable-read and update lock intent but other contexts do not.

Isolation level and read locks can be specified on:

- An application scope in the `persistence.xml` file. These isolation level and read lock type are properties specified in the persistence.xml file. They apply to all entities define in the persistence unit.
- A transaction scope via task name. Transaction scoped hints will override application scope values.
- Query instance with a query hint. Query hint can be used to override isolation and ReadLockMode for a particular query instance. A query hint will override isolation level and read locks specified at the application or transaction scope.

1. “Setting a TaskName using TaskNameAccessor” This task will show how to use the TaskNameAccessor API to set JPA TaskName at runtime.
2. “Specify TaskName in a JPA persistence unit” on page 247 This task will show how to specify a TaskName in JPA persistence unit

What to do next

For further information on the background and purpose of Access intent, see the topic on Access intent service.

Related concepts

“Access intent service” on page 201
Access intent is a WebSphere Application Server runtime service that enables you to precisely manage an application’s persistence.

Related tasks

“Task overview: Storing and retrieving persistent data with the Java Persistence API (JPA)” on page 218
The Java Persistence API (JPA) for the application server defines the management of persistence and object/relational mapping within Java Enterprise Edition (Java EE) and Java Standard Edition (Java SE) environments.

Setting a TaskName using TaskNameAccessor

Using the TaskNameAccessor API to set Java Persistence API (JPA) TaskName at runtime.

About this task

In the Enterprise JavaBeans (EJB) container, a task name is automatically set by default upon a transaction begins. This action is performed when a component or business method is invoked in a CMT session bean or when an application invoke the `sessionContext.getTransaction().begin()` in a BMT session bean. This TaskName consists of a concatenation of the fully package qualified session bean type, a dot character and the method name. For example: `com.acme.MyCmtSessionBean.methodABC`.

If using JPA in the context of the Web container, an application must use the TaskNameAccessor API to set the TaskName in the current thread of execution.

Note: Once a TaskName is set on a transaction context, application must not set the TaskName again in the same transaction. This will avoid problems with on the JDBC connection for different database access.

This example contains the TaskNameAccessor API definition

```
package com.ibm.websphere.persistence;

public abstract class TaskNameAccessor {

 /**
  * Returns the current task name attached in the current thread context.
  * @return current task name or null if none is found.
  */
 public static String getTaskName ();

 /**
  * Add a user-defined JPA access intent task name to the current thread
  * context.
  *
  * @param taskName
  * @return false if an existing task has already attached in the current
  *         thread or Transaction Synchronization Registry (TSR) is not
  *         available (i.e. in JSE environment).
  */
 public static boolean setTaskName(String taskName);

}
```

This code example shows how to set a TaskName using TaskNameAccessor.

```
package my.company;

@Remote
class Ejb1 {
    // assumer no tx from the caller
    @TransactionAttribute(Requires)
    public void caller_Method1() {

        // an implicit new transaction begins
        // TaskName "my.company.Ejb1.caller_Method1" set on TSR

        ejb1.callee_Method?();
    }

    @TransactionAttribute(RequiredNew)
    public void callee_Method2() {

        // an implicit new transaction begins i.e. TxRequiredNew.
        // TaskName "my.company.Ejb1.callee_Method2" set on TSR
    }

    @TransactionAttribute(Requires)
    public void callee_Method3() {
```

```
        // In caller's transaction, hence TaskName remains
        //     "my.company.Ejb1.caller_Method1"
    }

    @TransactionAttribute(NotSupported)
    public void callee_LocalTx () {

        // Unspecified transaction, a new local transaction implicitly started.
        // TaskName "my.company.Ejb1.callee_LocalTx" set on TSR
    }

}
```

Note: In the above example, an application must be aware of transaction boundary will be subtly changed if Ejb1 uses local interface (@Local). For example, when `caller_Method1()` calls `callee_Method3` or `callee_LocalTx`, this will be treated as a Java method call. No EJB transaction semantics are honored.

What to do next

Once you have completed this step, continue on with "Specify TaskName in a JPA persistence unit."

Related tasks

"JPA Access Intent" on page 244
Java Persistence API (JPA) Access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the JDBC isolation level and whether read, update or exclusive locks are acquired when retrieving data.

"Specify TaskName in a JPA persistence unit"
Specifying a TaskName in Java Persistence API (JPA) persistence unit

Specify TaskName in a JPA persistence unit

Specifying a TaskName in Java Persistence API (JPA) persistence unit

About this task

A TaskName is defined in the `persistence.xml` file using the `wsjpa.AccessIntent` property name in a persistence unit. The property value is a list of TaskNames, entity types and access intent definitions. The following example shows the contents of the `wsjpa.AccessIntent` property name in a persistence unit.

```
<property name  = "wsjpa.AccessIntent"
         value = "Tasks=' <taskName> { <entityName> ( <isolationLockValue> ) } ' "/>
                         A            A              A                    | | |
                         |            |              +--------- , --------+ | |
                         |            +----------------- , -----------------+ |
                         +---------------------- , --------------------------+
  Tasks                        ::= <task> [ ',' <task> ]*

  <task>                       ::= <taskName> '{' <entity> [ ',' <entity> ]* '}'

  <entity>                     ::= <entityName> '(' <isolationLockValues> ')'

  <taskName>                   ::= <fully_qualified_identifier>

  <entityName>                 ::= <fully_qualified_identifier>

  <fully_qualified_identifier> ::= <identifier> [ '.' <identifier> ]*

  <identifier>                 ::= <idStartCharacter> [ <idCharacter> ]*

  <idStartCharacter>           ::= Character.isJavaIdentifierStart | '?' | '*'

  <idStartCharacter>           ::= Character.isJavaIdentifierPart  | '?' | '*'
```

```
<isolationLockValues>        ::= <isolationLockValue> [ ',' <isolationLockValue> ]

<isolationLockValue>         ::= <isolation> | <readLock>

<isolation>                  ::= "isolation" '=' <isolationValue>

<readLock>                   ::= "readlock" '=' <readlockValue>

<isolationValue>             ::= "read-uncommitted"|"read-committed"|"repeatable-read"|"serializable"

<readlockValue>              ::= "read" | "write"
```

Before setting the TaskName in a persistence unit, keep the following in mind:

- White spaces are ignored between tokens.
- Only <isolation> and <readLock> contents are not case sensitive.
- <TaskName> is in the form of a fully package qualified method name, such as com.acme.bean.MyBean.increment, or an arbitrary user-defined task name, such as MyProfile.
- <entityName> is in the form of a fully package qualified class name such as com.acme.bean.Entity1.
- The wild card characters '?' or '*' can be used in <TaskName> and <entityName>. ″?″ matches any single character and ″*″ matches zero or more sequence characters.
- Only hintNames isolation and readLock are allowed on a task definition and the order is not significant
- If readLock has the value `write`, then isolation must be `repeatable-read` or `serializable`
- If readLock has the value `read`, it has no effect if the isolation is `read-uncommitted`.

The following code example shows how to specify a TaskName in JPA persistence unit.

```
package my.company;

@Remote
class Ejb1 {
    // assumer no tx from the caller
    @TransactionAttribute(Requires)
    public void caller_Method1() {

        // an implicit new transaction begins
        // TaskName "my.company.Ejb1.caller_Method1" set on TSR

        ejb1.callee_Method?();
    }

    @TransactionAttribute(RequiredNew)
    public void callee_Method2() {

        // an implicit new transaction begins i.e. TxRequiredNew.
        // TaskName "my.company.Ejb1.callee_Method2" set on TSR
    }

    @TransactionAttribute(Requires)
    public void callee_Method3() {

        // In caller's transaction, hence TaskName remains
        //     "my.company.Ejb1.caller_Method1"
    }

    @TransactionAttribute(NotSupported)
    public void callee_LocalTx () {

        // Unspecified transaction, a new local transaction implicitly started.
        // TaskName "my.company.Ejb1.callee_LocalTx" set on TSR
    }

}
```

Since a wild card can be used to specify TaskName and entity type, multiple specification matches may occur at runtime. The order defined in the `wsjpa.AccessIntent` property will be used to search for task names and entity types.

```
<properties>
  <property name="wsjpa.AccessIntent" value="Tasks="
        *.Task1 { *.Employee1 ( isolation=read-uncommitted                ),
                  *.Employee? ( isolation=repeatable-read,  readlock=write ),
                },
        *       { *.Employee3 ( isolation=serializable,     readlock=write ) },
      '" />
</properties>
```

Related tasks

"JPA Access Intent" on page 244
Java Persistence API (JPA) Access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the JDBC isolation level and whether read, update or exclusive locks are acquired when retrieving data.

"Setting a TaskName using TaskNameAccessor" on page 245
Using the TaskNameAccessor API to set Java Persistence API (JPA) TaskName at runtime.

Chapter 5. Client applications

Using application clients

An application client module is a Java Archive (JAR) file that contains a client for accessing a Java application.

About this task

Complete the following steps for developing different types of application clients.

1. Decide on a type of application client. WebSphere Application Server for z/OS does not support a separate client image. Therefore, an application client other than a thin client must run together with the client container that comes with the application server image. See http://www.ibm.com/support/docview.wss?rs=180&uid=swg27006921 for more information about supported platforms.
2. Develop the application client code.
   a. Develop ActiveX application client code.
   b. Develop J2EE application client code.
   c. Develop pluggable application client code.
   d. Develop thin application client code.
3. Assemble the application client using the Application Server Toolkit.
4. Deploy the application client.
   Start the Application Client Resource Configuration Tool.
5. Run the application client.

Example

Application Client for WebSphere Application Server

In a traditional client-server environment, the client requests a service and the server fulfills the request. Multiple clients use a single server. Clients can also access several different servers. This model persists for Java clients except that now these requests use a client runtime environment.

WebSphere Application Server supports the pluggable client.

In this model, the client application requires a servlet to communicate with the enterprise bean, and the servlet must reside on the same machine as the WebSphere Application Server.

The Application Client for WebSphere Application Server Version 7.0 (Application Client) consists of the following client applications:

- Java Platform, Enterprise Edition (Java EE) application client application (Uses services provided by the Java EE Client Container)
- Thin application client application (Does not use services provided by the Java EE Client Container)
- Applet application client application

The Application Client for WebSphere Application Server for z/OS supports the following models:

- Java EE application client application (Uses services provided by the Java EE Client Container)
- Thin application client application (Does not use services provided by the Java EE Client Container)

The Application Client is packaged with the following components:

- Java Runtime Environment (JRE) (or an optional full Software Development Kit) that IBM provides.
- WebSphere Application Server run time for Java EE application client applications or Thin application client applications

- IBM plug-in for Java platforms for Applet client applications (Windows only)

  Note: The Pluggable application client is a kind of Thin application client. However, the Pluggable application client uses a Sun JRE and Software Development Kit instead of the JRE and Software Development Kit that IBM provides. The Sun JRE has to be the same version as the IBM JRE.

The *Applet client* model has a Java applet embedded in a HyperText Markup Language (HTML) document residing on a remote client machine from the WebSphere Application Server. With this type of client, the user accesses an enterprise bean in the WebSphere Application Server through the Java applet in the HTML document.

The *Java EE application client* is a Java application program that accesses enterprise beans, Java DataBase Connectivity (JDBC) APIs, and Java Message Service message queues. The Java EE application client program runs on client machines. This program follows the same Java programming model as other Java programs; however, the Java EE application client depends on the Application Client run time to configure its execution environment, and uses the Java Naming and Directory Interface (JNDI) name space to access resources.

The *Pluggable and Thin application clients* provide a lightweight Java client programming model. These clients are useful in situations where the client application requires a thinner, more lightweight environment than the one offered by the Java EE application client. The difference between the Thin application client and the Pluggable application client is that the Thin application client includes a Java virtual machine (JVM) API, and the Pluggable application client requires the user to provide this code. The Pluggable application client supports the Sun JRE that is directly downloaded from Sun Web site, however, the Sun JRE has to be the same version as the IBM JRE. The Application Client for WebSphere Application Server for Windows installs the ORB.properties file to configure the Sun JRE to use the IBM ORB implementation that is installed as part of the Pluggable Application Client. The Thin application client uses the IBM Developer Kit for the Java platform.

The Java EE application client programming model provides the benefits of the Java EE platform for the Java client application. Use the Java EE application client to develop, assemble, deploy and launch a client application. The tooling provided with the WebSphere Application Server platform supports the seamless integration of these stages to help the developer create a client application from start to finish.

When you develop a client application using and adhering to the Java EE platform, you can put the client application code from one Java EE platform implementation to another. The client application package can require redeployment using each Java EE platform deployment tool, but the code that comprises the client application remains the same.

The Application Client run time supplies a container that provides access to system services for the client application code. The client application code must contain a main method. The Application Client run time invokes this main method after the environment initializes and runs until the Java virtual machine code terminates.

The Java EE platform supports the Application Client use of *nicknames* or *short names*, defined within the client application deployment descriptor. These deployment descriptors identify enterprise beans or local resources (JDBC, Java Message Service (JMS), JavaMail and URL APIs) for simplified resolution through JNDI. This simplified resolution to the enterprise bean reference and local resource reference also eliminates changes to the client application code, when the underlying object or resource either changes or moves to a different server. When these changes occur, the Application Client can require redeployment.

The Application Client also provides initialization of the run-time environment for the client application. The deployment descriptor defines this unique initialization for each client application. The Application Client run time also provides support for security authentication to enterprise beans and local resources.

The Application Client uses the Java Remote Method Invocation-Internet InterORB Protocol (RMI-IIOP). Using this protocol enables the client application to access enterprise bean references and to use Common Object Request Broker Architecture (CORBA) services provided by the Java EE platform implementation. Use of the RMI-IIOP protocol and the accessibility of CORBA services assist users in developing a client application that requires access to both enterprise bean references and CORBA object references.

When you combine the Java EE and CORBA environments or programming models in one client application, you must understand the differences between the two programming models to use and manage each appropriately.

View the Samples gallery for more information about the Application Client.

Application client functions

This topic provides information about available functions in the different types of clients.

Use the following table to identify the available functions in the different types of clients.

Note: WebSphere Application Server for z/OS supports only two types of application client:

- J2EE client
- Pluggable client

Available functions	ActiveX client	Applet client	J2EE client	Pluggable client	Thin client
Provides all the benefits of a J2EE platform	Yes	No	Yes	No	No
Portable across all J2EE platforms	No	No	Yes	No	No
Provides the necessary run-time support for communication between a client and a server	Yes	Yes	Yes	Yes	Yes
Supports the use of nicknames in the deployment descriptor files. **Note:** Although you can edit deployment descriptor files, do not use the administrative console to modify them.	Yes	No	Yes	No	No
Supports use of the RMI-IIOP protocol	Yes	Yes	Yes	Yes	Yes
Browser-based application	No	Yes	No	No	No
Enables development of client applications that can access enterprise bean references and CORBA object references	Yes	Yes	Yes	Yes	Yes
Enables the initialization of the client application run-time environment	Yes	No	Yes	No	No
Supports security authentication to enterprise beans	Yes	Limited	Yes	Yes	Yes
Supports security authentication to local resources	Yes	No	Yes	No	No
Requires distribution of application to client machines	Yes	No	Yes	Yes	Yes

Enables access to enterprise beans and other Java classes through Visual Basic, VBScript, and Active Server Pages (ASP) code	Yes	No	No	No	No
Provides a lightweight client suitable for download	No	Yes	No	Yes	Yes
Enables access JNDI APIs for enterprise bean resolution	Yes	Yes	Yes	Yes	Yes
Runs on client machines that use the Sun Java Runtime Environment	No	No	No	Yes	No
Supports CORBA services (using CORBA services can render the application client code nonportable)	No	No	Yes	No	No
Supports JMS connections to the default messaging provider	No	No	Yes	No	Yes
Supports JMS connections to the default messaging provider	No	No	Yes	Yes	Yes

ActiveX application clients

WebSphere Application Server provides an ActiveX to EJB bridge that enables ActiveX programs to access enterprise beans through a set of ActiveX automation objects.

The bridge accomplishes this access by loading the Java virtual machine (JVM) into any ActiveX automation container such as Visual Basic, VBScript, and Active Server Pages (ASP).

There are two main environments in which the ActiveX to EJB bridge runs:
- **Client applications**, such as Visual Basic and VBScript, are programs that a user starts from the command line, desktop icon, or Start menu shortcut.
- **Client services**, such as Active Server Pages, are programs started by some automated means like the Services control panel applet.

The ActiveX to EJB bridge uses the Java Native Interface (JNI) architecture to programmatically access the JVM code. Therefore the JVM code exists in the same process space as the ActiveX application (Visual Basic, VBScript, or ASP) and remains attached to the process until that process terminates. To create JVM code, an ActiveX client program calls the XJBInit() method of the XJB.JClassFactory object. For more information about creating JVM code for an ActiveX program, see ActiveX to EJB bridge, initializing JVM code.

After an ActiveX client program has initialized the JVM code, the program calls several methods to create a proxy object for the Java class. When accessing a Java class or object, the real Java object exists in the JVM code; the automation container contains the proxy for that Java object. The ActiveX program can use the proxy object to access the Java class, object fields, and methods. For more information about using Java proxy objects, see ActiveX to EJB bridge, using Java proxy objects. For more information about calling methods and access fields, see ActiveX to EJB bridge, calling Java methods and ActiveX to EJB bridge, accessing Java fields.

The client program performs primitive data type conversion through the COM IDispatch interface (use of the IUnknown interface is not directly supported). Primitive data types are automatically converted between native automation types and Java types. All other types are handled automatically by the proxy objects For more information about data type conversion, see ActiveX to EJB bridge, converting data types.

Any exceptions thrown in Java code are encapsulated and thrown again as a COM error, from which the ActiveX program can determine the actual Java exceptions. For more information about handling exceptions, see ActiveX to EJB bridge, handling errors.

The ActiveX to EJB bridge supports both free-threaded and apartment-threaded access and implements the free threaded marshaler (FTM) to work in a hybrid environment such as Active Server Pages. For more information about the support for threading, see ActiveX to EJB bridge, using threading.

Applet clients

The applet client provides a browser-based Java run time capable of interacting with enterprise beans directly, instead of indirectly through a servlet.

This client is designed to support users who want a browser-based Java client application programming environment that provides a richer and more robust environment than the one offered by the **Applet > Servlet > enterprise bean** model.

The programming model for this client is a hybrid of the Java application thin client and a servlet client. When accessing enterprise beans from this client, the applet can consider the enterprise bean object references as CORBA object references.

No tooling support exists for this client to develop, assemble or deploy the applet. You are responsible for developing the applet, generating the necessary client bindings for the enterprise beans and CORBA objects, and bundling these pieces together to install or download to the client machine. The Java applet client provides the necessary run time to support communication between the client and the server. The applet client run time is provided through the Java applet browser plug-in that you install on the client machine.

Generate client-side bindings using an assembly tool. An applet can utilize these bindings, or you can generate client-side bindings using the **rmic** command. This command is part of the IBM Developer Kit, Java edition that is installed with the WebSphere Application Server.

The applet client uses the RMI-IIOP protocol. Using this protocol enables the applet to access enterprise bean references and CORBA object references, but the applet is restricted in using some supported CORBA services.

If you combine the enterprise bean and CORBA environments in one applet, you must understand the differences between the two programming models, and you must use and manage each model appropriately.

The applet environment restricts access to external resources from the browser run-time environment. You can make some of these resources available to the applet by setting the correct security policy settings in the WebSphere Application Server `client.policy` file. If given the correct set of permissions, the applet client must explicitly create the connection to the resource using the appropriate API. This client does not perform initialization of any service that the client applet can need. For example, the client application is responsible for the initialization of the naming service, either through the CosNaming, or the Java Naming and Directory Interface (JNDI) APIs.

Java EE application clients

The Java Platform, Enterprise Edition (Java EE) application client programming model provides the benefits of the Java EE Platform for WebSphere Application Server Enterprise product.

The Java EE platform offers the ability to seamlessly develop, assemble, deploy and launch a client application. The tooling provided with the WebSphere platform supports the seamless integration of these stages to help the developer create a client application from start to finish.

When you develop a client application using and adhering to the Java EE platform, you can put the client application code from one Java EE platform implementation to another. The client application package can require redeployment using each Java EE platform deployment tool, but the code that comprises the client application does not change.

The Java EE application client run time supplies a container that provides access to system services for the application client code. The Java EE application client code must contain a main method. The Java EE application client run time invokes this main method after the environment initializes and runs until the Java virtual machine application terminates.

Application clients can use *nicknames* or *short names*, defined within the client application deployment descriptor with the Java EE platform. These deployment descriptors identify enterprise beans or local resources (Java Database Connectivity (JDBC) data sources, J2C connection factories, Java Message Service (JMS), JavaMail and URL APIs) for simplified resolution through JNDI use. This simplified resolution to the enterprise bean reference and local resource reference also eliminates changes to the application client code, when the underlying object or resource either changes or moves to a different server. When these changes occur, the application client can require redeployment. Although you can edit deployment descriptor files, do not use the administrative console to modify them.

Note: Connection pool is not supported in application client. The application client calls the database directly, without a datasource. If you want to use the getConnection() request from the application client, configure the JDBC provider in the application client deployment descriptors, using Rational® Application Developer (RAD) or assembly tool. The connection is established between application client and the database. Application client does not have connection pool, but JDBC provider configuration is available in the client deployment descriptors.

The Java EE application client also provides initialization of the run-time environment for the client application. The deployment descriptor defines this unique initialization for each client application. The Java EE application client run time also provides support for security authentication to the enterprise beans and local resources.

The Java EE application client uses the Java Remote Method Invocation technology run over Internet Inter-Orb Protocol (RMI-IIOP). Using this protocol enables the client application to access enterprise bean references and to use Common Object Request Broker Architecture (CORBA) services provided by the Java EE platform implementation. Use of the RMI-IIOP protocol and the accessibility of CORBA services assist users in developing a client application that requires access to both enterprise bean references and CORBA object references.

When you combine the Java EE and the CORBA WebSphere Application Server Enterprise environments or programming models in one client application, you must understand the differences between the two programming models to use and manage each appropriately.

Pluggable application clients

The Pluggable application client provides a lightweight, downloadable Java application run time capable of interacting with enterprise beans.

Note: The Pluggable application client is available only on the Windows platform.

The Pluggable application client requires that you have previously installed the Sun Java Runtime Environment (JRE) files. In all other aspects, the Pluggable application client, and the Thin application client are similar.

This client is designed to support those users who want a lightweight Java client application programming environment, without the overhead of the Java Platform, Enterprise Edition (Java EE) platform on the client machine. The programming model for this client is heavily influenced by the CORBA programming model, but supports access to enterprise beans.

When accessing enterprise beans from this client, the client application can consider the enterprise beans object references as CORBA object references.

Tooling does not exist on the client; however, tooling does exists on the server. You are responsible for developing the client application, generating the necessary client bindings for the enterprise bean and CORBA objects, and after bundling these pieces together, installing them on the client machine.

The Pluggable application client provides the necessary run time to support the communication needs between the client and the server.

The Pluggable application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access enterprise bean references and CORBA object references and use any supported CORBA services. Using the RMI-IIOP protocol along with the accessibility of CORBA services can assist a user in developing a client application that needs to access both enterprise bean references and CORBA object references.

When you combine the Java EE and CORBA environments in one client application, you must understand the differences between the two programming models to use and manage each appropriately.

The Pluggable application client run time provides the necessary support for the client application for object resolution, security, Reliability Availability and Serviceability (RAS), and other services. However, this client does not support a container that provides easy access to these services. For example, no support exists for using *nicknames* for enterprise beans or local resource resolution. When resolving to an enterprise bean (using either the Java Naming and Directory Interface (JNDI) API or CosNaming) sources, the client application must know the location of the name server and the fully qualified name used when the reference was bound into the name space.

When resolving to a local resource, the client application cannot resolve to the resource through a JNDI lookup. Instead the client application must explicitly create the connection to the resource using the appropriate API (JDBC, Java Message Service (JMS), and so on). This client does not perform initialization of any of the services that the client application might require. For example, the client application is responsible for the initialization of the naming service, either through CosNaming or JNDI APIs.

The Pluggable application client offers access to most of the available client services in the Java EE application client. However, you cannot access the services in the Pluggable application client as easily as you can in the Java EE application client. The Java EE client has the advantage of performing a simple JNDI name space lookup to access the desired service or resource. The Pluggable application client must code explicitly for each resource in the client application. For example, looking up an enterprise bean Home object requires the following code in a Java EE application client:

```
java.lang.Object ejbHome =
 initialContext.lookup("java:/comp/env/ejb/MyEJBHome");
MyEJBHome =
 (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
```

However, you need more explicit code in a Pluggable application client for Java:

```
java.lang.Object ejbHome =
 initialContext.lookup("the/fully/qualified/path/to/actual/home/in/namespace/MyEJBHome");
MyEJBHome =
 (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
```

In this example, the Java EE application client accesses a logical name from the `java:/comp` name space. The Java EE client run time resolves that name to the physical location and returns the reference to the client application. The pluggable client must know the fully qualified physical location of the enterprise bean Home object in the name space. If this location changes, the pluggable client application must also change the value placed on the lookup() statement.

In the Java EE client, the client application is protected from these changes because it uses the logical name. A change can require a redeployment of the EAR file, but the actual client application code remains the same.

The Pluggable application client is a traditional Java application that contains a *main* function. The WebSphere Pluggable application client provides run-time support for accessing remote enterprise beans, and provides the implementation for various services. This client can also access CORBA objects and CORBA-based services. When using both environments in one client application, you need to understand the differences between the enterprise bean and the CORBA programming models to manage both environments.

For instance, the CORBA programming model requires the CORBA CosNaming name service for object resolution in a name space. The enterprise beans programming model requires the JNDI name service. The client application must initialize and properly manage these two naming services.

Another difference applies to the enterprise bean model. Use the JNDI implementation in the enterprise bean model to initialize the Object Request Broker (ORB). The client application is unaware that an ORB is present. The CORBA model, however, requires the client application to explicitly initialize the ORB through the `ORB.init()` static method.

The Pluggable application client provides a batch command that you can use to set the CLASSPATH and JAVA_HOME environment variables to enable the Pluggable application client run time.

Thin application clients

The Thin application client provides a lightweight, downloadable Java application run time capable of interacting with enterprise beans.

WebSphere Application Server supports the pluggable client.

The Thin application client is designed to support those users who want a lightweight Java client application programming environment, without the overhead of the Java Platform, Enterprise Edition (Java EE) platform on the client machine. The programming model for this client is heavily influenced by the CORBA programming model, but supports access to enterprise beans.

When accessing enterprise beans from this client, the client application can consider the enterprise beans object references as CORBA object references.

Tooling does not exist on the client, it exists on the server. You are responsible for developing the client application, generating the necessary client bindings for the enterprise bean and CORBA objects, and bundling these pieces together to install on the client machine.

The Thin application client provides the necessary runtime to support the communication needs between the client and the server.

The Thin application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access not only enterprise bean references and CORBA object references, but also allows the client application to use any supported CORBA services. Using the RMI-IIOP protocol along with the accessibility of CORBA services can assist a user in developing a client application that needs to access both enterprise bean references and CORBA object references.

When you combine the Java EE and CORBA environments in one client application, you must understand the differences between the two programming models, to use and manage each appropriately.

The Thin application client run time provides the necessary support for the client application for object resolution, security, Reliability Availability and Servicability (RAS), and other services. However, this client does not support a container that provides easy access to these services. For example, no support exists

for using *nicknames* for enterprise beans or local resource resolution. When resolving to an enterprise bean (using either Java Naming and Directory Interface (JNDI) or CosNaming) sources, the client application must know the location of the name server and the fully qualified name used when the reference was bound into the name space. When resolving to a local resource, the client application cannot resolve to the resource through a JNDI lookup. Instead the client application must explicitly create the connection to the resource using the appropriate API (JDBC, Java Message Service (JMS), and so on). This client does not perform initialization of any of the services that the client application might require. For example, the client application is responsible for the initialization of the naming service, either through CosNaming or JNDI APIs.

The Thin application client offers access to most of the available client services in the Java EE application client. However, you cannot access the services in the thin client as easily as you can in the Java EE application client. The Java EE client has the advantage of performing a simple Java Naming and Directory Interface (JNDI) name space lookup to access the desired service or resource. The thin client must code explicitly for each resource in the client application. For example, looking up an enterprise bean Home requires the following code in a Java EE application client:

```
java.lang.Object ejbHome =  initialContext.lookup("java:comp/env/ejb/MyEJBHome");
MyEJBHome = (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome,  MyEJBHome.class);
```

However, you need more explicit code in a Thin application client:

```
java.lang.Object ejbHome =
  initialContext.lookup("the/fully/qualified/path/to/actual/home/in/namespace/MyEJBHome");
MyEJBHome = (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome,  MyEJBHome.class);
```

In this example, the Java EE application client accesses a logical name from the `java:comp` name space. The Java EE client run time resolves that name to the physical location and returns the reference to the client application. The Thin application client must know the fully qualified physical location of the enterprise bean Home in the name space. If this location changes, the thin client application must also change the value placed on the lookup() statement.

In the Java EE client, the client application is protected from these changes because it uses the logical name. A change might require a redeployment of the EAR file, but the actual client application code remains the same.

The Thin application client is a traditional Java application that contains a *main* function. The WebSphere Thin application client provides run-time support for accessing remote enterprise beans, and provides the implementation for various services. This client can also access CORBA objects and CORBA based services. When using both environments in one client application, you need to understand the differences between the enterprise bean and CORBA programming models to manage both environments.

For instance, the CORBA programming model requires the CORBA CosNaming name service for object resolution in a name space. The enterprise beans programming model requires the JNDI name service. The client application must initialize and properly manage these two naming services.

Another difference applies to the enterprise bean model. Use the Java Naming and Directory Interface (JNDI) implementation in the enterprise bean model to initialize the Object Request Broker (ORB). The client application is unaware that an ORB is present. The CORBA model, however, requires the client application to explicitly initialize the ORB through the `ORB.init()` static method.

The Thin application client provides a batch command that you can use to set the CLASSPATH and JAVA_HOME environment variables to enable the Thin application client run time.

WebSphere Application Server Version 6.1 and the Application client for WebSphere Application Server Version 6.1 also provides specialized types of Thin application client. The Web Services Thin Client is an unmanaged, stand-alone Java client environment that allows you to run JAX-RPC Web services client

applications to invoke Web services that are hosted by the application server. The Administration Thin Client enables client applications to perform administration tasks outside of the application server environment.

Application client troubleshooting tips

This topic provides debugging tips for resolving common Java 2 Platform Enterprise Edition (J2EE) application client problems. To use this troubleshooting guide, review the trace entries for one of the J2EE application client exceptions, and then locate the exception in the guide.

Some of the errors in the guide are samples, and the actual error you receive can be different than what is shown here. You might find it useful to rerun the `launchClient` command specifying the `-CCverbose=true` option. This option provides additional information when the J2EE application client run time is initializing.

Error: java.lang.NoClassDefFoundError

Explanation This exception is thrown when Java code cannot load the specified class.

Possible causes
- Invalid or non-existent class
- Class path problem
- Manifest problem

Recommended response

Check to determine if the specified class exists in a Java Archive (JAR) file within your Enterprise Archive (EAR) file. If it does, make sure the path for the class is correct. For example, if you get the exception:

```
java.lang.NoClassDefFoundError:
WebSphereSamples.HelloEJB.HelloHome
```

verify that the HelloHome class exists in one of the JAR files in your EAR file. If it exists, verify that the path for the class is WebSphereSamples.HelloEJB.

If both the class and path are correct, then it is a class path issue. Most likely, you do not have the failing class JAR file specified in the client JAR file manifest. To verify this situation, perform the following steps:

1. Open your EAR file with an assembly tool, and select the Application Client.
2. Add the names of the other JAR files in the EAR file to the Classpath field.

This exception is generally caused by a missing Enterprise Java Beans (EJB) module name from the Classpath field.

If you have multiple JAR files to enter in the Classpath field, be sure to separate the JAR names with spaces.

If you still have the problem, you have a situation where a class is loaded from the file system instead of the EAR file. This error is difficult to debug because the offending class is not the one specified in the exception. Instead, another class is loaded from the file system before the one specified in the exception. To correct this error, review the class paths specified with the -CCclasspath option and the class paths configured with the Application Client Resource Configuration Tool.

Alternatively, you can use the Client Container Resource Configuration Scripting tool.

Look for classes that also exist in the EAR file. You must resolve the situation where one of the classes is found on the file system instead of in the `.ear` file. Remove entries from the classpaths, or include the `.jar` files and classes in the `.ear` file instead of referencing them from the file system.

If you use the -CCclasspath parameter or resource classpaths in the tool, and you have configured multiple JAR files or classes, verify they are separated with the correct character for your operating system. Unlike the Classpath field, these class path fields use platform-specific separator characters.
Note: The system class path is not used by the Application Client run time if you use the launchClient batch or shell files. In this case, the system class path would not cause this problem. However, if you load the launchClient class directly, you do have to search through the system class path as well.

Error: com.ibm.websphere.naming.CannotInstantiateObjectException: Exception occurred while attempting to get an instance of the object for the specified reference object. [Root exception is javax.naming.NameNotFoundException: xxxxxxxxxx]

Explanation

This exception occurs when you perform a lookup on an object that is not installed on the host server. Your program can look up the name in the local client Java Naming and Directory Interface (JNDI) name space, but received a NameNotFoundException exception because it is not located on the host server. One typical example is looking up an EJB component that is not installed on the host server that you access. This exception might also occur if the JNDI name you configured in your Application Client module does not match the actual JNDI name of the resource on the host server.

Possible causes

- Incorrect host server invoked
- Resource is not defined
- Resource is not installed
- Application server is not started
- Invalid JNDI configuration

Recommended response

If you are accessing the wrong host server, run the `launchClient` command again with the -CCBootstrapHost parameter specifying the correct host server name. If you are accessing the correct host server, use the product `dumpnamespace` command line tool to see a listing of the host server JNDI name space. If you do not see the failing object name, the resource is either not installed on the host server or the appropriate application server is not started. If you determine the resource is already installed and started, your JNDI name in your client application does not match the global JNDI name on the host server. Use the Application Server Toolkit to compare the JNDI bindings value of the failing object name in the client application to the JNDI bindings value of the object in the host server application. The values must match.

Error: javax.naming.ServiceUnavailableException: A communication failure occurred while attempting to obtain an initial context using the provider url: ″iiop://[invalidhostname]″. Make sure that the host and port information is correct and that the server identified by the provider URL is a running name server. If no port number is specified, the default port number 2809 is used. Other possible causes include the network environment or workstation network configuration. Root exception is org.omg.CORBA.INTERNAL: JORB0050E: In Profile.getIPAddress(), InetAddress.getByName[invalidhostname] threw an UnknownHostException. minor code: 4942F5B6 completed: Maybe

Explanation

This exception occurs when you specify an invalid host server name.

Possible causes

- Incorrect host server invoked
- Invalid host server name

Recommended response

Run the `launchClient` command again and specify the correct name of your host server with the -CCBootstrapHost parameter.

Error: javax.naming.CommunicationException: Could not obtain an initial context due to a communication failure. Since no provider URL was specified, either the bootrap host and port of an existing ORB was used, or a new ORB instance was created and initialized with the default bootstrap host of ″localhost″ and the default bootstrap port of 2809. Make sure the ORB bootstrap host and port resolve to a running name server. Root exception is org.omg.CORBA.COMM_FAILURE: WRITE_ERROR_SEND_1 minor code: 49421050 completed: No

Explanation

This exception occurs when you run the `launchClient` command to a host server that does not have the Application Server started. You also receive this exception when you specify an invalid host server name. This situation might occur if you do not specify a host server name when you run the launchClient tool. The default behavior is for the launchClient tool to run to the local host, because WebSphere Application Server does not know the name of your host server. This default behavior only works when you are running the client on the same machine with WebSphere Application Server is installed.

Possible causes

- Incorrect host server invoked
- Invalid host server name
- Invalid reference to `localhost`
- Application server is not started
- Invalid bootstrap port

Recommended response

If you are not running to the correct host server, run the `launchClient` command again and specify the name of your host server with the `-CCBootstrapHost` parameter. Otherwise, start the Application Server on the host server and run the `launchClient` command again.

Error: javax.naming.NameNotFoundException: Name comp/env/ejb not found in context ″java:″

Explanation

This exception is thrown when the Java code cannot locate the specified name in the local JNDI name space.

Possible causes

- No binding information for the specified name
- Binding information for the specified name is incorrect
- Wrong class loader was used to load one of the program classes
- A resource reference does not include any client configuration information
- A client container on the deployment manager is trying to use enterprise extensions (not supported)

Recommended response

Open the EAR file with the Application Server Toolkit, and check the bindings for the failing name. Ensure this information is correct. If you are using Resource References, open the EAR file with the Application Client Resource Configuration Tool, and verify that the Resource Reference has client configuration information and the name of the Resource Reference exactly matches the JNDI name of the client configuration. If the values are correct, you might have a class loader error. For detailed information about the configuration tool and opening EAR files, read the Starting the Application Client Resource Configuration Tool in the *Developing and deploying applications* PDF book.

Error: java.lang.ClassCastException: Unable to load class: org.omg.stub.WebSphereSamples.HelloEJB._HelloHome_Stub at com.ibm.rmi.javax.rmi.PortableRemoteObject.narrow(portableRemoteObject.java:269)

Explanation

This exception occurs when the application program attempts to narrow to the EJB home class and the class loaders cannot find the EJB client side bindings.

Possible causes

- The files, *_Stub.class and _Tie.class, are not in the EJB `.jar` file
- Class loader could not find the classes

Recommended response

Look at the EJB `.jar` file located in the `.ear` file and verify the class contains the Enterprise Java Beans (EJB) client side bindings. These are class files with file names that end in `_Stub` and `_Tie`. If the binding classes are in the EJB `.jar` file, then you might have a class loader error.

Error: WSCL0210E: The Enterprise archive file [EAR file name] could not be found. com.ibm.websphere.client.applicationclient.ClientContainerException: com.ibm.etools.archive.exception.OpenFailureException

Explanation

This error occurs when the application client run time cannot read the Enterprise Archive (EAR) file.

Possible causes

The most likely cause of this error is that the system cannot find the EAR file in the path specified on the `launchClient` command.

Recommended response

Verify that the path and file name specified on the `launchclient` command are correct.

The launchClient command appears to hang and does not return to the command line when the client application has finished.

Explanation

When running your application client using the `launchClient` command the WebSphere Application Server run time might need to display the security login dialog. To display this dialog, WebSphere Application Server run time creates an Abstract Window Toolkit (AWT) thread. When your application returns from its main method to the application client run time, the application client run time attempts to return to the operating system and end the Java virtual machine (JVM) code. However, since there is an AWT thread, the JVM code will not end until `System.exit` is called.

Possible causes

The JVM code does not end because there is an AWT thread. Java code requires that `System.exit()` be called to end AWT threads.

Recommended response

- Modify your application to call `System.exit(0)` as the last statement.
- Use the `-CCexitVM=true` parameter when you call the `launchClient` command.

IBM Support has documents and tools that can save you time gathering information needed to resolve problems as described in Troubleshooting help from IBM. Before opening a problem report, see the Support page:

- http://www.ibm.com/software/webservers/appserv/zos_os390/support/

clientUpgrade command

Use the **clientUpgrade** command to migrate previous versions of client resources to Version 7 level resources.

Use the **clientUpgrade** command to migrate Version 5.1.x and Version 6.x client resources to Version 7 level resources. In the process of migrating these resources, the client-resources.xmi file located in the client jars is migrated to the latest level. A backup of the client-resources.xmi file is also located in the client jar. If this command is not executed against the client EAR files before they are installed on Version 7, the client EARs do not operate or install correctly.

The command file is located in the *app_server_root*/bin directory.

```
clientUpgrade EAR_file [-clientJar client_jar ][-logFileLocation logFileLocation]
[-traceString trace_spec [-traceFile file_name ]]
```

Parameters

Supported arguments include the following:

EAR_file
Use this parameter to specify the fully qualified path to the EAR file that contains client JAR files to process.

-clientJar
Use this optional parameter to specify a JAR file for processing. If not specified, the program transforms all client JAR files in the EAR file.

-logFileLocation *log_file_location*
Use this optional parameter to specify an alternate location to store the log output.

-traceString *trace_spec* **-traceFile** *file_name*
Use these optional parameters to gather trace information for IBM Service personnel. Specify a trace_spec of `"*=all=enabled"` (with quotation marks) to gather all trace information.

The following example demonstrates correct syntax:

```
clientUpgrade EAR_file -clientJar ejbJarFile
```

Developing application clients

This topic provides the steps for programming application clients to access resource objects defined on the server.

About this task

To use application clients to access a remote object on the server, develop your application clients as described in the following steps:

1. Create an instance of the object that you want to access from the remote server.
2. Specify the user ID and password on the connection method, when you create a connection to the server. Security must be enabled.
3. Assemble the application client `.ear` file using an assembly tool. Assemble the application client `.ear` file on any development machine where the assembly tool is installed.
4. Add the resource to the client deployment descriptor by completing the binding JNDI name for the resource object on the server.
5. Distribute the configured `.ear` file to the client machines.
6. Deploy the application client.
7. Configure the application client resources.

What to do next

After you develop the application client code, run the application client.

Developing ActiveX application client code

This topic provides an outline for developing an ActiveX Windows program, such as Visual Basic, VBScript, and Active Server Pages, to use the WebSphere ActiveX to EJB bridge to access enterprise beans.

Before you begin

Note: This topic assumes that you are familiar with ActiveX programming and developing on the Windows platform.

Consider the information given in ActiveX to EJB bridge as good programming guidelines.

About this task

To use the ActiveX to EJB bridge to access a Java class, develop your ActiveX program to complete the following steps:

1. Create an instance of the XJB.JClassFactory object.
2. Create Java virtual machine (JVM) code within the ActiveX program process, by calling the XJBInit() method of the XJB.JClassFactory object. After the ActiveX program has created an XJB.JClassFactory object and called the XJBInit() method, the JVM code is initialized and ready for use.
3. Create a proxy object for the Java class, by using the XJB.JClassFactory FindClass() and NewInstance() methods. The ActiveX program can use the proxy object to access the Java class, object fields, and methods.
4. Call methods on the Java class, using the Java method invocation syntax, and access Java fields as required.
5. Use the helper functions to do the conversion in cases where automatic conversion is not possible. You can convert between the following data types:
   - Java Byte and Visual Basic Byte
   - Visual Basic Currency types and Java 64-bit
6. Implement methods to handle any errors returned from the Java class. In Visual Basic or VBScript, use the **Err.Number** and **Err.Description** fields to determine the actual Java error.

What to do next

After you develop the ActiveX client code, start the ActiveX application.

Starting an ActiveX application

To run an ActiveX client application that is to use the ActiveX to Enterprise Java Beans (EJB) bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its `XJB.JAR` file and the Java run time. This initial configuration sets up the environment within which the ActiveX client application can run.

About this task

To perform the required configuration, complete one or more of the following tasks:

1. Start an ActiveX application and configure service programs.
2. Start an ActiveX application and configuring non-service programs

Starting an ActiveX application and configuring service programs

To run an ActiveX service program such as Active Server Page (ASP) that is to use the ActiveX to the Enterprise Java Bean (EJB) bridge, some initial configuration (to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its `XJB.JAR` file and the Java run time) is necessary. This configuration sets up the environment within which the ActiveX service program can run.

Before you begin

The XJB.JClassFactory must find the Java run time dynamic link library (DLL) when initializing. In a service program such as Internet Information Server you cannot specify a path for its processes independently; you must set the process paths in the system PATH variable. This limitation means that you can only have a single Java virtual machine (JVM) version available on a machine using ASP.

About this task

To add the Java Runtime Environment (JRE) directories to your system path, complete one of the following task.

On Windows 2000 systems, complete the following steps:

1. Open the Control Panel, then double-click the **System** icon.
2. Click the **Advanced** tab on the System Properties window.
3. Click **Environment Variables**.
4. Edit the Path variable in the System Variables window.
5. Add the following information to the beginning of the path that is displayed in the Variable Value field:

   ```
   C:\WebSphere\AppClient\Java\jre\bin;C:\WebSphere\AppClient\Java\jre\bin\classic;
   ```

   where `C:\WebSphere\AppClient` is the directory in which you installed the Java client in the WebSphere product.
6. Click **OK** in the Edit System Variable window to apply the changes.
7. Click **OK** in the Environment Variables window.
8. Click **OK** in the System Properties window.
9. Restart Windows 2000.

What to do next

After you change the system PATH variable you must reboot the Internet Information Server machine so that Internet Information Server can see the change.

Starting an ActiveX application and configuring non-service programs

To run an ActiveX program initiated from an icon or command line (a non-service program) that is to use the ActiveX to the Enterprise Java Beans (EJB) bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its `XJB.JAR` file and the Java run-time environment. This uses a batch file to set up the environment within which the ActiveX program can run.

About this task

To perform the required configuration, complete the following steps:

1. Edit the `setupCmdLineXJB.bat` file to specify appropriate values for the environment variables required by the ActiveX to EJB bridge. For more information about these environment variables, see ActiveX to EJB bridge, environment and configuration. For more information about creating a JVM for an ActiveX program, see ActiveX to EJB bridge, initializing the Java virtual machine (JVM). After the ActiveX program has created an XJB.JClassFactory object and called the XJBInit() method, the JVM is initialized and ready for use.
2. Start the ActiveX client application by using one of the following methods:
   - Use the `launchClientXJB.bat` file to start the application. For example:

```
launchClientXJB MyApplication.exe parm1 parm2
```

or

```
launchClientXJB MyApplication.vbp
```

- Use the `setupCmdLineXJB.bat` file to create an environment in which to run the application, then start the application from within that environment.

setupCmdLineXJB.bat, launchClientXJB.bat and other ActiveX batch files

This topic provides reference information about the aids that client applications and client services can use to access the ActiveX to EJB bridge. These enable the ActiveX to Enterprise JavaBeans (EJB) bridge to find its `XJB.JAR` file and the Java run-time environment.

Location

The include file is located in the *was_client_home*`\aspIncludes` directory. You can include the file into your Active Server Pages (ASP) application with the following syntax in your ASP page:

```
<--  #include virtual ="/WSASPIncludes/setupASPXJB.inc" -->
```

This syntax assumes that you have created a virtual directory in Internet Information Server called `WSASPIncludes` that points to the *was_client_home*`\aspIncludes` directory.

Usage notes

The following batch files are provided for client applications to use the ActiveX to EJB bridge:

- **setupCmdLineXJB.bat**

  Sets the client environment variables.
- **launchClientXJB.bat**

  Calls the `setupCmdLineXJB.bat` file and launches the application you specify as its arguments; for example:

  ```
  launchClientXJB.bat myapp.exe parm1 parm2
  ```

  or

  ```
  launchClientXJB MyApplication.vbp
  ```
- **Active Server Pages (ASP) include file**

  An include file is provided for ASP users to automatically set the following page-level (local) environment variables:
  - **com_ibm_websphere_javahome**. Path to the Java run-time directory installed with the WebSphere advanced server client.
  - **com_ibm_websphere_washome**. Path to the WebSphere advanced server client directory.
  - **com_ibm_websphere_namingfactory**. Sets the Java java.naming.factory.initial system property.
  - **com_ibm_websphere_computername**. (Optional) Name of the computer where the WebSphere Advanced Server Client is installed. If you intend to talk to a single specific computer, you are recommended to change this value to become the server name that you intend to access.
- **System settings**

  To enable the ActiveX to EJB bridge to access the Java run-time dynamic link library (DLL), the following directories must exist in the system PATH environment variable:

  was_client_home`\java\jre\bin;`*was_client_home*`\java\jre\bin\classic`

  Where *was_client_home* is the name of the directory where you installed the WebSphere Application Server client (for example, `C:\WebSphere\AppClient`).

  Note: This technique enables only one Java run time to activate on a machine, therefore all client services on that machine must use the same Java run time. Client applications do not have this limitation because they each have their own private, non-system scope.

JClassProxy and JObjectProxy classes

The majority of tasks for accessing your Java classes and objects are handled with the JClassProxy and JObjectProxy objects. This topic provides reference information about the object classes of the ActiveX to Enterprise Java Beans (EJB) bridge.

JClassFactory is the object used to access the majority of Java Virtual Machine (JVM) features. This object handles JVM initialization, accesses classes and creates class instances (objects). Use the JClassProxy and JObjectProxy objects to access the majority of your Java classes and objects:

- XJBInit(String astrJavaParameterArray())

  Initializes the JVM environment using an array of strings that represent the command line parameters you normally send to the `java.exe` file.

  If you have invalid parameters in the XJBInit() string array, the following error is displayed:

  ```
  Error: 0x6002 "XJBJNI::Init() Failed to create VM" when calling XJBInit()
  ```

  If you have C++ logging enabled, the activity log displays the invalid parameter.
- JClassProxy FindClass(String strClassName)

  Uses the current thread class loader to load the specified fully qualified class name and returns a JClassProxy object representing the Java Class object.
- JObjectProxy NewInstance()

  Creates a Class instance for the specified JClassProxy object using the parameters supplied to call the Class constructor. For more information about using the JMethodArgs method, see ActiveX to EJB bridge, calling Java methods.

```
JObjectProxy NewInstance(JClassFactory obj, Variant vArg1, Variant vArg2, Variant vArg3, ...)
JObjectProxy NewInstance(JClassFactory obj, JMethodArgs args)
```

- JMethodArgs GetArgsContainer()

  Returns a JMethodArgs object (Class instance).

  You can create a JClassProxy object from the JClassFactory.FindClass() method and from any Java method call that normally return a Java Class object. You can use this object as if you had direct access to the Java Class object. All of the class static methods and fields are accessible as are the java.lang.Class methods. In case of a clash between static method names of the reflected user class and those of the java.lang.Class (for example, getName()), the reflected static methods would execute first.

  For example, the following is a static method called getName(). The java.lang.Class object also has a method called getName():

  – In Java:

  ```
  class foo{
   foo(){};
   public static String getName(){return "abcdef";}
   public static String getName2(){return "ghijkl";}
   public String toString2(){return "xyz";}
  }
  ```

  – In Visual Basic:

  ```
  ...
  Dim clsFoo as Object
  set clsFoo = oXJB.FindClass("foo")
  clsFoo.getName()  ' Returns "abcdef" from the static foo class
  clsFoo.getName2() ' Returns "ghijkl" from the static foo class
  clsFoo.toString() ' Returns "class foo" from the java.lang.Class object.
  oFoo = oXJB.NewInstance(clsFoo)
  oFoo.toString() ' Returns some text from the java.lang.Object's
                  ' toString() method which foo inherits from.
  oFoo.toString2() ' Returns "xyz" from the foo class instance
  ```

  You can create a JObjectProxy object from the JClassFactory.NewInstance() method, and can be created from any Java method call that normally returns a Class instance object. You can use this

object as if you had direct access to the Java object and can access all the static methods and fields of the object. All of object instance methods and fields are accessible (including those accessible through inheritance).

The JMethodArgs object is created from the JClassFactory.GetArgsContainer() method. Use this object as a container for method and constructor arguments. You must use this object when overriding the object type when calling a method (for example, when sending a java.lang.String JProxyObject type to a constructor that normally takes a java.lang.Object type).

You can use two groups of methods to add arguments to the collection: Add and Set. You can use Add to add arguments in the order that they are declared. Alternatively, you can use Set to set an argument based on its position in the argument list (where the first argument is in position 1).

For example, if you had a Java Object Foo that took a constructor of Foo (int, String, Object), you could use a JMethodArgs object as shown in the following code extract:

```
...
Dim oArgs as Object
set oArgs = oXJB.GetArgsContainer()

oArgs.AddInt(CLng(12345))
oArgs.AddString("Apples")
oArgs.AddObject("java.lang.Object", oSomeJObjectProxy)

Dim clsFoo as Object
Dim oFoo as Object
set clsFoo = oXJB.FindClass("com.mypackage.foo")
set oFoo = oXJB.NewInstance(clsFoo, oArgs)

' To reuse the oArgs object, just clear it and use the add method
' again, or alternatively, use the Set method to reset the parameters
' Here, we will use Set
oArgs.SetInt(1, CLng(22222))
oArgs.SetString(2, "Bananas")
oArgs.SetObject(3, "java.lang.Object", oSomeOtherJObjectProxy)

Dim oFoo2 as Object
set oFoo2 = oXJB.NewInstance(clsFoo, oArgs)
```

- AddObject (String strObjectTypeName, Object oArg)

  Adds an arbitrary object to the argument container in the next available position, casting the object to the class name specified in the first parameter. Arrays are specified using the traditional [] syntax; for example:

  ```
  AddObject("java.lang.Object[][]", oMy2DArrayOfFooObjects)
  ```

  or

  ```
  AddObject("int[]", oMyArrayOfInts)
  ```

- AddByte (Byte byteArg)

  Adds a primitive byte value to the argument container in the next available position.

- AddBoolean (Boolean bArg)

  Adds a primitive boolean value to the argument container in the next available position.

- AddShort (Integer iArg)

  Adds a primitive short value to the argument container in the next available position.

- AddInt (Long lArg)

  Adds a primitive int value to the argument container in the next available position.

- AddLong (Currency cyArg)

  Adds a primitive long value to the argument container in the next available position.

- AddFloat (Single fArg)

  Adds a primitive float value to the argument container in the next available position.

- AddDouble (Double dArg)

  Adds a primitive double value to the argument container in the next available position.

- AddChar (String strArg)

  Adds a primitive char value to the argument container in the next available position.
- AddString (String strArg)

  Adds the argument in string form to the argument container in the next available position.
- SetObject (Integer iArgPosition, String strObjectTypeName, Object oArg)

  Adds an arbitrary object to the argument container in the specified position casting it to the class name or primitive type name specified in the second parameter. Arrays are specified using the traditional [] syntax; for example:

  ```
  SetObject(1, "java.lang.Object[][]", oMy2DArrayOfFooObjects)
  ```

  or

  ```
  SetObject(2, "int[]", MyArrayOfInts)
  ```
- SetByte (Integer iArgPosition, Byte byteArg)

  Sets a primitive byte value to the argument container in the position specified.
- SetBoolean (Integer iArgPosition, Boolean bArg)

  Sets a primitive boolean value to the argument container in the position specified.
- SetShort (Integer iArgPosition, Integer iArg)

  Sets a primitive short value to the argument container in the position specified.
- SetInt (Integer iArgPosition, Long lArg)

  Sets a primitive int value to the argument container in the position specified.
- SetLong (Integer iArgPosition, Currency cyArg)

  Sets a primitive long value to the argument container in the position specified.
- SetFloat (Integer iArgPosition, Single fArg)

  Sets a primitive float value to the argument container in the position specified.
- SetDouble (Integer iArgPosition, Double dArg)

  Sets a primitive double value to the argument container in the position specified.
- SetChar (Integer iArgPosition, String strArg)

  Sets a primitive char value to the argument container in the position specified.
- SetString (Integer iArgPosition, String strArg)

  Sets a java.lang.String value to the argument container in the position specified.
- Object Item(Integer iArgPosition)

  Returns the value of an argument at a specific argument position.
- Clear()

  Removes all arguments from the container and resets the next available position to one.
- Long Count()

  Returns the number of arguments in the container.

Java virtual machine initialization tips

Initialize the Java virtual machine (JVM) code with the ActiveX to Enterprise Java Beans (EJB) bridge. For an ActiveX client program (Visual Basic, VBScript, or ASP) to access Java classes or objects, the first step that the program must do is to create Java virtual machine (JVM) code within its process.

To create JVM code, the ActiveX program calls the XJBInit() method of the XJB.JClassFactory object. When an XJB.JClassFactory object is created and the XJBInit() method called, the JVM is initialized and ready to use.

- To enable the XJB.JClassFactory to find the Java run-time description definition language (DLL) when initializing, the Java Runtime Environment (JRE) bin and `bin\classic` directories must exist in the system path environment variable.
- The XJBInit() method accepts only one parameter: an array of strings. Each string in the array represents a command line argument that for a Java program you would normally specify on the

`Java.exe` command line. This string interface is used to set the class path, stack size, heap size and debug settings. You can get a listing of these parameters by typing `java -?` from the command line.
- If you set a parameter incorrectly, you receive a `0x6002 "Failed to initialize VM"` error message.
- Due to the current limitations of Java Native Interface (JNI), you cannot unload or reinitialize the JVM code after it has loaded. Therefore, after the XJBInit() method has been called once, subsequent calls have no effect other than to create a duplicate JClassFactory object for you to access. It is best to store your XJB.JClassFactory object globally and continue to reuse that object.
- The following Visual Basic extract shows an example of initializing JVM code:

```
Dim oXJB as Object
set oXJB = CreateObject("XJB.JClassFactory")
Dim astrJavaInitProps(0) as String
astrJavaInitProps(0) = _
     "-Djava.class.path=.;c:\myjavaclasses;c:\myjars\myjar.jar"
oXJB.XJBInit(astrJavaInitProps)
```

Example: Developing an ActiveX application client to enterprise beans

This reference topic provides an example of using Java proxy objects with the ActiveX to Enterprise JavaBeans (EJB) bridge.

To use Java proxy objects with the ActiveX to Enterprise JavaBeans (EJB) bridge:
- After an ActiveX client program (Visual Basic, VBScript, or Active Server Pages (ASP)) has initialized the XJB.JClassFactory object and thereby, the Java virtual machine (JVM), the client program can access Java classes and initialize Java objects. To complete this action, the client program uses the XJB.JClassFactory FindClass() and NewInstance() methods.
- In Java programming, two ways exists to access Java classes: direct invocation through the Java compiler and through the Java Reflection interface. Because the ActiveX to Java bridge needs no compilation and is a complete run-time interface to the Java code, the bridge depends on the latter Reflection interface to access its classes, objects, methods and fields. The XJB.JClassFactory FindClass() and NewInstance() methods behave very similarly to the Java Class.forName() and the Method.invoke() and Field.invoke() methods.
- XJB.JClassFactory.FindClass() takes the fully qualified class name as its only parameter and returns a Proxy Object (JClassProxy). You can use the returned Proxy object like a normal Java Class object and call static methods and access static fields. You can also create a Class Instance (or object), as described below. For example, the following Visual Basic code extract returns a Proxy object for the java.lang.Integer Java class:

  ```
  ...
  Dim clsMyString as Object
  Set clsMyString = oXJB.FindClass("java.lang.Integer")
  ```
- After the proxy is created, you can access its static information directly. For example, you can use the following code extract to convert a decimal integer to its hexadecimal representation:

  ```
  ...
  Dim strHexValue as String
  strHexValue = clsMyString.toHexString(CLng(255))
  ```
- The equivalent Java syntax is: `static String toHexString(int i)`. Because ints units in Java programming are really 32-bit (which translates to Long in Visual Basic), the CLng() function converts the value from the default int to a long. Also, even though the toHexString() function returns a java.lang.String, the code extract does not return an Object proxy. Instead, the returned java.lang.String is automatically converted to a native Visual Basic string.

  To create an object from a class, you use the JClassFactory.NewInstance() method. This method creates an Object instance and takes whatever parameters your class constructor needs. Once the object is created, you have access to all of its public instance methods and fields. For example, you can use the following Visual Basic code extract to create an instance of the java.lang.Integer string:

```
...
Dim oMyInteger as Object
set oMyInteger = oXJB.NewInstance(CLng(255))

Dim strMyInteger as String
strMyInteger = oMyInteger.toString
```

Example: Calling Java methods in the ActiveX to enterprise beans

In the ActiveX to Enterprise Java Beans (EJB) bridge, methods are called using the native language method invocation syntax.

The following differences between Java invocation and ActiveX Automation invocation exist:

- Unlike Java methods, ActiveX does not support method (and constructor) polymorphism; that is, you cannot have two methods in the same class with the same name.
- Java methods are case-sensitive, but ActiveX Automation is not case-sensitive.
- To compensate for Java polymorphic behavior, give the exact parameter types to the method call. The parameter types determine the correct method to invoke. For a listing of correct types to use, see ActiveX to EJB bridge, converting data types.
- For example, the following Visual Basic code fails if the CLng() method was not present or the toHexString syntax was incorrectly typed as ToHexString:

  ```
  ...
  Dim strHexValue as String
  strHexValue = clsMyString.toHexString(CLng(255))
  ```
- Sometimes it is difficult to force some development environments to leave the case of your method calls unchanged. For example, in Visual Basic if you want to call a method `close()` (lowercase), the Visual Basic code capitalizes it `"Close()"`. In Visual Basic, the only way to effectively work around this behavior is to use the CallByName() method. For example:

  ```
  o.Close(123)                                'Incorrect...
  CallByName(o, "close", vbMethod, 123)       'Correct...
  ```

  or in VBScript, use the `Eval` function:

  ```
  o.Close(123)                                 'Incorrect...
  Eval("o.Close(123)")                         'Correct...
  ```
- The return value of a function is always converted dynamically to the correct type. However, you must take care to use the set keyword in Visual Basic. If you expect a non-primitive data type to return, you must use set. (If you expect a primitive data type to return, you do not need to use set.) See the following example for more explanation:

  ```
  Set oMyObject = o.getObject
  iMyInt = o.getInt
  ```
- In some cases, you might not know the type of object returning from a method call, because wrapper classes are converted automatically to primitives (for example, java.lang.Integer returns an ActiveX Automation Long). In such cases, you might need to use your language built-in exception handling techniques to try to coerce the returned type (for example, On Error and Err.Number in Visual Basic).
- Methods with character arguments

  Because ActiveX Automation does not natively support character types supported by Java methods, the ActiveX to EJB bridge uses strings (byte or VT_I1 do not work because characters have multiple bytes in Java code). If you try to call a method that takes a char or java.lang.Character type you must use the JMethodArgs argument container to pass character values to methods or constructors. For more information about how this argument container is used, see Methods with "Object" Type as Argument and Abstract Arguments.
- Methods with "Object" Type as Argument and Abstract Arguments

  Because of the polymorphic nature of Java programming, the ActiveX to Java bridge uses direct argument type mapping to find a method. This method works well in most cases, but sometimes methods are declared with a Parent or Abstract class as an argument type (for example, java.lang.Object). You need the ability to send an object of arbitrary type to a method. To acquire this

ability, you must use the XJB.JMethodArgs object to coerce your parameters to match the parameters on your method. You can get a JMethodArgs instance by using the JClassFactory.GetArgsContainer() method.

The JMethodArgs object is a container for method parameters or arguments. This container enables you to add parameters to it one-by-one and then you can send the JMethodArgs object to your method call. The JClassProxy and JObjectProxy objects recognize the JMethodArgs object and attempt to find the correct method and let the Java language coerce your parameters appropriately.

For example, to add an element to a Hashtable object the method syntax is `Object put(Object key, Object value)`. In Visual Basic, the method usage looks like the following example code:

```
Dim oMyHashtable as Object
Set oMyHashtable = _
    oXJB.NewInstance(oXJB.FindClass("java.utility.Hashtable"))

' This line will not work. The ActiveX to EJB bridge cannot find a method
' called "put" that has a short and String as a parameter:
oMyHashtable.put 100, "Dogs"
oMyHashtable.put 200, "Cats"

' You must use a XJB.JMethodArgs object instead:
Dim oMyHashtableArgs as Object
Set oMyHashtableArgs = oXJB.GetArgsContainer
oMyHashtableArgs.AddObject("java.lang.Object", 100)
oMyHashtableArgs.AddObject("java.lang.Object", "Dogs")

oMyHashtable.put oMyHashTableArgs
' Reuse the same JMethodArgs object by clearing it.
oMyHashtableArgs.Clear
oMyHashtableArgs.AddObject("java.lang.Object", 200)
oMyHashtableArgs.AddObject("java.lang.Object", "Cats")

oMyHashtable.put oMyHashTableArgs
```

Java field programming tips

Using the ActiveX to Enterprise JavaBeans (EJB) bridge to access Java fields has the same case sensitivity issue that it has when invoking methods. Field names must use the same case as the Java field syntax.

Visual Basic code has the same problem with unsolicited case changing on fields as it does with methods. (For more information about this problem, see ActiveX to EJB bridge, calling Java methods). You might use the CallByName() function to set a field in the same way that you call a method in some cases. For fields, use `VBLet` for primitive types and `VBSet` for objects. For example:

```
o.MyField = 123                         'Incorrect...
CallByName(o, "MyField", vbLet, 123)    'Correct...
```

or in VBScript:

```
o.MyField = 123                          'Incorrect...
Eval("o.myField = 123")                  'Correct...
```

ActiveX to Java primitive data type conversion values

All primitive Java data types are automatically converted to native ActiveX Automation types. However, not all Automation data types are converted to Java types (for example, VT_DATE). Variant data types are used for data conversion.

Variant data types are a requirement of any Automation interface, and are used automatically by Visual Basic and VBScript. The tables below provide details about how primitive data types are converted between Automation types and Java types.

Table 8. ActiveX to Java primitive data type conversion

Visual Basic Type	Variant Type	Java Type	Notes®
Byte	VT_I1	byte	Byte in Visual Basic is unsigned, but is signed in Java data type.
Boolean	VT_BOOL	boolean	
Integer	VT_I2	short	
Long	VT_I4	int	
Currency	VT_CY	long	
Single	VT_R4	float	
Double	VT_R8	double	
String	VT_BSTR	java.lang.String	
String	VT_BSTR	char	
Date	VT_DATE	n/a	

Example: Using helper methods for data type conversion

Generally, data type conversion between ActiveX (Visual Basic and VBScript) and Java methods occurs automatically, as described in ActiveX to EJB bridge, converting data types. However, the byte helper function and currency helper function are provided for cases where automatic conversion is not possible:

- Byte helper function

  Because the Java Byte data type is signed (-127 through 128) and the Visual Basic Byte data type is unsigned (0 through 255), convert unsigned Bytes to a Visual Basic Integers, which look like the Java signed byte. To make this conversion, you can use the following helper function:

```
Private Function GetIntFromJavaByte(Byte jByte) as Integer
      GetIntFromJavaByte = (CInt(jByte) + 128) Mod 256 - 128
End Function
```

- Currency helper function

  Visual Basic 6.0 cannot properly handle 64-bit integers like Java methods can (as the Long data type). Therefore, Visual Basic uses the Currency type, which is intrinsically a 64-bit data type. The only side effect of using the Currency type (the Variant type VT_CY) is that a decimal point is inserted into the type. To extract and manipulate the 64-bit Long value in Visual Basic, use code like the following example. For more details on this technique for converting Currency data types, see Q189862, ″HOWTO: Do 64-bit Arithmetic in VBA″, on the Microsoft® Knowledge Base.

```
' Currency Helper Types
Private Type MungeCurr
    Value As Currency
End Type
Private Type Munge2Long
    LoValue As Long
    HiValue As Long
End Type

' Currency Helper Functions
Private Function CurrToText(ByVal Value As Currency) As String
    Dim Temp As String, L As Long
    Temp = Format$(Value, "#.0000")
    L = Len(Temp)
    Temp = Left$(Temp, L - 5) & Right$(Temp, 4)
    Do While Len(Temp) > 1 And Left$(Temp, 1) = "0"
        Temp = Mid$(Temp, 2)
    Loop
    Do While Len(Temp) > 2 And Left$(Temp, 2) = "-0"
        Temp = "-" & Mid$(Temp, 3)
    Loop
```

```
    CurrToText = Temp
End Function

Private Function TextToCurr(ByVal Value As String) As Currency
    Dim L As Long, Negative As Boolean
    Value = Trim$(Value)
    If Left$(Value, 1) = "-" Then
        Negative = True
        Value = Mid$(Value, 2)
    End If
    L = Len(Value)
    If L < 4 Then
        TextToCurr = CCur(IIf(Negative, "-0.", "0.") & _
                          Right$("0000" & Value, 4))
    Else
        TextToCurr = CCur(IIf(Negative, "-", "") & _
                          Left$(Value, L - 4) & "." & Right$(Value, 4))
    End If
End Function

' Java Long as Currency Usage Example
Dim LC As MungeCurr
Dim L2 As Munge2Long

' Assign a Currency Value (really a Java Long)
' to the MungeCurr type variable
LC.Value = cyTestIn

' Coerce the value to the Munge2Long type variable
LSet L2 = LC

' Perform some operation on the value, now that we
' have it available in two 32-bit chunks
L2.LoValue = L2.LoValue + 1

' Coerce the Munge value back into a currency value
LSet LC = L2
cyTestIn = LC.Value
```

Array tips for ActiveX application clients

Arrays are very similar between Java and Automation containers like Visual Basic and `VBScript`. This topic provides some important points to consider when passing arrays back and forth between these containers.

Here are some important points to consider when passing arrays back and forth between these containers:

- Java arrays cannot mix types. All Java arrays contain a single type, so when passing arrays of variants to a Java array, you must make sure that all of the elements in the variant array are of the same base type. For example, in Visual Basic code:

  ```
  ...
  Dim VariantArray(1) as Variant
  VariantArray(0) = CLng(123)
  VariantArray(1) = CDbl(123.4)
  oMyJavaObject.foo(VariantArray) '     Illegal!

  VariantArray(0) = CLng(123)
  VariantArray(1) = CLng(1234)
  oMyJavaObject.foo(VariantArray) '     This works
  ```

- Arrays of primitive types are converted using the rules defined in primitive data type conversion.
- Arrays of Java objects are handled through arrays of JObjectProxy objects.

- Arrays of JObjectProxy objects must be fully initialized and of the correct associated Java type. When initializing an array in Visual Basic (for example, Dim oJavaObjects(1) as Object), you must set each object to a JObjectProxy object before you send the array to a Java object. The bridge is unable to determine the type of null or empty object values.
- When receiving an array from a Java method, the lower-bound is always zero. Java methods only support zero-based arrays.
- Nested or multidimensional arrays are treated as zero-based multidimensional arrays in Visual Basic and VBScript containers.
- Uninitialized arrays or Array Types are unsupported. When calling a Java method that takes an array of objects as a parameter, you must fully initialize the array of JObjectProxy objects.

Error handling codes for ActiveX application clients

All exceptions thrown in Java code are encapsulated and thrown again as a COM error through the ISupportErrorInfo interface and the EXCEPINFO structure of IDispatch::Invoke(), the Err object in Visual Basic and VBScript. Because there are no error numbers associated with Java exceptions, whenever a Java exception is thrown, the entire stack trace is stored in the error description text and the error number assigned is 0x6003.

In Visual Basic or VBScript, you need to use the **Err.Number** and **Err.Description** fields to determine the actual Java error. Non-Java errors are thrown as you would expect via the IDispatch interface; for example, if a method cannot be found, then error 438 ″Object doesn't support this property or method″ is thrown.

Error number	Description
0x6001	Java Native Interface (JNI) error
0x6002	Initialization error
0x6003	Java exception. Error description is the Java Stack Trace.
0x6FFF	General Internal Failure

Threading tips

The ActiveX to Enterprise JavaBeans (EJB) bridge supports both free-threaded and apartment-threaded access and implements the Free Threaded Marshaler to work in a hybrid environment such as Active Server Pages (ASP). Each thread created in the ActiveX process is mirrored in the Java environment when the thread communicates through the ActiveX to EJB bridge.

Once all references to Java objects (there are no JObjectProxy or JClassProxy objects) are loaded in an ActiveX thread, the ActiveX to EJB bridge detaches the thread from the Java virtual machine (JVM) code. Therefore, you must be careful that any Java code that you access from a multithreaded Windows application is thread safe. Visual Basic code and VBScript applications are both essentially single threaded. Therefore, Visual Basic and VBScript applications do not have threading issues in the Java programs they access. Active Server Pages and multithreaded C and C++ programs can have issues.

Consider the following scenario:
1. A multithreaded Windows Automation Container (our ActiveX Process) starts. It exists on Thread A.
2. The ActiveX Process initializes the ActiveX to EJB bridge, which starts the JVM code. The JVM attaches to the same thread and internally calls it Thread 1.
3. The ActiveX Process starts two threads: B and C.
4. Thread B in the ActiveX Process uses the ActiveX to EJB bridge to access an object that was created in Thread A. The JVM attaches to thread B and calls it Thread 2.
5. Thread C in the ActiveX Process never talks to the JVM code, so the JVM never needs to attach to it. This is a case where the JVM code does not have a one-to-one relationship between ActiveX threads and Java threads.

6. Thread B later releases all of the JObjectProxy and JClassProxy objects that it used. The Java Thread 2 is detached.
7. Thread B again uses the ActiveX to EJB bridge to access an object that was created in Thread A. The JVM code attaches again to the thread and calls it Thread 3.

ActiveX process	JVM access by ActiveX process
Thread A - Created in 1	Thread 1 - Attached in 2
Thread B - Created in 4	Thread 2 - Attached in 4, detached in 6 Thread 3 - Attached in 7
Thread C - Created in 4	

Threads and Active Server Pages

Active Server Pages (ASP) in Microsoft Internet Information Server is a multithreaded environment. When you create the XJB.JClassFactory object, you can store it in the Application collection as an Application-global object. All threads within your ASP environment can now access the same ActiveX to EJB bridge object. Active Server Pages by default creates 10 Apartment Threads per ASP process per CPU. This means that when your ActiveX to EJB bridge object is initialized any of the 10 threads can call this object, not just the thread that created it.

If you need to simulate single-apartment behavior, you can create a Single-Apartment Threaded ActiveX dynamic link library (DLL) in Visual Basic code and encapsulate the ActiveX to the EJB bridge object. This encapsulation guarantees that all access to the JVM object is on the same thread. You need to use the <OBJECT> tag to assign the XJB.JClassFactory to an Application object and must be aware of the consequences of introducing single-threaded behavior to a Web application.

The Microsoft KnowlegeBase has several articles about ASP and threads, including:
- Q243543 INFO: Do Not Store STA Objects in Session or Application
- Q243544 INFO: Component Threading Model Summary Under Active Server Pages
- Q243548 INFO: Design Guidelines for VB Components Under ASP

Example: Viewing a System.out message

The ActiveX to Enterprise JavaBeans (EJB) bridge does not have a console available to view Java System.out messages. To view these messages when running a stand-alone client program (such as Visual Basic), redirect the output to a file.

This example redirects output to a file:

```
launchClientXJB.bat MyProgram.exe > output.txt
```

- To view the System.out messages when running a Service program such as Active Server Pages, you need to override the Java System.out OutputStream object to FileOutputStream. For example, in VBScript:

```
'Redirect system.out to a file
' Assume that oXJB is an initialized XJB.JClassFactory object
 Dim clsSystem
 Dim oOS
 Dim oPS
 Dim oArgs

' Get the System class
  Set clsSystem = oXJB.FindClass("java.lang.System")

' Create a FileOutputStream object
' Create a PrintStream object and assign to it our FileOutputStream
  Set oArgs = oXJB.GetArgsContainer  oArgs.AddObject "java.io.OutputStream", oOS
```

```
  Set oPS = oXJB.NewInstance(oXJB.FindClass("java.io.PrintStream"), oArgs)

' Set our System OutputStream to our file
  clsSystem.setOut oPS
```

Example: Enabling logging and tracing for application clients

The ActiveX to EJB bridge provides two logging and tracing formats: Windows Application Event Log and Java Trace Log.

- Windows Event Log

  The Windows Application Event Log shows JNI errors, Java console error messages, and XJB initialization messages. This log is most useful for determining `XJBInit()` errors and any unusual exceptions that do not come from the Java environment. By default, critical error logging will be enabled and debug and event logging is disabled.

  To enable or disable logging of certain event types to the Windows Event Log, specify one or more parameters to `XJBInit()`. If more than one parameter is set, they will be processed in the order in which they appear in the input string array to the `XJBInit()` method. Once the `XJBInit()` method is initialized, these parameters can no longer be set/reset for the life of the process. Using Java `java.lang.System.setProperty()` to set these values also has no effect.

  – `-Dcom.ibm.ws.client.xjb.native.logging.debug=enabled|disabled`

    Enables or disables debug level messages from displaying in the Windows operating system event log. This level of logging is most useful and shows most internal errors, user programming issues or configuration problems.

  – `-Dcom.ibm.ws.client.xjb.native.logging.event=enabled|disabled`

    Enables or disables event level messages from appearing in the Windows operating system event log.

  – `-Dcom.ibm.ws.client.xjb.native.logging.*=enabled|disabled`

    Enables or disables both event and debug level messages from appearing in the Windows operating system event log. It is not possible to disable some critical error messages from being displayed in the error log. Only debug and event level messages can be disabled.

  Viewing the Windows application event log with the event viewer:

  To open the event viewer in the Windows operating system:

  1. Click **Start** > **Settings** > **Control Panel**.
  2. Double-click **Administrative Tools**.
  3. Double-click **Event Viewer**.

  All ActiveX to EJB bridge events display the text `WebSphere XJB` in the source column and in the application log. For information about using Event Viewer, click the **Action** menu in Event Viewer, and then click **Help**.

  To open the even viewer in the Windows operating system, click **Start** > **Programs** > **Administrative Tools** > **Event Viewer**. All ActiveX to EJB bridge events have the text `WebSphere XJB` in the source column and display in the application log. For information about using Event Viewer, click the **Help** menu in Event Viewer.

- Java trace log

  The Java trace log displays information that you can use to debug method calls, class lookups, and argument coercion problems. Since the Java portion of the bridge mirrors the function of the COM IDispatch interface, the information in the trace log is similar to what you have come to expect from an IDispatch interface. To understand the trace log, you need a fundamental understanding of IDispatch.

  To enable user-logging, add the following parameters to the XJBInit() input string array:

```
"-DtraceString=com.ibm.ws.client.xjb.*=event=enabled"
"-DtraceFile=C:\MyTrace.txt"
```

ActiveX client programming best practices

The best way to access Java components is to use the Java language. It is recommended that you do as much programming as possible in the Java language and use a small simple interface between your COM Automation container (for example, Visual Basic) and the Java code. This interface avoids any overhead and performance problems that can occur when moving across the interface.

Note: The following topics are covered:

- Visual Basic guidelines
- CScript and Windows Scripting Host
- Active Server Pages guidelines
- J2EE guidelines

Visual Basic guidelines

The following guidelines are intended to help optimize your use of the ActiveX to EJB bridge with Visual Basic:

- Launch the Visual Basic replication through the `launchClientXJB.bat` file. If you want to run your Visual Basic application through the Visual Basic debugger, run the Visual Basic integrated development environment (IDE) within the ActiveX to EJB bridge environment. After you create your Visual Basic project, you can launch it from a command line; for example, `launchClientXJB MyApplication.vbp`. You can also launch the Visual Basic application alone in the ActiveX to EJB environment, by changing the Visual Basic shortcut on the Windows Start menu so that the `launchClientXJB.bat` file precedes the call to the `VB6.EXE` file.
- Exit the Visual Basic IDE before debugging programs.

  Because the Java virtual machine (JVM) code attaches to the running process, you must exit the Visual Basic editor before debugging your program. If you run the process, then exit your program within the Visual Basic IDE, the JVM code continues to run and you reattach the same JVM code when XJBInit() is called by the debugger. This causes problems if you try to update XJBInit() arguments (for example, classpath) because the changes are not be applied until you restart the Visual Basic program.
- Store the XJB.JClassFactory object globally.

  Because you cannot unload or reinitialize the JVM code, cache the resulting XJB.JClassFactory object as a global variable. The overhead of treating this object as a global variable or passing a single reference around is much less than recreating a new XJB.JClassFactory object and calling the XJBInit() argument more than once.

CScript and Windows Scripting Host

The following guidelines intend to help optimize your use of the ActiveX to EJB bridge with CScript and Windows Scripting Host (WSH):

- Launch in ActiveX to EJB environment.

  Launch the VBScript files in the ActiveX to EJB bridge environment, to run VBScript files in `.vbs` files. Two common ways exist to launch your script:
  - `launchClientXJB MyScript.vbs`
  - `launchClientXJB cscript MyScript.vbs`

Active Server Pages guidelines

The following guidelines intend to help optimize your use of the ActiveX to EJB bridge with Active Server Pages software:

- Use the ActiveX to EJB Helper functions from the Active Server Pages Application.

  Because Active Server Pages (ASP) code typically use VBScript, you can use the included helper functions in any VBScript environment with minor changes. For more information about these helper functions, see Helper functions for data type conversion. To run outside of the ASP environment,

remove or change all references to the Server, Request, Response, Application and Session objects; for example, change `Server.CreateObject` to `CreateObject`.

- Set JRE path globally in system.

  The XJB.JClassFactory object must be able to find the Java run time dynamic link library (DLL) when initializing. In Internet Information Server, you cannot specify a path for its processes independently; you must set the process paths in the system PATH variable. You can only have a single JVM version available on a machine using the ASP application. Also, remember that after you change the system PATH variable you must reboot the Internet Information Server machine so that the Internet Information Server can see the change.
- Set the system TEMP environment variable.

  If the system TEMP environment variable is not set, Internet Information Server stores all temporary files in the `WINNT` directory, which is usually not desired.
- Use high isolation or an isolated process.

  When using the ActiveX to Java bridge with Active Server Pages software, creating your Web application in its own process is recommended. You can only load one JVM instruction in a single process and if you want to have more than one application running with different JVM environment options (for example, different classpaths), then you need to have separate processes.
- Use the Application Unload option.

  When debugging your application, use **Unload** when viewing your ASP application properties in the Internet Information Server administration console to unload the process from memory and thereby unload the JVM code.
- Run one process per application.

  Use only one ASP application per J2EE application or JVM environment, in your ASP environment. If you need separate class paths or JVM settings, you need separate ASP applications (virtual directories with high isolation or an isolated process).
- Store the XJB.JClassFactory object in application scope.

  Because of the one-to-one relationship required between a JVM instruction and a process, and because the JVM code can never detach or shut down from a process independently, cache the XJB.JClassFactory object at application scope and call the XJBInit() method only once.

  Because the ActiveX to EJB bridge employs a free-threaded marshaler, take advantage of the multi-threaded nature of Internet Information Server and the ASP environment. If you choose to reinitialize the XJB.JClassFactory object at Page scope (local variables), then the XJBInit() method can only initialize your local XJB.JClassFactory variable. It is more efficient to use the XJBInit() method once.
- Use VBScript conversion functions.

  Because VBScript code only supports variant data types, use the CStr(), CByte(), CBool(), CCur(), CInt(), Clng(), CSng() and CDbl() functions to tell the activeX to EJB bridge which data type you are using; for example `oMyObject.Foo(CDbl(1.234))`.

J2EE guidelines

The following guidelines are intended to help optimize your use of the ActiveX to EJB bridge with the J2EE environment;

- Store client container objects globally.

  Because you can only have one JVM instruction per process, and a single J2EE client container (com.ibm.websphere.client.applicationclient.launchClient) per JVM instruction, initialize your J2EE client container only once and reuse it. For ASP applications, store the J2EE client container in an application level variable and initialize it only once (either on the Application_OnStart() event in the `global.asa` file or by checking to see if it IsEmpty()).

  A side effect to storing the client container object globally is that you cannot change the client container parameters without destroying the object and creating a new one. These parameters include the EAR file, BootstrapHost, class path, and so on. If you run a Visual Basic application and want to change the client container parameters, you must end the application and restart it. If you run an Active Server

Pages application, you must first unload the application from Internet Information Server (see ″Use the Application Unload Button″ under Active Server Pages guidelines). Then load the Active Server Pages application with the different client container parameters. The parameters set the first time the Active Server Pages application loads. Since the client container is stored on the Internet Information Server, all the browser clients share the parameters using the Active Server Pages application. This behavior is normal for Active Server Pages code, but can be confusing when you try to run to different WebSphere Application Servers using the same Active Server Pages application, which is not supported.

- Reuse custom temporary directory for EAR file extraction.

  By default, the client container launches and extracts the application `.ear` file to your `temp` directory and then sets up the thread class loader to use the extracted EAR file directory and the JAR files included in the client JAR manifest. This process is time consuming and because of some limitations with JVM shutdown through Java Native Interface (JNI) and file locking, these files are never cleaned up.

  Specifically, each time the client container launch() method is called, it extracts the EAR file to a random directory name in your temporary directory on your hard drive. The current Java thread class loader is then changed to point to this extracted directory which in turn locks the files within. In a normal J2EE Java client, these files automatically clean up after the application exits. This cleanup occurs when the client container shutdown hook is called (which never happens in the ActiveX to EJB bridge), which leaves the temporary directory there.

  To avoid these problems, you can specify a directory to extract the EAR file by setting the com.ibm.websphere.client.applicationclient.archivedir Java system property before calling the client container launch() method. If the directory does not exist or is empty, you extract the EAR file normally. If the EAR file was previously extracted, the directory is reused. This feature is particularly important for server processes (for example, ASP), which can stop and restart, potentially calling the launchClient() method several times.

  If you need to update your EAR file, delete the temporary directory first. The next time you create the client container object, it extracts the new EAR file to the temporary directory. If you do not delete the temporary directory or change the system property value to point to a different temporary directory, the client container reuses the currently extracted EAR file, and does not use your changed EAR file.

  Note: When specifying the com.ibm.websphere.client.applicationclient.archivedir property, ensure that the directory you specify *is unique* for each EAR file you use. For example, do not point `MyEar1.ear` and `MyEar2.ear` files to the same directory.

  If you choose not to use this system property, go regularly to your Windows `temp` directory and delete the WSTMP* subdirectories. Over a relatively short period of time, these subdirectories can waste a significant amount of space on the hard drive.

Developing applet client code

Applet clients are capable of communicating over the HTTP protocol and the RMI-IIOP protocol.

Before you begin

Applet clients have the following setup requirements:

- These clients are available on the Windows platforms. Check the prerequisites page for information on platform support and product prerequisites.
- The browser installation precedes the client code installation.

About this task

Unlike typical applets that reside on either Web servers or WebSphere Application Servers and can only communicate using the HTTP protocol, applet clients are capable of communicating over the HTTP protocol and the RMI-IIOP protocol. This additional capability gives the applet direct access to enterprise beans.

1. Install the Application Client for WebSphere Application Server.

2. Select the applet client feature.
3. From the IBM Control Panel for Java, enter the following code:

```
-Xmx512M
-Djava.security.policy=<app_client_root>\properties\client.policy
-Dwas.install.root=<app_client_root>
-Djava.ext.dirs=<app_client_root>\java\jre\lib\ext;
<app_client_root>\lib;
<app_client_root>\plugins;
<app_client_root>\lib\ext;
<app_client_root>\installedConnectors\
-Djava.class.path=<app_client_root>\properties
-Dcom.ibm.CORBA.ConfigURL=file:<app_client_root>\properties\sas.client.props
-Dcom.ibm.SSL.ConfigURL=file:<app_client_root>\properties\ssl.client.props
```

Note: The previous entries are automatically placed into the WebSphere Application Server control panel for the Java plug-in user who installed the WebSphere Application Server Application Client. If this sample is being run by a user other than the person who installed the client, the user must enter the entries.

- The Java **Run-Time Parameters** field is similar to the command prompt when using command line options. Therefore, you can enter most options available from the command prompt (for example, -cp, classpath, and others) in this field as well.
- Access the IBM Control Panel for Java from the **Start** menu. Click **Start** > **Control panel** > select the IBM Control Panel for Java.
- The applet container is the Web browser and the Java plug-in combination. You must first install the Applet client feature from the Application Client for WebSphere Application Server so that the browser recognizes the IBM product Java plug-in.

View the Samples gallery for more information about application clients.

Accessing secure resources using SSL and applet clients

By default, the applet client is configured to have security enabled. If you have administrative security turned on at the server from which you are accessing resources, then you can use secure sockets layer (SSL) when needed.

About this task

If you decide that the security requirements for the applet differ from other application client types, then create a new version of the `sas.client.props` and `ssl.client.props` files.

1. Make a copy of the following files so that you can use them for an applet:
   - *<app_client_root>*\properties\sas.client.props
   - *<app_client_root>*\properties\ssl.client.props
2. Edit the copies of the `sas.client.props` and `ssl.client.props` files that you made with your changes.
3. Click **Start** > **Control panel** > select the product Java plug-in to open the Java control panel. To use the files you created in step 1, modify the following values:
   - `-Dcom.ibm.CORBA.ConfigURL=file:`*<app_client_root>*`\properties\sas.client.props`
   - `-Dcom.ibm.SSL.ConfigURL=file:`*<app_client_root>*`\properties\ssl.client.props`

   For more information on the `sas.client.props` and `ssl.client.props` files and WebSphere Application Server security, see the Security section of the information center.

Applet client security requirements

When code is loaded, it is assigned permissions based on the security policy in effect. This policy specifies the permissions that are available for code from various locations. You can initialize this policy from an external policy file.

By default, the client uses the *<app_server_root>*`/properties/client.policy` file. You must update this file with the following permission:

SocketPermission grants permission to open a port and make a connection to a host machine, which is your WebSphere Application Server. In the following example, yourserver.yourcompany.com is the complete host name of your WebSphere Application Server:

```
permission java.util.PropertyPermission "*", "read";
permission java.net.SocketPermission "yourserver.yourcompany.com ,"connect";
```

Example: Applet client tag requirements

Standard applets require the HTML <APPLET> tag to identify the applet to the browser. The <APPLET> tag invokes the Java virtual machine (JVM) of the browser. It can also be replaced by <OBJECT> and <EMBED> tags.

The following code example illustrates the applet code using the <APPLET> tag.

```
<APPLET code="MyAppletClass.class" archive="Applet.jar, EJB.jar" width="600" height="500" >
</APPLET>
```

The following code example illustrates the applet code using the <OBJECT> and <EMBED> tags.

```
<OBJECT classid="clsid: 8AD9C840-044E-11D1-B3E9-00805F499D93"
width="600" height="500">
<PARAM NAME=CODE VALUE=MyAppletClass.class>
<PARAM NAME="archive" VALUE='Applet.jar, EJB.jar'>
<PARAM TYPE="application/x-java-applet;version=1.5.0">
<PARAM NAME="scriptable" VALUE="false">
<PARAM NAME="cache-option" VALUE="Plugin">
<PARAM NAME="cache-archive" VALUE="Applet.jar, EJB.jar">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.5.0" CODE=MyAppletClass.class
ARCHIVE="Applet.jar, EJB.jar" WIDTH="600" HEIGHT="500"
scriptable="false">
<NOEMBED>
</COMMENT>
</NOEMBED>WebSphere Java Application/Applet Thin Client for
Windows is required.
</EMBED>
</OBJECT>
```

Note: The classid and type values changed from WebSphere Application Server version 6.0.2 for version 6.1. Prior to version 6.1, the classid value was clsid:8AE2D840-EC04-11D4-AC77-006094334AA9 and the type value was application/x-websphere-client. In order to successfully invoke the applet client in WebSphere Application Server version 6.1, these values need to be changed to those in the preceding example.

For more information about the applet client tag, see the Sun Microsystems article, http://java.sun.com/j2se/1.5.0/docs/guide/plugin/developer_guide/using_tags.html.

Example: Applet client code requirements

The code used by an applet to talk to an enterprise bean is the same as that used by a stand-alone Java program or a servlet, except for one additional property called java.naming.applet. This property informs the InitialContext and the Object Request Broker (ORB) that this client is an applet rather than a stand-alone Java application or servlet.

When you initialize an instance of the InitialContext class, the first two lines in this code snippet illustrate what both a stand-alone Java program and a servlet issue to specify the computer name, domain, and port. In this example, <yourserver.yourdomain.com> is the computer name and domain where WebSphere Application Server resides, and 900 is the configured port. After the bootstrap values (<yourserver.yourdomain.com>:900) are defined, the client to server communications occur within the underlying infrastructure. In addition to the first two lines for applets, you must add the third line to your code, which identifies this program as an applet, for example:

```
prop.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
prop.put(Context.PROVIDER_URL, "iiop://<yourserver.yourdomain.com>:900)
prop.put(Context.APPLET, this);
```

Developing Java EE application client code

This topic provides the steps required to develop Java Platform, Enterprise Edition (Java EE) application client code.

About this task

A *Java EE application client* program operates similarly to a standard Java EE program in that it runs its own ASCII Java virtual machine code and is invoked at its main method. This JVM run-time environment is part of the client container, which provides the following services for the application client:

- Security
- Communications protocol support (for RMI/IIOP, HTTP, and so on)
- Naming support

The Java Virtual Machine application client program differs from a standard Java program because it uses the Java Naming and Directory Interface (JNDI) namespace to access resources. In a standard Java program, the resource information is coded in the program.

Storing the resource information separately from the client application program makes the client application program portable and more flexible.

1. Write the client application program. Write the Java EE application client program on any development machine. At this stage, you do not require access to the WebSphere Application Server.

   Rules: If you are writing a client application program that will run on z/OS, the following rules apply:
   - Client programs may start their own transactions but cannot join in or start transactions in the WebSphere Application Server for z/OS run-time.
   - Application client code must contain a main method.
   - All input and output files for the application client must be in ASCII, because the client run-time runs in an ASCII JVM.

   Using the `javax.naming.InitialContext` class, the client application program uses the look-up operation to access the Java Naming and Directory Interface (JNDI) namespace. The `InitialContext` class provides the `lookup` method to locate resources.

   The following example illustrates how a client application program uses the `InitialContext` class:

```
import javax.naming.*

public class myAppClient
{
    public static void main(String argv[])
    {
        InitialContext initCtx = new InitialContext();
        Object homeObject = initCtx.lookup("java:comp/env/ejb/BasicCalculator");
        BasicCalculatorHome bcHome = (BasicCalculatorHome)
        javax.rmi.PortableRemoteObject.narrow(homeObject, BasicCalculatorHome.class);
        BasicCalculatorHome bc = bcHome.create();                              ...
    }
}
```

   In this example, the program looks up an enterprise bean called `BasicCalculator`. The `BasicCalculator` Enterprise JavaBeans (EJB) reference is located in the client JNDI namespace at `java:comp/env/ejb/BasicCalculator` . Since the actual Enterprise Java Bean run on the server, the application client run time returns a reference to the `BasicCalculator` home interface.

   If the client application program lookup was for a resource reference or an environment entry, then the look up function returns an instance of the configured type as defined by the client application

deployment descriptor. For example, if the program lookup was a JDBC data source, the lookup would return an instance of `javax.sql.DataSource`. Although you can edit deployment descriptor files, do not use the administrative console to modify them.

2. Assemble the application client using an assembly tool.

   The JNDI namespace knows what to return on a lookup because of the information assembled by the assembly tool.

   Assemble the Java EE application client on any development machine with the assembly tool installed.

   When you assemble your application client, provide the application client run time with the required information to initialize the execution environment for your client application program. Refer to documentation for the assembly tool for implementation details.

   Remember following when you configure resources used by your client application program:
   - Resource environment references are different than resource references. Resource environment references allow your application client to use a logical name to look-up a resource bound into the server JNDI namespace. A resource reference allows your application to use a logical name to look up a local Java EE resource. The Java EE specification does not specify a particular implementation of a resource. The following table contains supported resource types and identifies the resources to which the WebSphere Application Server provides a client implementation.

Resource Type	Client Configuration Notes	Client implementation provided by WebSphere Application Server
javax.sql.DataSource	Supports specification of any data source implementation class	No
java.net.URL	Supports specification of custom protocol handlers	Provided by Java Runtime Environment files
javax.mail.Session	Supports custom protocol configuration	Yes - POP3, SMTP, IMAP
javax.jms.QueueConnectionFactory, javax.jms.TopicConnectionFactory, javax.jms.Queue, javax.jms.Topic	Supports configuration of WebSphere embedded messaging, IBM MQ Series and other JMS providers	Yes - WebSphere embedded messaging

3. Assemble the Enterprise Archive (EAR) file.

   The application is contained in an enterprise archive or `.ear` file. The `.ear` file is composed of:
   - Enterprise bean, application client, and user-defined modules or `.jar` files
   - Web applications or `.war` files
   - Metadata describing the applications or application `.xml` files

   You must assemble the `.ear` file on the server machine.

4. Distribute the EAR file.

   The client machines configured to run this client must have access to the `.ear` file.

   If all the machines in your environment share the same image and platform, run the Application Client Resource Configuration Tool (ACRCT) on one machine to configure the external resources, and then distribute the configured `.ear` file to the other machines.

   If your environment is set up with a variety of client installations and platforms, run the ACRCT for each unique configuration.

   You can either distribute the `.ear` files to the correct client machines, or make them available on a network drive.

   Distributing the `.ear` files is the responsibility of the system and network administrator.

5. Deploy the application client.
6. Configure the application client resources.

If the client application defines the local resources, run the ACRCT (`clientConfig` command) on the local machine to reconfigure the `.ear` file. Use the ACRCT to change the configuration. For example, the `.ear` file can contain a DB2 resource, configured as `C:\DB2`. If, however, you installed DB2 in the `D:\Program Files\DB2` directory, use the ACRCT to create a local version of the `.ear` file.

7. Deploy the application client.

   If you plan to deploy the client on z/OS, you have two options for running the Application Client Resource Configuration Tool (ACRCT):
   - Run the ACRCT on Windows, or
   - Review "Starting the Application Client Resource Configuration Tool and opening an EAR file" on page 298.

   Both of these options produce equivalent output; only the tool interfaces are different. The ACRCT on Windows presents a graphical user interface, whereas the ACRCT for z/OS uses a scripting interface.
8. Use the WebSphere Administrative console to install the application client on z/OS.

What to do next

After completing these steps, launch the application client.

Java EE application client class loading

When you run your Java Platform, Enterprise Edition (Java EE) application client, a hierarchy of class loaders is created to load classes used by your application.

The following list describes the hierarchy of class loaders:
- The Application Client for WebSphere Application Server (Application Client) run time sets this value to the `WAS_LOGGING` environment variable.
- The *extensions class loader* class loader is a child to the bootstrap class loader. This class loader contains JAR files in the `java/jre/lib/ext` directory or those JAR files defined by the `-Djava.ext.dirs` parameter on the Java command. The Application Client client run time does not set `-Djava.ext.dirs` parameters. So it uses the JAR files in the `java/jre/lib/ext` directory.
- The *system class loader* class loader contains JAR files and classes that are defined by the `-classpath` parameter on the Java command. The Application Client run time sets this parameter to the `WAS_CLASSPATH` environment variable.
- The *WebSphere class loader* class loader loads the Application Client run time and any classes placed in the Application Client user directories. The directories used by this class loader are defined by the `WAS_EXT_DIRS` environment variable. The `WAS_BOOTCLASSPATH`, `WAS_CLASSPATH`, and the `WAS_EXT_DIRS` environment variables are set in the *app_server_root*`/bin/setupCmdLine` script for WebSphere Application Server installations, or in the *app_server_root*`/bin/setupClient` script for client installations.

As the Java EE application client run time initializes, additional class loaders are created as children of the WebSphere class loader. If your client application uses resources such as Java DataBase Connectivity (JDBC) API, Java Message Service (JMS) API, or Uniform Resource Locator (URL), a different class loader is created to load each of those resources. Finally, the Application Client run time sets the WebSphere class loader to load classes within the `.ear` file by processing the client JAR manifest repeatedly. The system class path, defined by the CLASSPATH environment variable is never used and is not part of the hierarchy of class loaders.

To package your client application correctly, you must understand which class loader loads your classes. When the Java code loads a class, the class loader used to load that class is assigned to it. Any classes subsequently loaded by that class will use that class loader or any of its parents, but it will not use children class loaders.

In some cases the Application Client run time can detect when your client application class is loaded by a different class loader from the one created for it by the Application Client run time. When this detection occurs, you see the following message:

WSCL0205W: The incorrect class loader was used to load [0]

This message occurs when your client application class is loaded by one of the parent class loaders in the hierarchy. This situation is typically caused by having the same classes in the .ear file and on the hard drive. If one of the parent class loaders locates a class, that class loader loads it before the Application Client run time class loader. In some cases, your client application still functions correctly. In most cases, however, you receive ″class not found″ exceptions.

Configuring the classpath fields

When packaging your Java EE client application, you must configure various class path fields. Ideally, you should package everything required by your application into your .ear file. This is the easiest way to distribute your Java EE client application to your clients. However, you should not package such resources as JDBC APIs, JMS APIs, or URLs. In the case of these resources, use class path references to access those classes on the hard drive. You might also have other classes installed on your client machines that you do not need to redistribute. In this case, you also want to use classpath references to access the classes on the hard drive, as described below.

Referencing classes within the EAR file

WebSphere product Java EE applications do not use the system class path. Use the MANIFEST Class path entry to refer to other JAR files within the .ear file. Configure these values using an assembly tool. For example, if your client application needs to access the path of the EJB JAR file, add the deployed enterprise bean module name to your application client class path. The format of the Class path field for each of the different modules (Application Client, EJB, Web) is the same:
- The values must refer to .jar and .class files that are contained within the .ear file.
- The values must be relative to the root of the .ear file.
- The values cannot refer to absolute paths in the file systems.
- Multiple values must be separated by spaces, not colons or semicolons.

Note: This is the Java method for allowing applications to function platform independent.
Typically, you add modules (.jar files) to the root of the .ear file. In this case, you only need to specify the name of the module (.jar file) in the Class path field. If you choose to add a module with a path, you need to specify the path relative to the root of the .ear file.

For referencing .class files, you must specify the directory relative to the root of the .ear file. With an assembly tool, you can add individual class files to the .ear file. It is recommended that these additional class files are packaged in a .jar file. Add this .jar file to the module Class path fields. If you add .class files to the root of the .ear file, add **./** to the module Class path fields.

Consider the following example directory structure in which the file myapp.ear contains an application client JAR file named myclient.jar and a mybeans.jar EJB module. Additional classes reside in class1.jar and utility/class2.zip files. A class named xyz.class is not packaged in a JAR file but is in the root of the EAR file. Specify **./ mybeans.jar utility/class2.zip class1.jar** as the value of the Classpath property. The search order is: myapp.ear/myclient.jar myapp.ear/xyz.class myapp.ear/mybeans.jar myapp.ear/utility/class2.zip myapp.ear/class1.jar

Referencing classes that are not in the EAR file

Use the launchClient -CCclasspath parameter. This parameter is specified at run time and takes platform-specific class path values, which means multiple values are separated by semi-colons or colons. The client and the server are similar in this respect.

Resource class paths

When you configure resources used by your client application using the Application Client Resource Configuration Tool (ACRCT), or the z/OS ACRCT scripting tool, you can specify class paths that are required by the resource. For example, if your application is using a JDBC to a DB2 database, add `db2java.zip` to the class path field of the database provider. These class path values are platform-specific and require semi-colons or colons to separate multiple values.

On WebSphere Application Server for i5/OS®, if you use the IBM Developer Kit for Java JDBC provider to access DB2/400, you do not have to add the `db2_classes.jar` file to the class path. However, if you use the IBM Toolbox for Java JDBC provider, specify the location of the `jt400.jar` file.

Using the launchClient API

If you use the `launchClient` command, the WebSphere class loader hierarchy is created for you. However, if you use the `launchClient` API, you must perform this setup yourself. Copy the `launchClient` shell command in defining the Java system properties.

Assembling application clients

Application client projects contain programs that run on networked client systems. An application client project is deployed as a Java archive (JAR) file.

About this task

Assemble a client module to contain application client code. Group enterprise beans, Web components, and resource adapter code in separate modules.

Use an assembly tool to assemble an application client module in any of the following ways:
- Import an existing application client JAR file.
- Create a new application client module.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules. Ensure that **Java EE** capability is enabled.
3. Migrate application client JAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an assembly tool. To migrate files, import your application client JAR files to the assembly tool.
4. Create a new application client.
5. Verify the contents of the new application client in either of the following ways:
   - In the Project Explorer view, expand **Application Client Projects** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the application client module in a Navigator view.

What to do next

After you finish assembling all of your application's modules, you are ready to deploy your application.

To deploy your application, refer to "Deploying J2EE application clients on workstation platforms" on page 292.

For more information, see the online help for the assembly tool.

Running the Pluggable application client code

This topic provides steps to install and use the Pluggable application client.

Before you begin

WebSphere Application Server supports the pluggable client.

Note: As you prepare to install the pluggable application client, remember that pluggable clients are only available on Windows systems.

Both J2EE application clients and Thin application clients can access JMS resources provided by the default messaging provider.

1. Install the pluggable application client by selecting option **Pluggable Application Client** from the **Custom client installation** panel.
2. Set the Java application pluggable client environment by using the **setupClient** command, located in:
   app_client_root\AppClient\bin\setupClient.bat
3. Add your specific Java client application JAR files to the CLASSPATH and start your Java client application from this environment, after setting the environment variables.
4. Run a Java command to invoke your client application.

Example

View the Samples Gallery for more information about the Application Client.

Running Thin application client code

You can run Thin application client environments on machines installed with either a client installation or a server installation. The client installation provides a setup command shell which sets up your environment for either a Thin application client or a Java Platform, Enterprise Edition (Java EE) client application. The server installation provides a command shell which sets up your environment for Java EE application clients only.

Before you begin

Both Java EE application clients and Thin application clients can access JMS resources provided by the default messaging provider.

About this task

WebSphere Application Server supports the pluggable client.

Note: Thin application clients are not packaged with JDBC provider classes. For example, the WebSphere Application Server Version 7.0 Thin application client is not packaged with Apache Derby 10.2 classes. Likewise, the version 6.1 Thin application client is not packaged with Cloudscape® Version 5.1, Cloudscape Version 10.0, or Cloudscape version 10.1 classes. Therefore, to utilize the JDBC provider classes (such as Apache Derby, Oracle, DB2, Informix®, or Sybase) on a thin client, you must:

1. Add the classes to your Thin application client environment.
2. Make the classes visible to the Thin application client. To do this, add the path to the classes in the client classpath within the script that launched the client program.

Otherwise, any attempt to load a database class (such as through the JNDI lookup of a datasource) results in a ClassNotFoundException.

The Java invocation to run a Thin application client varies between a client and a server. If your Thin application client needs to run on both a client installation and a server installation, follow the steps for developing Thin application clients on a server machine.

1. Install the Thin application client by selecting option **Java EE and Thin application client** from the Application Client for WebSphere Application Server installation.
2. Perform one of the following:
   - Develop Thin application client code for a client machine.
   - Develop Thin application client code for a server machine.

Example

View the Samples gallery for more information about the Application Client.

Note: WebSphere Application Server and the Application client for WebSphere Application Server also provides specialized types of Thin application client. The Web Services Thin Client is an unmanaged, stand-alone Java client environment that allows you to run JAX-RPC Web services client applications to invoke Web services that are hosted by the application server. The Administration Thin Client enables client applications to perform administration tasks outside of the application server environment.

Running Thin application client code on a client machine

This topic provides the steps necessary to run Thin application client code on a client machine.

Before you begin

You must install the Thin application client from the Application Client for WebSphere Application Server installation before performing this task. For more information, see Developing Thin application client code.

1. Set up the Thin application client environment. Run the setupClient command.

   app_client_root/AppClient/bin/setupClient.sh

2. Compile your client application. Run the Java compilation command.

```
$JAVA_HOME/bin/javac -classpath "$WAS_CLASSPATH:
<list_of_your_application_jars_and_classes>" -extdirs $WAS_EXT_DIRS
<your_application_class>.java
```

   •

Example

View the Samples gallery for more information about the Application Client.

Running Thin application client code on a server machine

This topic provides the steps necessary to run Thin application client code on a server machine.

Before you begin

You must install WebSphere Application Server before performing this task.

1. Set up the Thin application client environment.

   Use the setupCmdLine shell.

   app_server_root/bin/setupCmdLine.sh

2. Run the Java compilation command to compile your client application.

```
$JAVA_HOME/bin/javac -classpath "$WAS_CLASSPATH:
<list_of_your_application_jars_and_classes>" -extdirs $WAS_EXT_DIRS
<your_application_class>.java
```

3. Run the application client.

   Perform one of the following methods:
   - Run a Java command to call your main class directly.

```
"$JAVA_HOME/bin/java" $WAS_LOGGING
-Djava.security.auth.login.config="$USER_INSTALL_ROOT/properties/wsjaas_client.conf"
-Djava.ext.dirs="$JAVA_HOME/jre/lib/ext:$WAS_EXT_DIRS:$WAS_HOME/plugins: $WAS_HOME/lib/WMQ/java/lib"
-Djava.naming.provider.url=<an_IIOP_URL_or_a_corbaloc_URL_to_your
application_server_machine_name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
-Dserver.root="$WAS_HOME" $USER_INSTALL_PROP "$CLIENTSAS" "$CLIENTSSL"
-classpath "$WAS_CLASSPATH;<list_of_your_application_jars_and_classes>"
<fully_qualified_class_name_to_run> <your_application_parameters>
```

For more information on IIOP and `corbaloc` URLs, see Developing applications that use JNDI.

- Enter a command to use the WebSphere Application Server launcher.

```
"$JAVA_HOME/bin/java" $WAS_LOGGING
-Djava.security.auth.login.config="$USER_INSTALL_ROOT/properties/wsjaas_client.conf"
 "-Dws.ext.dirs=<list_of_your_application_jars_and_classes>
$WAS_EXT_DIRS;$WAS_USER_DIRS"
-Djava.naming.provider.url=<an_IIOP_URL_or_a_corbaloc_URL_to_your
application_server_machine_name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
"-Dserver.root=$WAS_HOME"
"$CLIENTSAS" "$CLIENTSSL" $USER_INSTALL_PROP -classpath "$WAS_CLASSPATH"
com.ibm.ws.bootstrap.WSLauncher
<fully_qualified_class_name_to_run> <your_application_parameters>
```

Example

View the Samples gallery for more information about the Application Client.

Deploying J2EE application clients on workstation platforms

You can deploy the J2EE application clients on workstation platforms using the methods described in this topic.

Before you begin

After developing an application client, deploy this application on client machines. *Deployment* consists of pulling together the various artifacts that the application client requires.

The *Application Client Resource Configuration Tool* (ACRCT) defines resources for the application client. These configurations are stored in the client `.jar` file within the application `.ear` file. The application client run time uses these configurations for resolving and creating an instance of the resources for the application client.

If you plan to deploy the client on z/OS, run the Application Client Resource Configuration Tool (ACRCT) on Windows. You can also run the ACRCT for distributed platforms locally. If you need to install the ACRCT, see Installing Application Client for WebSphere Application Server.

Note: This task only applies to J2EE application clients. Only perform this task if you configured your J2EE application client to use resource references.

About this task

1. Start the ACRCT and open an EAR file.
2. Configure new data source providers.
3. Configure mail providers and sessions.
4. Configure URL providers and sessions.
5. Configure Java messaging resources.
6. Configure new environment entries.

7. (Optional) Remove application client resources.
8. Save the EAR file.

Resource Adapters for the client

A resource adapter is a system-level software driver that a Java application uses to connect to an enterprise information system (EIS). A resource adapter plugs into an application client and provides connectivity between the EIS and the enterprise application.

The resource adapter support for the J2EE client applications is a subset of the support for the server. For any resource adapter installed using the clientRAR tool, the client resource adapter is used in a non-managed environment and must conform to the J2EE Connector Architecture Specification Version 1.5 or higher. Only outbound connections to the EIS are supported through the ManagedConnectionFactory interfaces. The inbound messaging support (from the EIS), life cycle management, and work management aspects of the specification are not supported on the client.

For a client application to use a resource adapter, it must be installed in the directory specified by the environment variable, CLIENT_CONNECTOR_INSTALL_ROOT, defined when the setupCmdLine script runs. The launchClient tool, Application Client Resource Configuration Tool (ACRCT) and clientRAR tool all use this variable to find the default location of all installed resource adapters. To install a resource adapter in the client, use the clientRAR tool. Once the resource adapter is installed, it must be configured using the ACRCT. The client configuration tool adds the resource adapter configuration to the EAR file. Then, connection factories and administered objects are defined.

When running J2EE application clients, the `launchClient` script specifies a system property called `com.ibm.ws.client.installedConnector`, which is set to the same value as the `CLIENT_CONNECTOR_INSTALL_ROOT` variable. This is the default location for installed resource adapters and can be overridden for each `launchClient` call by specifying the `-CCD` parameter. When the client container is activated, all resource adapter subdirectories under the specified default location for the resource adapters directory are added to the classpath. This action allows the client application to use the resource adapters without using the ACRCT to specify any of the client resources.

Using resource adapters is a new mechanism for easily extending client applications.

Configuring resource adapters

Use the Application Client Resource Configuration Tool (ACRCT) to configure resource adapters.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure new resource adapters. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new resource adapters from the tree.
4. Expand the JAR file to view its contents.
5. Right-click the **Resource Adapters** folder, and click **New**.
6. Configure the resource adapter settings in the resulting property dialog.
7. Click **OK**.
8. Click **File** > **Save** on the menu bar to save your changes.

clientRAR tool

This topic describes the command line syntax for the client resource adapter installation tool.

If this tool is used to add or delete resource adapters on the server, then only the client can use the resource adapter. If the resource adapter is installed on the server using the wsadmin tool or the administrative console, then do not use the clientRAR tool remove it. Only resource adapters that are installed using the clientRAR tool should be removed using the clientRAR tool.

The command line invocation syntax for the clientRAR tool follows:

```
clientRAR [-help | -?] [-CRDcom.ibm.ws.client.installedConnectors=<dir>] <task> <archive>

where
-help, -?
Print the usage information.
-CRDcom.ibm.ws.client.installedConnectors
The directory where resource adapters are installed.
This will override the system property of the same name
(com.ibm.ws.client.installedConnectors).


<task>
The task to perform: add - install, delete - uninstall.

<archive>
if task=add then this is the fully qualified name of the resource adapter archive file.
If task=delete then this is the filename of the resource adapter archive to be uninstalled.
```

The following examples demonstrate correct syntax.

Configuring new connection factories for resource adapters

Use the Application Client Resource Configuration Tool (ACRCT) to configure new connection factories for resource adapters.

About this task

Complete this task to configure new connection factories for resource adapters.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure new connection factories. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new connection factories from the tree.
4. Expand the JAR file to view its contents.
5. Click the **Resource Adapters** folder.
6. Expand the resource adapter for which you want to create connection factories.
7. Right-click the **Connection Factories** folder and click **New**.
8. Configure the connection factory properties in the resulting property dialog.
9. Click **OK**.
10. Click **File** > **Save** on the menu bar to save your changes.

Resource adapter connection factory settings:

Use this panel to view or change the configuration properties of the selected resource adapter connection factory.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Adapters**. Right-click the **Connection Factories** folder, and click **New**. The following fields appear on the **General** tab.

Name:

The name by which this connection factory is known for administrative purposes within WebSphere Application Server. The name must be unique within the resource adapter connection factories across the product administrative domain.

Data type String

Description:

An optional description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String

JNDI Name:

The JNDI name that is used to match this resource adapter connection factory definition to the deployment descriptor. This entry should be a `resource-ref` name.

Data type String

User Name:

The **User Name** used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly when getting a connection. If this field is used, then the Properties field `UserName` is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User Name** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly when getting a connection.

Data type String

Password:

Specifies an encrypted password. If you complete this field, then the **Password** field in the Properties box is ignored.

If you specify a value for the **UserName** property, you must also specify a value for the **Password** property.

Data type String

Re-Enter Password:

Confirms the password.

Type:

A drop-down list of all the `connectionFactoryInterfaces` as defined for the factories in the Resource Adapter Archive.

For each **Type**, there is a set of properties specified in the Properties box. This set of properties is constructed by retrieving the properties from each connection definition object. For any existing connection factories that are displayed for updating, this list of properties is overlaid with the properties specified for the objects. When the **Type** field is changed, the properties also change to reflect the correct properties for that type.

Data type String

Configuring administered objects

This section helps you configure new administered objects.

Before you begin

Before you configure new administered objects, you must complete the following prerequisites:

1. Install the Resource Adapter Archive file (RAR) using the clientRAR tool.
2. Configure the resource adapter for the `.ear` file, using the Application Client Resource Configuration Tool (ACRCT) tool.

About this task

Complete this task to configure new administered objects for installed resource adapters.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure new administered objects. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new administered objects from the tree.
4. Expand the JAR file to view its contents.
5. Click the **Resource Adapters** folder.
6. Expand the resource adapter for which you want to create administered objects.
7. Right-click the **Administered Objects** folder and click **New**.
8. Configure the administered object properties in the resulting property dialog.
9. Click **OK**.
10. Click **File** > **Save** on the menu bar to save your changes.

Administered objects settings:

Use this panel to view or change the configuration properties of the selected administered objects.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Adapters** > *resource_adapter_instance*. Right-click **Administered Objects** and click **New**. The following fields appear on the **General** tab.

The settings for administered objects are handled similarly to connection factories. When updating administered objects, use the same panels that you used to create administered objects.

Name:

The name by which this administered object is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the resource adapter administered objects across the product administrative domain.

Data type String

Description:

An optional description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String

JNDI Name:

This entry is a `resource-env-ref` name, a `message-destination-ref` name (if the `message-destination-ref` has no link), or a `message-destination` link.

Data type String

Type:

A drop-down list of all the administered object class-interface pairs as defined for the admin objects in the Resource Adapter Archive (RAR) file.

For each **Type**, there is a set of properties specified in the Properties box. This set of properties is constructed by retrieving the properties from each administered object definition. For any existing administered objects that are displayed for updating, this list of properties is overlaid with the properties specified for the objects. When the **Type** field is changed, the properties also change to reflect the correct properties for that type.

Data type String

Resource adapter settings

Use this panel to view or change the configuration properties of the resource adapter. These configuration properties control how resource adapters are created.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Adapter**. Right-click **Resource Adapter** and click **New**. The following fields appear on the **General** tab.

Name

The name by which this Resource Adapter is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the Resource Adapters across the product administrative domain.

Data type String

Description

A description of this resource adapter for administrative purposes within IBM WebSphere Application Server.

Data type String

Class Path

Any additional class path. The path to the resource adapter directory is automatically added.

Data type String
Default The path to your Resource Adapter directory.

Native Path

The native path where the Resource Adapter is located. Enter any additional native class path here.

Data type String

Resource Adapter Name
A mandatory field that points to an installed resource adapter subdirectory. The entry does not represent the full directory name for the resource adapter. The full directory name is the installed resource adapter path, plus the resource adapter name.

Data type String

Installed Resource Adapter Path
The directory where resource adapters are installed. If you do not complete this field, then the default takes effect.

If you specify the value, `${CONNECTOR_INSTALL_ROOT}`, then this value replaces the value of the `CLIENT_CONNECTOR_INSTALL_ROOT` variable on the machine on which the client application runs. This action allows the application to run easily on different machines, where the client installation might be in different locations.

Data type String
Default ${CONNECTOR_INSTALL_ROOT}

Starting the Application Client Resource Configuration Tool and opening an EAR file
You can perform many tasks by starting the Application Client Resource Configuration Tool (ACRCT). Many of these tasks also involve then opening an EAR file.

Before you begin

Note: This task only applies to J2EE application clients.

About this task

Use these steps to start the Application Client Resource Configuration Tool. When you start the tool, one of the most common tasks that you perform is opening and modifying the components of EAR files.

1. Open a command prompt and change to the *app_server_root*`\bin` directory.
2. Run the `clientConfig.bat` file.

   Note: When running the `clientConfig.bat` file on the Microsoft® Windows Vista™ operating system, run the script using the administrator account that was created when the Windows Vista operating system was installed.
3. Open an EAR file within the Application Client Resource Configuration Tool (ACRCT):
   a. Click **File** > **Open**.
   b. Select the file and click **Open**.
   •
4. Save your changes to the file and close the tool:
   a. Click **File** > **Save**.
   b. Click **File** > **Exit**.
   •

Data sources for the Application Client
WebSphere Application Server and the Application Client for WebSphere Application Server do not provide client database drivers to be used directly from a J2EE application client. If your application client accesses a database directly, you must provide the database drivers on the client machine.

You can contact your database vendor to acquire client database driver code and licenses. In addition, data sources configured on the server and looked up on the client do not participate in global transactions. Instead of accessing the database directly, it is recommended that your client application use an enterprise bean. Accessing a database through an enterprise bean eliminates the need to have database drivers on the client machine because the database access is handled by the enterprise bean running on WebSphere Application Server. For a current list of providers that are supported on WebSphere Application Server visit the WebSphere Application Server prerequisite Web site. (See the following link.)

Data source properties for application clients

Use this page to create or modify the data sources.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Data Source Providers** > *Data source provider instance*. Right-click **Data Sources** and click **New**. The following fields are displayed on the **General** tab:

Name

Specifies the display name of this data source.

Data type String

Description

Specifies a text description of the data source.

Data type String

JNDI Name

The application client run time uses this field to retrieve configuration information.

Database Name

The name of the database to which you want to connect.

User

Use the user ID with the Password property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the User ID property, then you must also specify a value for the Password property. The connection factory User ID and Password properties are used if the calling application does not provide a user ID and password explicitly.

Password

Use the password with the User ID property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the Password property, then you must also specify a value for the User ID property.

Re-Enter Password

Confirms the password.

Custom Properties

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new data source providers (JDBC providers) for application clients

You can create new data source providers, also known as JDBC providers, for your application client using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create new data source providers, also known as JDBC providers, for your application client. In a separate administrative task, install the Java code for the required data source provider on the client machine on which the application client resides.

About this task

Use this task to connect application clients to relational databases.

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file for which you want to configure the new data source provider. The EAR file contents display in a tree view.
2. Select the JAR file in which you want to configure the new data source provider from the tree.
3. Expand the JAR file to view its contents.
4. Click the **Data Source Providers** folder. Do one of the following:
   - Right-click the folder and click **New Provider**.
   - Click **Edit > New** on the menu bar.
5. Configure the data source provider properties in the resulting property dialog.
6. Click **OK** when you finish.
7. Click **File > Save** on the menu bar to save your changes.

Example: Configuring data source provider and data source settings

You can configure data source provider and data source settings.

The purpose of this article is to help you to configure data source provider and data source settings.

- Required fields:
  - Data Source Provider Properties page: name
  - Data Source Properties page: name, jndiName
- Special cases:
  - The user name and password fields have no equivalent XMI tags. You must specify these fields in the custom properties.
  - The password is encrypted when you use the Application Client Resource Configuration Tool (ACRCT). If you do not use the ACRCT the field cannot be encrypted.
- Example:

```
<resources.jdbc:JDBCProvider xmi:id="JDBCProvider_1" name="jdbcProvider:name"
description="jdbcProvider:description" implementationClassName="jdbcProvider:
ImplementationClass">
<classpath>jdbcProvider:classPath</classpath>
<factories xmi:type="resources.jdbc:WAS40DataSource" xmi:id="WAS40DataSource_1"
name="jdbcFactory:name" jndiName="jdbcFactory:jndiName"
description="jdbcFactory:description" databaseName="jdbcFactory:databasename">
<propertySet xmi:id="J2EEResourcePropertySet_13">
<resourceProperties xmi:id="J2EEResourceProperty_13" name="jdbcFactory:customName"
value="jdbcFactory:customValue"/>
<resourceProperties xmi:id="J2EEResourceProperty_14" name="user"
value="jdbcFactory:user"/>
<resourceProperties xmi:id="J2EEResourceProperty_15" name="password"
```

```
value="{xor}NTs9PBk+PCswLSZlMT4yOg==">
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_14">
<resourceProperties xmi:id="J2EEResourceProperty_16" name="jdbcProvider:customName"
value="jdbcProvider:customeValue"/>
</propertySet>
</resources.jdbc:JDBCProvider>
```

Data source provider settings for application clients

Use this page to create a data source under a JDBC provider which provides the specific JDBC driver implementation class.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Data Source Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the display name for the data source.

For example you can set this field to *Test Data Source*.

Data type String

Description:

Specifies a text description for the resource.

Data type String

Class Path:

A list of paths or `.jar` file names which together form the location for the resource provider classes.

Implementation class:

Use this setting to perform database specific functions.

Data type String
Default Dependent on JDBC driver implementation class

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new data sources for application clients

About this task

During this task, you create new data sources for your application client.

1. Click the data source provider for which you want to create a data source in the tree. Take one of the following actions as needed:
   - Configure a new data source provider.
   - Click an existing data source provider.
2. Expand the data source provider to view its **Data Sources** folder.
3. Click the data source folder. Take one of the following actions as needed:
   - Right click the data source folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the data source properties in the displayed fields.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring mail providers and sessions for application clients

You can edit the configurations of mail sessions and providers for your application clients using the Application Client Resource Configuration Tool (ACRCT).

About this task

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of mail sessions and providers for your application clients to use.

1. Start the ACRCT.
2. Open an EAR file.
3. Locate the mail objects in the tree that is displayed for the EAR file. For example, if your file contains mail sessions, expand **Resources** → ***application*.jar** → **Mail Providers** → ***java_mail_provider_instance*** → **Mail Sessions**.

   In this example, ***java_mail_provider_instance*** is a particular mail provider.

Results

The mail session instances are located in the **JavaMail Sessions** folder.

Mail provider settings for application clients

Use this page to implement the JavaMail API and create mail sessions.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Mail Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

The name of the JavaMail resource provider.

Description:

An optional description for the resource provider.

Class Path:

Specifies a list of paths or JAR file names which together form the location for the resource provider classes.

Protocol:

Specifies the name of the protocol.

Classname:

Specifies the name of the class implementing the protocol. Leave this field blank if you want to use the default implementation.

Type:

This menu contains the following two values: `TRANSPORT` or `STORE`.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Mail session settings for application clients

Use this page to configure mail session properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Mail Providers** > *mail provider instance*. Right-click **Mail Sessions** and click **New**. The following fields appear on the **General** tab:

Name:

Represents the administrative name of the JavaMail session object.

Description:

Provides an optional description for your administrative records.

JNDI Name:

The application client run time uses this field to retrieve configuration information.

Mail Transport Host:

Specifies the server to connect to when sending mail.

Mail Transport Protocol:

Specifies the transport protocol to use when sending mail.

Mail Transport User:

Specifies the user ID to use when the mail transport host requires authentication.

Mail Transport Password:

Specifies the password to use when the mail transport host requires authentication.

Enable strict Internet address parsing:

Specifies whether the recipient addresses must be parsed strictly in compliance with RFC 822, which is a specifications document issued by the Internet Architecture Board.

This setting is not generally used for most mail applications. RFC 822 syntax for parsing addresses effectively enforces a strict definition of a valid e-mail address. If you select this setting, JavaMail will adhere to RFC 822 syntax and reject recipient addresses that do not parse into valid e-mail addresses (as defined by the specification). If you do not select this setting, JavaMail will not adhere to RFC 822 syntax and will accept recipient addresses that do not comply with the specification. By default, this setting is deselected. You can view the RFC 822 specification at the following URL for the World Wide Web Consortium (W3C): http://www.w3.org/Protocols/rfc822/.

Re-Enter Password:

Confirms the password.

Mail From:

Specifies the mail originator.

Mail Store Host:

Specifies the mail account host (or ″domain″) name.

Mail Store User:

Specifies the user ID of the mail account.

Mail Store Password:

Specifies the password of the mail account.

Re-Enter Password:

Confirms the password.

Mail Store Protocol:

Specifies the protocol to be used when receiving mail.

Mail Debug:

When `true`, JavaMail interaction with mail servers, along with these mail session properties are printed to the `stdout` file.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring mail provider and mail session settings for application clients

You can configure mail provider and mail session settings. This topic provides the required fields, special cases, and an example.

The purpose of this topic is to help you configure mail provider and mail session settings.

- Required fields:

  - Mail Provider Properties page: name, and at least one protocol provider
  - Mail Session Properties page: name, jndiName, outgoing server and protocol, and/or incoming server and protocol
- Special cases:
  - If you use the ACRCT tool, the password field will be encrypted. You cannot encrypt the password field if you do not use the ACRCT tool.
- Example:

```
<resources.mail:MailProvider xmi:id="builtin_mailprovider" name="Built-in Mail Provider" description="The built-in mail provider">
  <factories xmi:type="resources.mail:MailSession"
             xmi:id="MailSession_1207766754834" name="MailSession"
             jndiName="mail/session" description="Sample mail session" category="Sample"
             mailTransportHost="smtp.coldmail.com" mailTransportUser="transportUser"
             mailTransportPassword="{xor}Lz4sLChvLTs="
             mailFrom="smith@coldmail.com" mailStoreHost="imap.coldmail.com" mailStoreUser="storeUser"
             mailStorePassword="{xor}Lz4sLChvLTs="
             debug="true" strict="true"
             mailTransportProtocol="builtin_smtp" mailStoreProtocol="builtin_imap">
    <propertySet xmi:id="J2EEResourcePropertySet_1207766778585">
      <resourceProperties xmi:id="J2EEResourceProperty_1207766778585" name="key" type="java.lang.String" value="value" required="false"/>
    </propertySet>
  </factories>
  <protocolProviders xmi:id="builtin_smtp" protocol="smtp" classname="com.sun.mail.smtp.SMTPTransport" type="TRANSPORT"/>
  <protocolProviders xmi:id="builtin_pop3" protocol="pop3" classname="com.sun.mail.pop3.POP3Store" type="STORE"/>
  <protocolProviders xmi:id="builtin_imap" protocol="imap" classname="com.sun.mail.imap.IMAPStore" type="STORE"/>
  <protocolProviders xmi:id="builtin_smtps" protocol="smtps" classname="com.sun.mail.smtp.SMTPSSLTransport" type="TRANSPORT"/>
  <protocolProviders xmi:id="builtin_pop3s" protocol="pop3s" classname="com.sun.mail.pop3.POP3SSLStore" type="STORE"/>
  <protocolProviders xmi:id="builtin_imaps" protocol="imaps" classname="com.sun.mail.imap.IMAPSSLStore" type="STORE"/>
</resources.mail:MailProvider>
```

Configuring new mail sessions for application clients

You can use the Application Client Resource Configuration Tool (ACRCT) to configure new mail sessions for your application client.

Before you begin

During this task, you configure new mail sessions for your application client. The mail sessions are associated with the pre-configured default mail provider supplied by the product.

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file. The EAR file contents are displayed in a tree view.
2. Select the JAR file in which you want to configure the new JavaMail session.
3. Expand the JAR file to view its contents.
4. Click **Mail Providers** > **Mail Provider** > **Mail Sessions**. Complete one of the following actions:
   - Right click the **Mail Sessions** folder and select **New Factory**.
   - Click **Edit** > **New** on the menu bar.
5. Configure the Mail Session properties in the displayed fields.
6. Click **OK**.
7. Click **File** > **Save** on the menu bar to save your changes.

URLs for application clients

A *Uniform Resource Locator* (URL) is an identifier that points to an electronically accessible resource, such as a directory file on a machine in a network, or a document stored in a database.

URLs appear in the format *scheme*:*scheme_information*.

You can represent a *scheme* as `http`, `ftp`, `file`, or another term that identifies the type of resource and the mechanism by which you can access the resource.

In a World Wide Web browser location or address box, a URL for a file available using HyperText Transfer Protocol (HTTP) starts with `http:`. An example is `http://www.ibm.com`. Files available using File Transfer Protocol (FTP) start with `ftp:`. Files available locally start with `file:`.

The *scheme_information* commonly identifies the Internet machine making a resource available, the path to that resource, and the resource name. The scheme_information for HTTP, FTP and File generally starts with two slashes (//), then provides the Internet address separated from the resource path name with one slash (/). For example,

`http://www.ibm.com/software/webservers/appserv/library.html`.

For HTTP and FTP, the path name ends in a slash when the URL points to a directory. In such cases, the server generally returns the default index for the directory.

URL providers for the Application Client Resource Configuration Tool

A URL provider implements the function for a particular URL protocol, such as HyperText Transfer Protocol (HTTP). This provider, comprised of a pair of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Configuring new URL providers for application clients

You can create URL providers and URLs for your client application using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create URL providers and URLs for your client application. In a separate administrative task, you must install the Java code for the required URL provider on the client machine on which the client application resides.

About this task

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new URL provider. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new URL provider from the tree.
4. Expand the JAR file to view the contents.
5. Click the folder called **URL Providers**. Complete one of the following actions:
   - Right click the folder and select **New**.
   - Click **Edit** > **New** on the menu bar.
6. Configure the URL provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File** > **Save** on the menu bar to save your changes.

Configuring URL providers and sessions using the Application Client Resource Configuration Tool

You can edit the configurations of URL providers and URLs to be used by your application clients using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of URL providers and URLs to be used by your application clients.

About this task

1. Start the ACRCT.
2. Open an EAR file.

3. Locate the URL objects in the tree that displays. For example, if your file contains URL providers and URLs, expand **Resources** -> ***application*.jar** -> **URL Providers** -> ***url_provider_instance***
   where ***url_provider_instance*** is a particular URL provider.
4. If you expand the tree further, you will also see the **URLs** folders containing the URL instances for each URL provider instance.

URL settings for application clients:

Use this page to implement the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP).

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **URL Providers** > *URL provider instance*. Right-click **URLs** and click **New**. The following fields appear on the **General** tab.

This provider, comprised of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Name:

The administrative name for the URL.

Description:

This is an optional description of the URL for your administrative records.

JNDI Name:

The application client run time uses this field to retrieve configuration information.

URL:

A Uniform Resource Locator (URL) name that points to an Internet or intranet resource. For example: `http://www.ibm.com`.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

URL provider settings for application clients:

Use this page create new URL providers.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right click **URL Providers**, and click **New**. The following fields appear on the **General** tab.

A URL provider implements the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP). This provider, comprised of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Name:

Administrative name for the URL.

Description:

Optional description of the URL, for your administrative records.

Class Path:

A list of paths or JAR file names which together form the location for the resource provider classes.

Protocol:

Protocol supported by this stream handler. For example, `nntp`, `smtp`, `ftp`, and so on.

To use the default protocol, leave this field blank.

Stream handler class:

Fully qualified name of a User-defined Java class that extends the `java.net.URLStreamHandler` for a particular URL protocol, such as FTP.

To use the default stream handler, leave this field blank.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring URL and URL provider settings for application clients

You can configure URL and URL provider settings. This topic provides the required fields and an example.

The purpose of this article is to help you to configure URL and URL provider settings.

- Required fields:
  - URL Properties page: name, jndiName, url
  - URL Provider Properties page: name
- Example:

```
<resources.url:URLProvider xmi:id="URLProvider_1" name="urlProvider:name"
description="urlProvider:description"
streamHandlerClassName="urlProvider:streamHandlerClass"
protocol="urlProvider:protocol">
<classpath>urlProvider:classpath</classpath>
<factories xmi:type="resources.url:URL" xmi:id="URL_1" name="urlFactory:name"
jndiName="urlFactory:jndiName" description="urlFactory:description"
spec="urlFactory:url">
<propertySet xmi:id="J2EEResourcePropertySet_18">
<resourceProperties xmi:id="J2EEResourceProperty_20" name="urlFactory:customName"
value="urlFactory:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_19">
```

```
<resourceProperties xmi:id="J2EEResourceProperty_21" name="urlProvider:customName"
value="urlProvider:customValue"/>
</propertySet>
</resources.url:URLProvider>
```

Configuring new URLs with the Application Client Resource Configuration Tool

You can use URLs for your client application using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create URLs for your client application.

About this task

1. Click the URL provider for which you want to create a URL in the tree. Complete one of the following:
   - Configure a new URL provider.
   - Click an existing URL provider.
2. Expand the URL provider to view the **URLs** folder.
3. Click the URL folder. Complete one of the following actions:
   - Right click the folder and click **New**.
   - Click **Edit -> New** on the menu bar.
4. Configure the URL properties in the displayed fields.
5. Click **OK** when you finish.
6. Click **File** > **Save** in the menu bar to save your changes.

Asynchronous messaging in WebSphere Application Server using JMS

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interface. The JMS interface provides a common way for Java programs (clients and Java Platform, Enterprise Edition (Java EE) applications) to create, send, receive, and read asynchronous requests as JMS messages.

This topic provides a generic overview of asynchronous messaging using the JMS support provided by WebSphere Application Server.

The base support for asynchronous messaging using the JMS API provides the common set of JMS interfaces and associated semantics that define how a JMS client can access the facilities of a JMS provider. This support enables WebSphere product Java EE applications, as JMS clients, to exchange messages asynchronously with other JMS clients, by using JMS destinations (queues or topics). A Java EE application can use JMS queue destinations for point-to-point messaging and JMS topic destinations for publish and subscribe messaging. A Java EE application can explicitly poll for messages on a destination, and then retrieve messages for processing by business logic beans (enterprise beans).

With the base JMS and XA support, the Java EE application uses standard JMS calls to process messages, including any responses or outbound messaging. An enterprise bean can handle responses acting as a sender bean, or within the enterprise bean that receives the incoming messages. Optionally, this process can use two-phase commit within the scope of a transaction. This level of function for asynchronous messaging is called *bean-managed messaging*, and gives an enterprise bean complete control over the messaging infrastructure, for example, connection and session pool management. The common container has no role in bean-managed messaging.

WebSphere Application Server also supports automatic asynchronous messaging using message-driven beans (a type of enterprise bean defined in the Enterprise JavaBeans (EJB) 2.0 specification) and JMS listeners (part of the JMS application server facilities). Messages are automatically retrieved from JMS

destinations, optionally within a transaction, then sent to the message-driven bean in a Java EE application, without the application having to explicitly poll JMS destinations.

Java Message Service providers for clients

This topic describes the different ways client applications can use Java Message Service (JMS) providers with WebSphere Application Server.

IBM WebSphere Application Server supports asynchronous messaging through the use of a JMS provider and its related messaging system. JMS providers must conform to the JMS specification version 1.1. To use message-driven beans the JMS provider must support the optional Application Server Facility (ASF) function defined within that specification, or support an inbound resource adapter as defined in the JCA specification version 1.5.

The service integration technologies of IBM WebSphere Application Server can act as a messaging system when you have configured a service integration bus that is accessed through the default messaging provider. This support is installed as part of WebSphere Application Server, administered through the administrative console, and is fully integrated with the WebSphere Application Server runtime.

WebSphere Application Server also includes support for the following JMS providers:

WebSphere MQ
Provided for use with supported versions of WebSphere MQ.

Generic
Provided for use with any 3rd party messaging system which supports ASF.

For backwards compatibility with earlier releases, WebSphere Application Server also includes support for the V5 default messaging provider which enables you to configure resources for use with the WebSphere Application Server version 5 Embedded Messaging system. The V5 default messaging provider can also be used with a service integration bus.

WebSphere applications can use messaging resources provided by any of these JMS providers. However the choice of provider is most often dictated by requirements to use or integrate with an existing messaging system. For example, you may already have a messaging infrastructure based on WebSphere MQ. In this case you may either connect directly using the included support for WebSphere MQ as a JMS provider, or configure a service integration bus with links to a WebSphere MQ network and then access the bus through the default messaging provider.

The service integration bus also provides access to a default messaging provider. This is a J2EE 1.4 compliant JMS messaging provider which is fully integrated with WebSphere Application Server. You can use it in multiple server configurations for messaging interactions with a WebSphere MQ network.

Configuring Java messaging client resources

To configure Java messaging client resources, you create new JMS provider configurations for your application client. The application client can use a messaging service through the Java Message Service APIs. A JMS provider provides two kinds of J2EE factories. One is a *JMS connection factory*, and the other is a *JMS destination factory*.

Before you begin

In a separate administrative task, install the Java Message Service (JMS) client on the client machine where the application client resides. The messaging product vendor must provide an implementation of the JMS client. For more information, see your messaging product documentation.

Note: When completing this task, you can either create a new messaging provider, or you can use an existing one.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new JMS provider. The EAR file contents are in the displayed tree view.
3. Select the JAR file in which you want to configure the new JMS provider from the tree.
4. Expand the JAR file to view its contents.
5. Optionally right-click **Messaging Providers** and select **New**, if you want to create and use a new messaging provider.
6. Configure the JMS provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File** > **Save**.

Configuring new JMS providers with the Application Client Resource Configuration Tool

You can create new Java Message Service (JMS) provider configurations for the Application Client. The Application Client makes use of a messaging service through the JMS interfaces.

Before you begin

During this task, you create new Java Message Service (JMS) provider configurations for the Application Client. The Application Client makes use of a messaging service through the JMS interfaces. A JMS provider provides two kinds of J2EE resources. One is a JMS connection factory, and the other is a JMS destination.

In a separate administrative task, you must install the JMS client on the client machine where your particular application client resides. The messaging product vendor must provide an implementation of the JMS client. For more information, see your messaging product documentation.

About this task

1. Start the Application Client Resource Configuration Tool and open the EAR file for which you want to configure the new JMS provider. The EAR file contents are displayed in a tree view.
2. From the tree, select the JAR file in which you want to configure the new JMS provider.
3. Expand the JAR file to view its contents.
4. Right-click **Messaging Providers**. Complete one of the following actions:
   - Right click the folder and select **New**.
   - On the menu bar, click **Edit > New**.
5. In the resulting property dialog, configure the JMS provider properties.
6. Click **OK** when finished.
7. Click **File -> Save** on the menu bar to save your changes.

JMS provider settings for application clients

Use this page to configure properties of the Java Message Service (JMS) provider, if you want to use a JMS provider other than the default messaging provider or the WebSphere MQ as a JMS provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right click **Messaging Providers**, and click **New**. The following fields appear on the **General** tab.

Name:

The name by which the JMS provider is known for administrative purposes.

Data type String

Description:

A description of the JMS provider, for administrative purposes.

Data type String

Class Path:

A list of paths or `.jar` file names which together form the location for the resource provider classes.

Context factory class:

The Java class name of the initial context factory for the JMS provider.

For example, for an LDAP service provider the value has the form: `com.sun.jndi.ldap.LdapCtxFactory`.

Data type String

Provider URL:

The JMS provider URL for external JNDI lookups.

For example, an LDAP URL for a JMS provider has the form: `ldap://hostname.company.com/contextName`.

Data type String

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Default Provider connection factory settings

Use this panel to view or change the configuration properties of the selected JMS connection factory for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server. These configuration properties control how connections are created between the JMS provider and the service integration bus that it uses

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Connection Factories** and click **New**. The following fields appear on the **General** tab.

Settings that have a default value display the appropriate value. Any settings that have fixed values have a drop down menu.

Name:

The name of the connection factory.

Data type String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String

JNDI Name:

The JNDI name that is used to match this Resource Adapter connection factory definition to the deployment descriptor. This entry is a `resource-ref` name.

Data type String

User Name:

The **User Name** used with the **Password** property for connecting to an application.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly. If a user name and password are specified, then an authentication alias is created for the factory where the password is encrypted.

Data type String

Password:

The password used to authenticate connection to an application.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

Data type String

Re-Enter Password:

Confirms the password.

Bus Name:

The name of the bus to which the connection factory connects.

Data type String

Client Identifier:

The name of the client. Required for durable topic subscriptions.

Data type String

Nonpersistent Messaging Reliability:

The reliability applied to nonpersistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus** destination. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Default
ReliablePersistent

Range

None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.

Best effort nonpersistent
Messages are never written to disk, and are thrown away if memory cache overruns.

Express nonpersistent
Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.

Reliable nonpersistent
Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.

Reliable persistent
Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.

Assured persistent
Highest degree of reliability where assured message delivery is supported.

As Bus destination
Use the delivery option configured for the bus destination.

Persistent Message Reliability:

The reliability applied to persistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus destination**. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Default
ReliablePersistent

Range

None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.

Best effort nonpersistent
Messages are never written to disk, and are thrown away if memory cache overruns.

Express nonpersistent
Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.

Reliable nonpersistent
Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.

Reliable persistent
Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.

Assured persistent
Highest degree of reliability where assured message delivery is supported.

As Bus destination
Use the delivery option configured for the bus destination.

Durable Subscription Home:

The name of the durable subscription home.

Data type String

Share durable subscriptions:

Controls whether or not durable subscriptions are shared across connections with members of a server cluster.

Normally, only one session at a time can have a TopicSubscriber for a particular durable subscription. This property enables you to override this behavior, to enable a durable subscription to have multiple simultaneous consumers.

Data type Selection list
Default In cluster
Range

In cluster
Allows sharing of durable subscriptions when connections are made from within a server cluster.

Always shared
Durable subscriptions can be shared across connections.

Never shared
Durable subscriptions are never shared across connections.

Read Ahead:

Controls the read-ahead optimization during message delivery.

Default	Default
Range	Default, AlwaysOn and AlwaysOff

Target:

The name of the Workload Manager target group containing the messaging engine.

Data type	String

Target Type:

The type of Workload Manager target group that contains the messaging engine.

Default	BusMember
Range	BusMember, Custom, ME

Target Significance:

The priority of significance for the target specified.

Default	Preferred
Range	Preferred, Required

Target Inbound Transport Chain:

The name of the protocol that resolves to a group of messaging engines.

Data type	String

Provider Endpoints:

The list of comma separated endpoints used to connect to a bootstrap server.

Type a comma-separated list of endpoint triplets with the syntax: host:port:protocol.

Example	`merlin:7276:BootstrapBasicMessaging,Gandalf:5557:BootstrapSecureMessaging` where `BootstrapBasicMessaging` corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).
Default	• If the host name is not specified, then the default `localhost` is used as a default value. • If the port number is not specified, then `7276` is used as a default value. • If the chain name is not specified, a predefined chain, such as `BootstrapBasicMessaging`, is used as a default value.

Connection Proximity:

The proximity that the messaging engine should have to the requester.

Default Bus
Range Bus, Host, Cluster, Server

Temporary Queue Name Prefix:

The prefix to apply to the names of temporary queues. This name is a maximum of 12 characters.

Data type String

Temporary Topic Name Prefix:

The prefix to apply to the names of temporary topics. This name is a maximum of 12 characters.

Data type String

Default Provider queue connection factory settings

Use this panel to view or change the configuration properties of the selected JMS queue connection factory for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server. These configuration properties control how connections are created between the JMS provider and the service integration bus that it uses

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Queue Connection Factories** and click **New**. The following fields appear on the **General** tab.

Settings that have a default value display the appropriate value. Any settings that have fixed values have a drop down menu.

Name:

The name of the queue connection factory.

Data type String

Description:

A description of this queue connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String

JNDI Name:

The JNDI name that is used to match this queue connection factory definition to the deployment descriptor. This entry is a `resource-ref` name.

Data type String

User Name:

The **User Name** used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly. If this field is used, then the Properties field `UserName` is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User Name** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly. If a user name and password are specified, then an authentication alias is created for the factory where the password is encrypted.

Data type String

Password:

The password used to create an encrypted. If you complete this field, then the Password field in the Properties box is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

Data type String

Re-Enter Password:

Confirms the password.

Bus Name:

The name of the bus to which the queue connection factory connects.

Data type String

Client Identifier:

The client identifier. Required for durable topic subscriptions.

Data type String

Nonpersistent Messaging Reliability:

The reliability applied to nonpersistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus** destination. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Default ReliablePersistent

Range

None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.

Best effort nonpersistent
Messages are never written to disk, and are thrown away if memory cache overruns.

Express nonpersistent
Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.

Reliable nonpersistent
Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.

Reliable persistent
Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.

Assured persistent
Highest degree of reliability where assured message delivery is supported.

As Bus destination
Use the delivery option configured for the bus destination.

Persistent Message Reliability:

The reliability applied to persistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus destination**. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Default ReliablePersistent

Range

None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.

Best effort nonpersistent
Messages are never written to disk, and are thrown away if memory cache overruns.

Express nonpersistent
Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.

Reliable nonpersistent
Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.

Reliable persistent
Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.

Assured persistent
Highest degree of reliability where assured message delivery is supported.

As Bus destination
Use the delivery option configured for the bus destination.

Read Ahead:

Controls the read-ahead optimization during message delivery.

Default	Default
Range	Default, AlwaysOn and AlwaysOff

Target:

The name of the Workload Manager target group containing the messaging engine.

Data type	String

Target Type:

The type of Workload Manager target group that contains the messaging engine.

Default	BusMember
Range	BusMember, Custom, Destination, ME

Target Significance:

The priority of significance for the target specified.

Default	Preferred
Range	Preferred, Required

Target Inbound Transport Chain:

The name of the protocol that resolves to a group of messaging engines.

Data type String

Provider Endpoints:

The list of comma separated endpoints used to connect to a bootstrap server.

Type a comma-separated list of endpoint triplets with the syntax: host:port:protocol.

Example `localhost:7777:BootstrapBasicMessaging`

where

`BootstrapBasicMessaging` corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).

Default

- If the host name is not specified, then the default `localhost` is used as a default value.
- If the port number is not specified, then `7276` is used as a default value.
- If the chain name is not specified, a predefined chain, such as `BootstrapBasicMessaging`, is used as a default value.

Connection Proximity:

The proximity that the messaging engine should have to the requester.

Default Bus, Cluster, Server
Range Bus, Host

Temporary Queue Name Prefix:

The prefix to apply to the names of temporary queues. This name is a maximum of 12 characters.

Data type String

Default Provider topic connection factory settings

Use this panel to view or change the configuration properties of the selected JMS topic connection factory for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server. These configuration properties control how connections are created between the JMS provider and the service integration bus that it uses.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Topic Connection Factories** and click **New**. The following fields appear on the **General** tab.

Settings that have a default value display that appropriate value. Any settings that have fixed values have a drop down menu.

Name:

The name of the topic connection factory.

Data type String

Description:

A description of this topic connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String

JNDI Name:

The JNDI name that is used to match this topic connection factory definition to the deployment descriptor. This entry is a `resource-ref` name.

Data type String

User Name:

The **User Name** used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly. If this field is used, then the Properties field `UserName` is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User Name** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly. If a user name and password are specified, then an authentication alias is created for the factory where the password is encrypted.

Data type String

Password:

The password used to create an encrypted. If you complete this field, then the Password field in the Properties box is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

Data type String

Re-Enter Password:

Confirms the password.

Bus Name:

The name of the bus to which the topic connection factory connects.

Data type String

Client Identifier:

The name of the client. This field is required for durable topic subscriptions.

Data type String

Nonpersistent Messaging Reliability:

The reliability applied to nonpersistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus** destination. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Default ReliablePersistent

Range

None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.

Best effort nonpersistent
Messages are never written to disk, and are thrown away if memory cache overruns.

Express nonpersistent
Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.

Reliable nonpersistent
Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.

Reliable persistent
Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.

Assured persistent
Highest degree of reliability where assured message delivery is supported.

As Bus destination
Use the delivery option configured for the bus destination.

Persistent Message Reliability:

The reliability applied to persistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus destination**. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Default ReliablePersistent

Range

None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.

Best effort nonpersistent
Messages are never written to disk, and are thrown away if memory cache overruns.

Express nonpersistent
Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.

Reliable nonpersistent
Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.

Reliable persistent
Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.

Assured persistent
Highest degree of reliability where assured message delivery is supported.

As Bus destination
Use the delivery option configured for the bus destination.

Durable Subscription Home:

The name of the durable subscription home.

Data type String

Share durable subscriptions:

Controls whether or not durable subscriptions are shared across connections with members of a server cluster.

Normally, only one session at a time can have a TopicSubscriber for a particular durable subscription. This property enables you to override this behavior, to enable a durable subscription to have multiple simultaneous consumers.

Data type Selection list
Default In cluster
Range

In cluster
Allows sharing of durable subscriptions when connections are made from within a server cluster.

Always shared
Durable subscriptions can be shared across connections.

Never shared
Durable subscriptions are never shared across connections.

Read Ahead:

Controls the read-ahead optimization during message delivery.

Default	Default
Range	Default, AlwaysOn and AlwaysOff

Target:

The name of the Workload Manager target group containing the messaging engine.

Data type	String

Target Type:

The type of Workload Manager target group that contains the messaging engine.

Default	BusMember
Range	BusMember, Custom, ME

Target Significance:

The priority of significance for the target specified.

Default	Preferred
Range	Preferred, Required

Target Inbound Transport Chain:

The name of the protocol that resolves to a group of messaging engines.

Data type	String

Provider Endpoints:

The list of comma separated endpoints used to connect to a bootstrap server.

Type a comma-separated list of endpoint triplets with the syntax: host:port:protocol.

Example	`localhost:7777:BootstrapBasicMessaging` where `BootstrapBasicMessaging` corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).
Default	• If the host name is not specified, then the default `localhost` is used as a default value. • If the port number is not specified, then `7276` is used as a default value. • If the chain name is not specified, a predefined chain, such as `BootstrapBasicMessaging`, is used as a default value.

Connection Proximity:

The proximity that the messaging engine should have to the requester.

Default	Bus
Range	Bus, Host, Cluster, Server

Temporary Topic Name Prefix:

The prefix to apply to the names of temporary topics. This name is a maximum of 12 characters.

Data type	String

Default Provider queue destination settings

Use this panel to view or change the configuration properties of the selected JMS queue destination for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Queue Destinations**. Click **New**. The following fields appear on the **General** tab.

Name:

The name of the queue destination factory. You must complete this field.

Data type	String

Description:

A description of this queue destination for administrative purposes within WebSphere Application Server.

Data type	String

JNDI Name:

The JNDI name used to match this definition to a deployment descriptor `resource-env-ref` name.

Data type	String

Queue Name:

The name of the queue.

Data type	String

Delivery Mode:

The delivery mode for messages sent to this destination.

Data type	String
Range	Application, Persistent or NonPersistent
Default	Application

Time to Live:

The default length of time from its dispatch time that a message sent to this destination should be retained by the system, where **0** indicates that time to live value does not expire. Value from the producer is used if the Time to Live field is not completed.

Data type	Integer
Units	Milliseconds

Priority:

The priority for messages sent to this destination. The value from the producer is used if not completed.

Data type	Integer
Range	0 to 9 with **0** as the lowest priority and **9** as the highest priority

Read Ahead:

Used to control read-ahead optimization during message delivery.

Data type	String
Range	AsConnection, AlwaysOn and AlwaysOff
Default	AsConnection

Default Provider topic destination settings

Use this panel to view or change the configuration properties of the selected JMS topic destination for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Topic Destinations**, and click **New**. The following fields appear on the **General** tab.

Name:

The name of the topic destination entry.

Data type	String

Description:

A description of the entry.

Data type	String

JNDI Name:

The JNDI name used to match this definition to a deployment descriptor `resource-env-ref` name.

Data type	String

Topic Space:

The name of the topic space. This field is required.

Data type	String
Default	DEFAULT_TOPIC_SPACE

Topic Name:

The name of the topic. This field is required.

Data type	String

Delivery Mode:

The default mode for messages sent to this destination.

Data type	String
Range	Application, Persistent or NonPersistent
Default	Application

Time to Live:

The default length of time from its dispatch time that a message sent to this destination should be retained by the system, where **0** indicates that time to live value does not expire. Value from the producer is used if not completed.

Data type	Long
Units	Milliseconds

Priority:

The priority for messages sent to this destination. Value from producer is used if not completed.

Data type	Integer
Range	0 to 9 with **0** as the lowest priority and **9** as the highest priority

Read Ahead:

Used to control read-ahead optimization during message delivery.

Data type	String
Range	AsConnection, AlwaysOn and AlwaysOff
Default	AsConnection

Version 5 Default Provider queue connection factory settings for application clients

Use this panel to browse or change the configuration properties of the selected JMS queue connection factory for point-to-point messaging for use by WebSphere Application Server version 5 applications. These configuration properties control how connections are created between the JMS provider and the default messaging system that it uses.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Provider** > **Version 5 Default Provider**. Right-click **Queue Connection Factories** and click **New**. The following fields appear on the **General** tab.

A queue connection factory is used to create JMS connections to queue destinations. The queue connection factory is created by the internal WebSphere Application Server product JMS provider. A Version 5 Default Provider queue connection factory has the following properties:

Name:

The name by which this queue connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

JNDI Name:

The application client run time uses this field to retrieve configuration information.

User ID:

The User ID used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a User ID and password explicitly, for example, if the calling application uses the method createQueueConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Data type String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Re-Enter Password:

Confirms the password.

Node:

The WebSphere node name of the administrative node where the JMS server runs for this connection factory. Connections created by this factory connect to that JMS server.

Data type String

Application Server:

Enter the name of the application server. This name is not the host name of the machine, but the name of the configured application server.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Version 5 Default Provider topic connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected topic connection factory for use with the internal product Java Message Service (JMS) provider. These configuration properties control how connections are created between the JMS provider and the messaging system that it uses.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Version 5 Default Provider**. Right click **Topic Connection Factories** and click **New**. The following fields appear on the **General** tab.

A Version 5 Default Provider topic connection factory has the following properties.

Name:

The name by which this queue connection factory is known for administrative purposes within WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere Application Server administrative domain.

Data type String

Description:

A description of this topic connection factory for administrative purposes within WebSphere Application Server.

Data type String

JNDI Name:

The application client run time uses this field to retrieve configuration information.

User ID:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly, for example, if the calling application uses the method createTopicConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Data type String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type String

Re-Enter Password:

Confirms the password.

Node:

The WebSphere Application Server node name of the administrative node where the JMS server runs for this connection factory. Connections created by this factory connect to that JMS server.

Data type Enum
Range Pull-down list of nodes in the WebSphere Application Server administrative domain.

Application Server:

Enter the name of the application server. This name is not the host name of the machine, but the name of the configured application server.

Port:

Which of the two ports that connections use to connect to the JMS Server. The QUEUED port is for full-function JMS publish/subscribe support, the DIRECT port is for nonpersistent, nontransactional, nondurable subscriptions only.

Note: Message-driven beans cannot use the direct listener port for publish or subscribe support. Therefore, any topic connection factory configured with the Port set to `Direct` cannot be used with message-driven beans.

Data type Enum
Default QUEUED

Range

QUEUED
The listener port used for full-function JMS compliant, publish or subscribe support.

DIRECT
The listener port used for direct TCP/IP connection (nontransactional, nonpersistent, and nondurable subscriptions only) for publish or subscribe support.

The TCP/IP port numbers for these ports are defined on the product internal JMS server.

Client ID:

The JMS client identifier used for connections to the MQSeries® queue manager.

Data type String

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Version 5 Default Provider queue destination settings for application clients

Use this panel to view or change the configuration properties of the selected queue destination for use with product Java Message Service (JMS) provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Version 5 Default Provider**. Right click **Queue Destinations** and click **New**. The following fields are displayed on the **General** tab.

A queue destination is used to configure the properties of a JMS queue. A Version 5 Default Provider queue destination has the following properties.

Name:

The name by which the queue is known for administrative purposes within WebSphere Application Server.

Data type String

Description:

A description of the queue, for administrative purposes.

Data type String

JNDI Name:

The application client run time uses this field to retrieve configuration information.

Persistence:

Whether all messages sent to the destination are persistent, nonpersistent, or have their persistence defined by the application.

Data type Enum

Default APPLICATION_DEFINED

Range

Application defined
Messages on the destination have their persistence defined by the application that put them onto the queue.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Persistent
Messages on the destination are persistent.

Nonpersistent
Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property.

Data type Enum

Default APPLICATION_DEFINED

Range

Application defined
The priority of messages on this destination is defined by the application that put them onto the destination.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Specified
The priority of messages on this destination is defined by the **Specified priority** property. *If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest).

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type Integer

Units Message priority level

Range 0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or whether messages on the queue expire (have an unlimited expiry timeout).

Data type Enum

Default APPLICATION_DEFINED

Range

Application defined
The expiry timeout for messages in this queue is defined by the application that put them onto the queue.

Specified
The expiry timeout for messages in this queue is defined by the **Specified expiry** property.If you select this option, you must define a time out on the **Specified expiry** property.

Unlimited
Messages in this queue have no expiry timeout, and those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, specify the number of milliseconds (greater than 0) after which messages on this queue expire.

Data type Integer

Units Milliseconds

Range Greater than or equal to 0

- 0 indicates that messages never timeout.
- Other values are an integer number of milliseconds.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Version 5 Default Provider topic destination settings for application clients

Use this panel to view or change the configuration properties of the selected topic destination for use with the internal product Java Message Service (JMS) provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Version 5 Default Provider**. Right click **Topic Destinations** and click **New**. The following fields appear on the **General** tab.

A topic destination is used to configure the properties of a JMS topic for the associated JMS provider. A Version 5 Default Provider topic has the following properties.

Name:

The name by which the topic is known for administrative purposes.

Data type String

Description:

A description of the topic, for administrative purposes within WebSphere Application Server.

Data type String

JNDI Name:

The application client run-time environment uses this field to retrieve configuration information.

Topic Name: The name of the topic as defined to the JMS provider.

Data type String

Persistence:

Whether all messages sent to the destination are persistent, nonpersistent, or have their persistence defined by the application.

Data type Enum
Default APPLICATION_DEFINED
Range

Application defined
Messages on the destination have their persistence defined by the application that put them onto the queue.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Persistent
Messages on the destination are persistent.

Nonpersistent
Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property.

Data type Enum
Default APPLICATION_DEFINED
Range

Application defined
The priority of messages on this destination is defined by the application that put them onto the destination.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Specified
The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, specify the message priority for this queue, in the range 0 (lowest) through 9 (highest).

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout).

Data type	Enum
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property. *If you select this option, you must define a timeout on the **Specified expiry** property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire.

Data type	Integer
Units	Milliseconds
Range	Greater than or equal to 0 • 0 indicates that messages never time out. • Other values are an integer number of milliseconds.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere MQ Provider queue connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected queue connection factory for use with the MQSeries product Java Message Service (JMS) provider. These configuration properties control how connections are created between the JMS provider and WebSphere MQ.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **WebSphere MQ Provider**. Right click **Queue Connection Factories**, and click **New**. The following fields are displayed on the **General** tab.

Note:

- The property values that you specify must match the values that you specified when configuring WebSphere MQ for JMS resources. For more information about configuring WebSphere MQ for JMS resources, see WebSphere MQ *Using Java*, available from the WebSphere MQ library.
- In WebSphere MQ, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

A queue connection factory for the JMS provider has the following properties.

Name:

The name by which this queue connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

JNDI Name:

The application client run time uses this field to retrieve configuration information.

User ID:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Data type String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type String

Default Null

Re-Enter Password:

Confirms the password.

Queue Manager:

The name of the MQSeries queue manager for this connection factory.

Connections created by this factory connect to that queue manager.

Data type String

Host:

The name of the host on which the WebSphere MQ queue manager runs for client connection only.

Data type String
Default Null
Range A valid TCP/IP host name

Port:

The TCP/IP port number used for connection to the WebSphere MQ queue manager, for client connection only.

This port must be configured on the WebSphere MQ queue manager.

Data type Integer
Default Null
Range A valid TCP/IP port number, configured on the WebSphere MQ queue manager.

Channel:

The name of the channel used for connection to the WebSphere MQ queue manager, for client connection only.

Data type String
Default Null
Range 1 through 20 ASCII characters

Transport type:

Specifies whether the WebSphere MQ client connection or JNDI bindings are used for connection to the WebSphere MQ queue manager. The external JMS provider controls the communication protocols between JMS clients and JMS servers. Tune the transport type when you are using non-ASF nonpersistent, nondurable, nontransactional messaging or when you want to satisfy security issues and the client is local to the queue manager node.

Data type Enum
Units Not applicable

Default BINDINGS

Range

BINDINGS
JNDI bindings are used to connect to the queue manager. BINDINGS is a shared memory protocol and can only be used when the queue manager is on the same node as the JMS client and poses security risks that should be addressed through the use of EJB roles.

CLIENT
WebSphere MQ client connection is used to connect to the queue manager. CLIENT is a typical TCP-based protocol.

DIRECT
For WebSphere MQ Event Broker using DIRECT mode. DIRECT is a lightweight sockets protocol used in nontransactional, nondurable and nonpersistent Publish/Subscribe messaging. DIRECT only works for clients and message-driven beans using the non-ASF protocol.

QUEUED
QUEUED is a standard TCP protocol.

Recommended

Queue connection factory transport type
BINDINGS is faster by 30% or more, but it lacks security. When you have security concerns, BINDINGS is more desirable than CLIENT.

Topic connection factory transport type
DIRECT is the fastest type and should be used where possible. Use BINDINGS when you want to satisfy additional security tasks and the queue manager is local to the JMS client. QUEUED is the fallback for all other cases. WebSphere MQ 5.3 before CSD2 with the DIRECT setting can lose messages when used with message-driven beans and under load. This loss also happens with client-side applications unless the broker maxClientQueueSize is set to 0. You can set this to 0 with the command:

```
#wempschangeproperties WAS_nodeName_server1
 -e default -o DynamicSubscriptionEngine -n
 maxClientQueueSize -v 0 -x executionGroupUUID
```

where executionGroupUUID can be found by starting the broker and looking in the Event Log/Applications for event 2201. This value is usually ffffffff-0000-0000-000000000000.

Note: The WebSphere MQ 5.3 JMS cannot be used within WAS 6.1 because WAS 6.1 has a Java 5 runtime. Therefore, cross-memory connections cannot be established with WebSphere MQ 5.3 queue managers. This can result in a performance degradation if you were previously using WebSphere MQ 5.3 and BINDINGS for your connections and move to CLIENT network connections in migrating to WAS 6.1. If you are using WebSphere MQ 5.3 for z/OS, you might also have to purchase an additional feature pack.

When running on 64-bit z/OS, the Transport type must be set to CLIENT because the 64-bit WebSphere MQ z/OS is not currently available, and BINDINGS mode cannot be used to connect to 31-bit WebSphere MQ z/OS. You might also need to purchase an additional WebSphere MQ feature pack for this support.

Client ID:

The JMS client identifier used for connections to the MQSeries queue manager.

Data type String

CCSID:

The coded character set identifier for use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Data type String

For more information about supported CCSIDs, and about converting between message data from one coded character set to another, see the *WebSphere MQ System Administration* and the *WebSphere MQ Application Programming Reference* books. These references are available from the WebSphere MQ messaging multiplatform and platform-specific books Web pages; for example, at http://www.ibm.com/software/integration/wmq/library/, the IBM Publications Center, or from the WebSphere MQ collection kit, SK2T-0730.

Message Retention:

Select this check box to specify that unwanted messages are to be left on the queue. Otherwise, unwanted messages are handled according to their disposition options.

Data type Enum
Units Not applicable
Default Cleared
Range
Selected
Unwanted messages are left on the queue.
Cleared
Unwanted messages are handled according to their disposition options.

Temporary model:

The name of the model definition used to create temporary connection factories if a connection factory does not already exist.

Data type String
Range 1 through 48 ASCII characters

Temporary queue prefix:

The prefix used for dynamic queue naming.

Data type String

Fail if quiesce:

Specifies whether applications return from a method call if the queue manager has entered a controlled failure.

Data type Check box
Default Selected

Local Server Address:

Specifies the local server address.

Data type String

Polling Interval:

Specifies the interval, in milliseconds, between scans of all receivers during asynchronous message delivery

Data type	Integer
Units	Milliseconds
Default	5000

Rescan interval:

Specifies the interval in milliseconds between which a topic is scanned to look for messages that have been added to a topic out of order.

This interval controls the scanning for messages that have been added to a topic out of order with respect to a WebSphere MQ browse cursor.

Data type	Integer
Units	Milliseconds
Default	5000

SSL cipher suite:

Specifies the cipher suite to use for SSL connection to WebSphere MQ.

Set this property to a valid cipher suite provided by your JSSE provider. The value must match the CipherSpec specified on the SVRCONN channel as the **Channel** property.

You must set this property, if you set the **SSL Peer Name** property.

SSL certificate store:

Specifies a list of zero or more Certificate Revocation List (CRL) servers used to check for SSL certificate revocation. If you specify a value for this property, you must use WebSphere MQ JVM at Java 2 version 1.4.

The value is a space-delimited list of entries of the form:

`ldap://`*`hostname`*`:[`*`port`*`]`

A single slash (/) follows this value. If *port* is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers. For more information about CRL security, see the section “Working with Certificate Revocation Lists” in the *WebSphere MQ Security book*; for example at: http://publibfp.boulder.ibm.com/epubs/html/csqzas01/csqzas012w.htm#IDX2254.

SSL peer name:

For SSL, a *distinguished name* skeleton that must match the name provided by the WebSphere MQ queue manager. The distinguished name is used to check the identifying certificate presented by the server at connection time.

If this property is not set, such certificate checking is performed.

The SSL peer name property is ignored if **SSL Cipher Suite** property is not specified.

This property is a list of attribute name and value pairs separated by commas or semicolons. For example:
`CN=QMGR.*, OU=IBM, OU=WEBSPHERE`

The example given checks the identifying certificate presented by the server at connect-time. For the connection to succeed, the certificate must have a Common Name beginning QMGR., and must have at least two Organizational Unit names, the first of which is IBM and the second WEBSPHERE. Checking is not case-sensitive.

For more details about distinguished names and their use with WebSphere MQ, see the section "Distinguished Names" in the WebSphere MQ Security book.

Connection pool:

Specifies an optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the JMS provider to improve performance. This is independent from any WebSphere MQ connection pooling. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Data type	Check box
Default	Selected

WebSphere MQ Provider topic connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected topic connection factory for use with the WebSphere MQ product Java Message Service (JMS) provider. These configuration properties control how connections are created between the JMS provider and WebSphere MQ.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **WebSphere MQ Provider**. Right-click **Topic Connection Factories** and click **New**.

Note:

- The property values that you specify must match the values that you specified when configuring WebSphere MQ product JMS resources. For more information about configuring WebSphere MQ product JMS resources, see the WebSphere MQ *Using Java* book.
- In WebSphere MQ, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

MA0C broker: When creating a WebSphere Application Server v6 topic connection factory for the MA0C broker, you should consider the following attribute values:

BrokerControlQueue
: This value is fixed at SYSTEM.BROKER.CONTROL.QUEUE for the MA0C broker and is the queue the broker reads from.

BrokerVersion
: Set this value to BASIC for the MA0C broker.

ClientID
: Set this value to whatever you like for the MA0C broker (the value is string and is merely an identifier for your client application).

XA Enabled
Set this value to TRUE or FALSE for the MA0C broker (the setting you use is a performance enhancement flag - you will probably want to set this to 'true' most of the time).

BrokerMessage Selection
This value is fixed at CLIENT for the MA0C broker because the broker relies on client side message selection.

Direct Broker Authorization Type
This value is not required by the MA0C broker.

A topic connection factory for the WebSphere MQ product JMS provider has the following properties.

Name:

The name by which this topic connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS provider.

Data type	String

Description:

A description of this topic connection factory for administrative purposes within IBM WebSphere Application Server.

Data type	String

JNDI Name:

The Java Naming and Directory Interface (JNDI) name that is used to bind the topic connection factory into the application server name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String
Units	En_US ASCII characters
Range	1 through 45 ASCII characters

User ID:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User** property, you must also specify a value for the **Password** property.

The connection factory **User** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly, for example, if the calling application uses the method createTopicConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Data type String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type String

Re-Enter Password:

Confirms the password.

Queue Manager:

The name of the WebSphere MQ queue manager for this connection factory. Connections created by this factory connect to that queue manager.

Data type String

Host:

The name of the host on which the WebSphere MQ queue manager runs for client connections only.

Data type String
Range A valid TCP/IP host name

Port:

The TCP/IP port number used for connection to the WebSphere MQ queue manager, for client connection only.

This port must be configured on the WebSphere MQ queue manager.

Data type Integer
Range A valid TCP/IP port number, configured on the WebSphere MQ queue manager.

Channel:

The name of the channel used for client connections to the WebSphere MQ queue manager for client connection only.

Data type String
Range 1 through 20 ASCII characters

Transport Type:

Whether WebSphere MQ client connection or JNDI bindings are used for connection to the WebSphere MQ queue manager.

Data type Enum
Default BINDINGS
Range
CLIENT
WebSphere MQ client connection is used to connect to the WebSphere MQ queue manager.
BINDINGS
JNDI bindings are used to connect to the WebSphere MQ queue manager.

Client ID:

The JMS client identifier used for connections to the WebSphere MQ queue manager.

Data type String

CCSID:

The coded character set identifier to be used with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Data type String

Broker Control Queue:

The name of the broker control queue to which all command messages (except publications and requests to delete publications) are sent.

Data type String
Units En_US ASCII characters
Range 1 through 48 ASCII characters

Broker Queue Manager:

The name of the WebSphere MQ queue manager that provides the Publisher and Subscriber message broker.

Data type String
Units En_US ASCII characters
Range 1 through 48 ASCII characters

Broker Publish Queue:

The name of the broker input queue that receives all publication messages for the default stream.

The name of the broker's input queue (stream queue) that receives all publication messages for the default stream. Applications can also send requests to delete publications on the default stream to this queue.

Data type String
Units En_US ASCII characters
Range 1 through 48 ASCII characters

Broker Subscribe Queue:

The name of the broker queue from which nondurable subscription messages are retrieved.

The name of the broker queue from which nondurable subscription messages are retrieved. The subscriber specifies the name of the queue when it registers a subscription.

Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker CCSubQ:

The name of the broker queue from which nondurable subscription messages are retrieved for a ConnectionConsumer request. This property applies only for use of the Web container.

Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker Version:

Specifies whether the message broker is provided by the WebSphere MQ MA0C SupportPac™ or newer versions of WebSphere family message broker products.

Data type	Enum
Default	Advanced
Range	**Advanced** The message broker is provided by newer versions of WebSphere family message broker products (MQ Integrator and MQ Publish and Subscribe). **Basic** The message broker is provided by the WebSphere MQ MA0C SupportPac (WebSphere MQ - Publish and Subscribe).

Cleanup level:

Specifies the level of clean up provided by the publish or subscribe cleanup utility.

Data type	Enum
Default	SAFE
Range	**ASPROP** **NONE** **STRONG**

Cleanup interval:

Specifies the interval, in milliseconds, between background executions of the publish/subscribe cleanup utility.

Data type	Integer

Units Milliseconds
Default 6000

Message selection:

Specifies where broker message selection is performed.

Data type Enum
Default BROKER
Range

BROKER
Message selection is done at the broker location.

Message CLIENT
Message selection is done at the client location.

Publish acknowledge interval:

The interval, in number of messages, between publish requests that require acknowledgement from the broker.

Data type Integer
Default 25

Sparse subscriptions:

Enables sparse subscriptions.

Data type Check box
Default Cleared

Status refresh interval:

The interval, in milliseconds, between transactions to refresh publish or subscribe status.

Data type Integer
Default 6000

Subscription store:

Specifies where WebSphere MQ stores data relating to active JMS subscriptions.

Data type Enum
Default MIGRATE
Range

MIGRATE

QUEUE

BROKER

Multicast:

Specifies whether this connection factory uses multicast transport.

Data type Enum
Default NOT USED
Range

NOT USED
This connection factory does not use multicast transport.

ENABLED
This connection factory always uses multicast transport.

ENABLED_IF_AVAILABLE
This connection factory uses multicast transport.

ENABLED_RELIABLE
This connection factory uses reliable multicast transport.

ENABLED_RELIABLE_IF_AVAILABLE
This connection factory uses reliable multicast transport if available.

Direct authentication:

Specifies whether to use direct broker authorization.

Data type Enum
Default NONE
Range

NONE Direct broker authorization is not used.

PASSWORD
Direct broker authorization is authenticated with a password.

CERTIFICATE
Direct broker authorization is authenticated with a certificates.

Proxy Host Name:

Specifies the host name of a proxy to be used for communication with WebSphere MQ.

Data type String

Proxy Port:

Specifies the port number of a proxy to be used for communication with WebSphere MQ.

Data type Integer
Default 0

Fail if quiesce:

Specifies whether applications return from a method call if the queue manager has entered a controlled failure.

Data type Check box
Default Selected

Local Server Address:

Specifies the local server address.

Data type	String

Polling Interval:

Specifies the interval, in milliseconds, between scans of all receivers during asynchronous message delivery.

Data type	Integer
Units	Milliseconds
Default	5000

Rescan interval:

Specifies the interval in milliseconds between which a topic is scanned to look for messages that have been added to a topic out of order.

This interval controls the scanning for messages that have been added to a topic out of order with respect to a WebSphere MQ browse cursor.

Data type	Integer
Units	Milliseconds
Default	5000

SSL cipher suite:

Specifies the cipher suite to use for SSL connection to WebSphere MQ.

Set this property to a valid cipher suite provided by your JSSE provider. The value must match the CipherSpec specified on the SVRCONN channel as the **Channel** property.

You must set this property, if you set the **SSL Peer Name** property.

SSL certificate store:

Specifies a list of zero or more Certificate Revocation List (CRL) servers used to check for SSL certificate revocation. If you specify a value for this property, you must use WebSphere MQ JVM at Java 2 version 1.4.

The value is a space-delimited list of entries of the form:

`ldap://hostname:[port]`

A single slash (/) follows this value. If *port* is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers. For more information about CRL security, see the section “Working with Certificate Revocation Lists” in the *WebSphere MQ Security book*; for example at: http://publibfp.boulder.ibm.com/epubs/html/csqzas01/csqzas012w.htm#IDX2254.

SSL peer name:

For SSL, a *distinguished name* skeleton that must match the name provided by the WebSphere MQ queue manager. The distinguished name is used to check the identifying certificate presented by the server at connection time.

If this property is not set, such certificate checking is performed.

The SSL peer name property is ignored if **SSL Cipher Suite** property is not specified.

This property is a list of attribute name and value pairs separated by commas or semicolons. For example:

```
CN=QMGR.*, OU=IBM, OU=WEBSPHERE
```

The example given checks the identifying certificate presented by the server at connect-time. For the connection to succeed, the certificate must have a Common Name beginning QMGR., and must have at least two Organizational Unit names, the first of which is IBM and the second WEBSPHERE. Checking is not case-sensitive.

For more details about distinguished names and their use with WebSphere MQ, see the section “Distinguished Names” in the WebSphere MQ Security book.

Connection pool:

Specifies an optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the JMS provider to improve performance. This is independent from any WebSphere MQ connection pooling. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Data type	Check box
Default	Selected

WebSphere MQ Provider queue destination settings for application clients

Use this panel to view or change the configuration properties of the selected queue destination for use with the WebSphere MQ product Java Message Service (JMS) provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **WebSphere MQ Provider**. Right-click **Queue Destinations** and click **New**. The following fields are displayed on the **General** tab.

Note:

- The property values that you specify must match the values that you specified when configuring WebSphere MQ product for JMS resources. For more information about configuring WebSphere MQ product for JMS resources, see the WebSphere MQ *Using Java* book.
- In WebSphere MQ, names can have a maximum of 48 characters.

A queue for use with the WebSphere MQ product JMS provider has the following properties.

Name:

The name by which the queue is known for administrative purposes within IBM WebSphere Application Server.

Data type String

Description:

A description of the queue, for administrative purposes.

Data type String

JNDI Name:

The application client run-time environment uses this field to retrieve configuration information.

Persistence:

Whether all messages sent to the destination are persistent, nonpersistent or have their persistence defined by the application.

Data type Enum
Default APPLICATION_DEFINED
Range

Application defined
Messages on the destination have their persistence defined by the application that put them onto the queue.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Persistent
Messages on the destination are persistent.

Nonpersistent
Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property.

Data type Enum
Units Not applicable
Default APPLICATION_DEFINED
Range

Application defined
The priority of messages on this destination is defined by the application that put them onto the destination.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Specified
The priority of messages on this destination is defined by the **Specified priority** property. *If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, specify the message priority for this queue, in the range 0 (lowest) through 9 (highest).

Data type Integer
Units Message priority level
Range 0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout value for this queue is defined by the application or the by **Specified expiry** property or whether messages on the queue never expire (have an unlimited expiry time out).

Data type Enum
Units Not applicable
Default APPLICATION_DEFINED
Range

Application defined
The expiry timeout for messages on this queue is defined by the application that put them onto the queue.

Specified
The expiry timeout for messages on this queue is defined by the **Specified expiry** property. If you select this option, you must define a timeout on the **Specified expiry** property.

Unlimited
Messages on this queue have no expiry timeout and those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire.

Data type Integer
Units Milliseconds
Range Greater than or equal to 0
- 0 indicates that messages never time out
- Other values are an integer number of milliseconds

Base Queue Name:

The name of the queue to which messages are sent, on the queue manager specified by the **Base queue manager name** property.

Data type String

Base Queue Manager Name:

The name of the WebSphere MQ queue manager to which messages are sent.

This queue manager provides the queue specified by the **Base queue name** property.

Data type String

Units En_US ASCII characters
Range A valid WebSphere MQ Queue Manager name, as 1 through 48 ASCII characters

CCSID:

The coded character set identifier to use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSID identifier supported by WebSphere MQ queue manager.

Data type String

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Data type Enum
Default NORMAL
Range
NORMAL
Normal integer encoding is used.
REVERSED
Reversed integer encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Decimal encoding:

Indicates that if native encoding is not enabled to select whether decimal encoding is normal or reversed.

Data type Enum
Default NORMAL
Range
NORMAL
Normal decimal encoding is used.
REVERSED
Reversed decimal encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Floating point encoding:

Indicates that if native encoding is not enabled to select the type of floating point encoding.

Data type Enum
Default IEEENORMAL
Range
IEEENORMAL
IEEE normal floating point encoding is used.
IEEEREVERSED
IEEE reversed floating point encoding is used.
S390 S390 floating point encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Native encoding:

Indicates that the queue destination use native encoding (appropriate encoding values for the Java platform) when you select this check box.

Data type Enum
Default Cleared
Range
Cleared
Native encoding is not used, so specify the following properties for integer, decimal and floating point encoding.
Selected
Native encoding is used (to provide appropriate encoding values for the Java platform).

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Target client:

Whether the receiving application is JMS-compliant or is a traditional WebSphere MQ application.

Data type Enum
Default WebSphere MQ
Range
WebSphere MQ
The target is a traditional WebSphere MQ application that does not support JMS.
JMS The target application supports JMS.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere MQ Provider topic destination settings for application clients

Use this panel to view or change the configuration properties of the selected topic destination for use with the WebSphere MQ product Java Message Service (JMS) provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **WebSphere MQ Provider**. Right click **Topic Destinations**, and click **New**. The following fields are displayed on the **General** tab.

Note:

- The property values that you specify must match the values that you specified when configuring WebSphere MQ product JMS resources. For more information about configuring WebSphere MQ product JMS resources, see the WebSphere MQ *Using Java* book.
- In WebSphere MQ, names can have a maximum of 48 characters.

A topic destination is used to configure the properties of a JMS topic for the associated JMS provider. A topic for use with the WebSphere MQ product JMS provider has the following properties.

Name:

The name by which the topic is known for administrative purposes.

Data type String

Description:

A description of the topic for administrative purposes within IBM WebSphere Application Server.

JNDI Name:

The application client run time uses this field to retrieve configuration information.

Persistence:

Specifies whether all messages sent to the destination are persistent, nonpersistent, or have their persistence defined by the application.

Data type Enum
Default APPLICATION_DEFINED
Range

Application defined
Messages on the destination have their persistence defined by the application that put them in the queue.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Persistent
Messages on the destination are persistent.

Nonpersistent
Messages on the destination are not persistent.

Priority:

Specifies whether the message priority for this destination is defined by the application or the **Specified priority** property.

Data type Enum
Default APPLICATION_DEFINED
Range

Application defined
The priority of messages on this destination is defined by the application that put them in the destination.

Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

Specified
The priority of messages on this destination is defined by the **Specified priority** property. If you select this option, you must define a priority for the **Specified priority** property.

Specified Priority:

If the **Priority** property is set to `Specified`, type the message priority for this queue, in the range 0 (lowest) through 9 (highest).

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type Integer
Units Message priority level
Range 0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or by the **Specified expiry** property or by messages on the queue never expire (have an unlimited expiry timeout).

Data type Enum
Default APPLICATION_DEFINED
Range

Application defined
The expiry timeout for messages on this queue is defined by the application that put them in the queue.

Specified
The expiry timeout for messages in this queue is defined by the **Specified expiry** property. If you select this option, you must define a timeout value for the **Specified expiry** property.

Unlimited
Messages on this queue have no expiry timeout, and these messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type the number of milliseconds (greater than 0) after which messages on this queue expire.

Data type Integer
Units Milliseconds
Range Greater than or equal to 0
- 0 indicates that messages never time out.
- Other values are an integer number of milliseconds.

Base Topic Name:

The name of the topic to which messages are sent.

Data type String

CCSID:

The coded character set identifier to use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSID identifiers that WebSphere MQ supports.

Data type String

Units Integer
Range 1 through 65535

Integer encoding:

Indicates whether integer encoding is normal or reversed when native encoding is not enabled.

Data type Enum
Default NORMAL
Range
NORMAL
Normal integer encoding is used.
REVERSED
Reversed integer encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Data type Enum
Default NORMAL
Range
NORMAL
Normal decimal encoding is used.
REVERSED
Reversed decimal encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Floating point encoding:

Indicates the type of floating point encoding when native encoding is not enabled.

Data type Enum
Default IEEENORMAL
Range
IEEENORMAL
IEEE normal floating point encoding is used.
IEEEREVERSED
IEEE reversed floating point encoding is used.
S390 S/390® floating point encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Native encoding:

Indicates that the queue destination uses native encoding (appropriate encoding values for the Java platform) when you select this check box.

Data type Enum
Default Cleared

Range

Cleared
Native encoding is not used, so specify the previous properties for integer, decimal and floating point encoding.

Selected
Native encoding is used (to provide appropriate encoding values for the Java platform).

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

BrokerDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved.

The subscriber specifies the name of the queue when it registers a subscription.

Data type String
Units En_US ASCII characters
Range 1 through 48 ASCII characters

BrokerCCDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the Web container.

Data type String
Units En_US ASCII characters
Range 1 through 48 ASCII characters

Target Client:

Specifies whether the receiving application is JMS compliant or is a traditional WebSphere MQ application.

Data type Enum
Default WebSphere MQ
Range

WebSphere MQ
The target is a traditional WebSphere MQ application that does not support JMS.

JMS The target is a JMS compliant application.

Multicast:

Specifies whether this connection factory uses multicast transport.

Data type Enum
Default AS_CF

Range

AS_CF This connection factory uses multicast transport.

DISABLED
This connection factory does not use multicast transport.

NOT_RELIABLE
This connection factory always uses multicast transport.

RELIABLE
This connection factory uses multicast transport when the topic destination is not reliable.

ENABLED
This connection factory uses reliable multicast transport.

Generic JMS connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected Java Message Service (JMS) connection factory for use with the associated JMS provider. These configuration properties control how connections are created between the JMS provider and the messaging system that it uses.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > *new_JMS_Provider_instance*. Right-click **Connection Factories**, and click **New**. The following fields are displayed on the **General** tab.

A Java Message Service (JMS) connection factory creates connections to JMS destinations. The JMS connection factory is created by the associated JMS provider. A JMS connection factory for a generic JMS provider (other than the internal default messaging provider or WebSphere MQ as a JMS provider) has the following properties:

Name:

The name by which this JMS connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the associated JMS provider.

Data type String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

JNDI Name:

The application client run time uses this field to retrieve configuration information.

User ID:

Indicates the user ID used with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a userid and password explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Data type String

Password:

The password used with the **User ID** property for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type String
Default Null

Re-Enter Password:

Confirms the password entered in the **Password** field.

External JNDI Name:

The JNDI name that is used to bind the queue into the application server name space.

As a convention, use the fully qualified JNDI name, for example, `jms/`*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI API by the platform.

Data type String

Connection Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for publication or subscription).

Select one of the following options:
Queue
A JMS queue destination for point-to-point messaging.
Topic A JMS topic destination for publish subscribe messaging.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Generic JMS destination settings for application clients

Use this panel to view or change the configuration properties of the selected JMS destination for use with the associated JMS provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > *new JMS Provider instance*. Right-click **Destinations**, and click **New**. The following fields are displayed on the **General** tab.

A JMS destination is used to configure the properties of a JMS destination for the associated generic JMS provider. Connections to the JMS destination are created by the associated JMS connection factory. A JMS destination for use with a generic JMS provider (not the default messaging provider or WebSphere MQ as a JMS provider) has the following properties.

Name:

The name by which the queue is known for administrative purposes within WebSphere Application Server.

Data type String

Description:

A description of the queue, for administrative purposes.

JNDI Name:

The JNDI name of the actual (physical) name of the JMS destination bound into JNDI.

External JNDI Name:

The JNDI name that is used to bind the queue into the application server name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Destination Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for publishing or subscribing).

Select one of the following options:

Queue
A JMS queue destination for point-to-point messaging.

Topic A JMS topic destination for pub/sub messaging.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring JMS provider, JMS connection factory and JMS destination settings for application clients

You can configure JMS Provider, JMS Connection Factory and JMS Destination settings. This topic provides the required fields, special cases, and an example.

The purpose of this article is to help you to configure JMS Provider, JMS Connection Factory and JMS Destination settings.

- Required fields include:
  - JMS Provider Properties page: name, and at least one protocol provider
  - JMS Connection Factory Properties page: name, jndiName, destination type
  - JMS Destination Properties page: name, jndiName, destination type
- Special cases:
  - The destination type must be `QUEUE`, or `TOPIC`.
- Example:

```
<resources.jms:JMSProvider xmi:id="JMSProvider_3" name="genericJMSProvider:name"
description="genericJMSProvider:description"
externalInitialContextFactory="genericJMSProvider:contextFactoryClass"
externalProviderURL="genericJMSProvider:providerUrl">
<classpath>genericJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms:GenericJMSDestination"
xmi:id="GenericJMSDestination_1" name="jmsDestination:name"
jndiName="jmsDestination:jndiName" description="jmsDestination:description"
externalJNDIName="jmsDestination:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_15">
<resourceProperties xmi:id="J2EEResourceProperty_17" name="jmsDestination:customName"
value="jmsDestination:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms:GenericJMSConnectionFactory"
xmi:id="GenericJMSConnectionFactory_1" name="jmsCF:name" jndiName="jmsCF:jndiName"
description="jmsCF:description" userID="jmsCF:user" password="{xor}NTIsHBllMT4yOg=="
externalJNDIName="jmsCF:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_16">
<resourceProperties xmi:id="J2EEResourceProperty_18" name="jmsCF:customName"
value="jmsCF:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_17">
<resourceProperties xmi:id="J2EEResourceProperty_19"
name="genericJMSProvider:customName" value="genericJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Configuring new JMS connection factories for application clients

Use this task to create a new Java Message Service (JMS) connection factory configuration for your application client.

1. Click the JMS provider for which you want to create a connection factory in the tree. Complete one of the following actions:
   - Configure a new JMS provider.
   - Click an existing JMS provider.
2. Expand the JMS provider to view its **Connection Factories** folder.
3. Click the connection factory folder, and complete one of the following actions:
   - Right-click the folder and select**New**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS connection factory properties in the displayed fields.

5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring new JMS destinations for application clients

Use this task to create a new Java Message Service (JMS) destination configuration for your application client.

1. Click the JMS provider in the tree for which you want to create a destination. Complete one of the following actions:
   - Configure a new JMS provider.
   - Click an existing JMS provider.
2. Expand the JMS provider to view its **Destinations** folder.
3. Click the provider folder, and complete one of the following actions:
   - Right-click the folder and select **New**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS destination properties in the displayed fields.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring new resource environment providers for application clients

You can create new resource environment provider configurations for your application client using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create new resource environment provider configurations for your application client.

About this task

To configure a new resource environment provider, perform the following steps:

1. Start the Application Configuration Resource Tool and open the EAR file for which you want to configure the new Java Message Service (JMS) provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file in which you want to configure the new JMS provider.
3. Expand the JAR file to view its contents.
4. Click the **Resource Environment Providers** folder. Take one of the following actions:
   - Right-click the provider folder, and click **New**.
   - Click **Edit > New** on the menu bar.
5. Configure the JMS provider properties in the displayed fields.
6. Click **OK** when you finish.
7. Click **File > Save** on the menu bar to save your changes.

Resource environment provider settings for application clients

Use this page to specify resource environment entry properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected Java Archive (JAR) file. Right-click **Resource Environment Providers**, and click **New**. The following fields are displayed on the **General** tab:

Name:

Specifies the administrative name for the resource environment provider.

Description:

Specifies a description of the resource environment provider for your administrative records.

Class Path:

Specifies the path to the JAR file that contains the implementation classes for the resource environment provider.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new resource environment entries for application clients

You can create new resource environment entries for your client application using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create new resource environment entries for your client application.

About this task

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new resource environment entry. The EAR file contents are in the displayed tree view.
3. Click the desired resource environment provider, and complete the following action to configure new providers:
   - Configure a new resource environment provider.
4. Expand the resource environment provider to view the **Resource Environment Entries** folder.
5. Click the resource environment entries folder, and complete one of the following actions:
   - Right-click the folder and select **New**.
   - Click **Edit > New** on the menu bar.
6. Configure the resource environment entry properties in the displayed fields.
7. Click **OK**.
8. Click **File > Save** on the menu bar to save your changes.

Resource environment entry settings for application clients

Use this page to specify resource environment entry properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Environment Providers** > *resource environment instance*. Right-click **Resource Environment Entries**, and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the administrative name for the resource environment entry.

Description:

Specifies a description of the URL for your administrative records.

JNDI Name:

Specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts.

Use this name to link to the binding information of the platform. The binding associates the resources defined in the deployment descriptor of the module to the actual (or physical) resources bound into JNDI by the platform.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Managing application clients

You can manage Java Platform, Enterprise Edition (Java EE) application clients using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

Perform the following tasks after deploying application clients. This task only applies to Java EE application clients.

After deploying application clients on z/OS, you might want to or need to update the resources that you configured for those clients. To do so, you may complete one of the following tasks:

- Run the Application Client Resource Configuration Tool (ACRCT) on Windows, according to the following steps, and then reinstall the application on z/OS; or
- Review “Starting the Application Client Resource Configuration Tool and opening an EAR file” on page 298.

1. Update data source and data source provider configurations.
2. Update URLs and URL provider configurations.
3. Update mail session configurations.
4. Update JMS provider, connection factories, and destination configurations.
5. Update MQ JMS provider, MQ connection factories and MQ destination configurations.
6. Update Resource Environment Entry and Resource Environment Provider configurations.
7. (Optional) Remove application client resources.

Updating data source and data source provider configurations with the Application Client Resource Configuration Tool

You can update the configuration of an existing data source or data source provider using the Application Client Resource Configuration Tool (ACRCT).

About this task

During this task, you update the configuration of an existing data source or data source provider. Perform this task when your database configuration changes.

1. Start the Application Client Resource Configuration Tool (ACRCT), and open the Enterprise Archive (EAR) file containing the data source or data source provider. The EAR file contents display in a tree view.
2. Select Java Archive (JAR) file from the navigation tree containing the data source or data source provider to update.
3. Expand the JAR file to view its contents until you locate the particular data source or data source provider to update. Take one of the following actions:
   - Right-click the data source object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the displayed fields. For detailed field help, see:
   - Data source provider properties
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Updating URLs and URL provider configurations for application clients

You can update URLs and URL provider configurations for application clients using the Application Client Resource Configuration Tool (ACRCT).

1. Start the tool and open the Enterprise Archive (EAR) file containing the URL or URL provider. The EAR file contents are displayed in a tree view.
2. Select from the tree the Java Archive (JAR) file containing the URL or URL provider to update.
3. Expand the JAR file to view its contents.
4. Keep expanding the JAR file contents until you locate the particular URL or URL provider to update. Take one of the following actions:
   a. Right-click the URL object and click **Properties**.
   b. Click **Edit** > **Properties** on the menu bar.
5. Update the properties in the displayed fields.
6. Click **OK** when you finish.
7. Click **File** > **Save** on the menu bar to save your changes.

Updating mail session configurations for application clients

You can update the configuration of an existing JavaMail session using the Application Client Resource Configuration Tool (ACRCT).

About this task

During this task, you update the configuration of an existing JavaMail session. You cannot update the name of the default JavaMail provider, and you cannot delete the default JavaMail provider from the navigation tree.

1. Start the tool and open the Enterprise Archive (EAR) file containing the JavaMail session. The EAR file contents are displayed in the navigation tree view.
2. Select the Java Archive (JAR) file containing the JavaMail session to update from the navigation tree.
3. Expand the JAR file to view its contents.
4. Keep expanding the JAR file contents until you locate the particular JavaMail session to update. Take one of the following actions:
   a. Right-click the object and click **Properties**
   b. Click **Edit** > **Properties** from the menu bar.
5. Update the properties in the displayed fields.
6. Click **OK** when you finish.
7. Select **File** > **Save** from the menu bar to save your changes.

Updating Java Message Service provider, connection factories, and destination configurations for application clients

You can update the configuration of an existing Java Message Service (JMS) provider, connection factory or destination using the Application Client Resource Configuration Tool (ACRCT).

About this task

During this task, you update the configuration of an existing Java Message Service (JMS) provider, connection factory or destination.

1. Start the tool and open the Enterprise Archive (EAR) file containing the Java Message Service (JMS) provider, connection factory, or destination. The EAR file contents display in a tree view.
2. Select the Java Archive (JAR) file containing the JMS provider, connection factory, or destination to update from the navigation tree.
3. Expand the JAR file to view its contents until you locate the particular JMS provider, connection factory, or destination to update. When you find it, do one of the following actions:
   - Right-click the provider, and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the displayed fields. For detailed field help, see:
   - JMS provider properties
   - WebSphere Application Server Queue connection factory properties
   - WebSphere Application Server Topic connection factory properties
   - WebSphere Application Server Queue destination properties
   - WebSphere Application Server Topic destination properties
5. Click **OK**.
6. Click **File > Save** to save your changes.

Updating WebSphere MQ as a Java Message Service provider, and its JMS resource configurations, for application clients

You can update an existing configuration of WebSphere MQ as a Java Message Service (JMS) provider, and update the configuration of WebSphere MQ connection factories or WebSphere MQ destinations.

About this task

Use this task to update an existing configuration of WebSphere MQ as a Java Message Service (JMS) provider, and to update the configuration of WebSphere MQ connection factories or WebSphere MQ destinations.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the Enterprise Archive (EAR) file containing the WebSphere MQ JMS provider, WebSphere MQ connection factory, or WebSphere MQ destination. The EAR file contents are displayed in the navigation tree view.
3. Select the Java Archive (JAR) file containing the JMS provider, connection factory, or destination to update.
4. Expand the JAR file to view its contents until you locate the particular JMS provider, connection factory, or destination that you want to update. Complete one of the following actions:
   - Right-click the appropriate object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
5. Update the properties in the displayed fields. For detailed field help, see:
   - JMS provider properties
   - MQ Queue connection factory properties
   - MQ Topic connection factory properties
   - MQ Queue destination properties
   - MQ Topic destination properties
6. Click **OK**.

7. Click **File > Save** to save your changes.

Updating resource environment entry and resource environment provider configurations for application clients

You can update the configuration of an existing resource environment entry or resource environment provider using the Application Client Resource Configuration Tool (ACRCT).

About this task

During this task, you update the configuration of an existing resource environment entry or resource environment provider.

1. Start the tool and open the Enterprise Archive (EAR) file containing the resource environment entry or resource environment provider. The EAR file contents display in a navigation tree view.
2. Select from the tree the Java Archive (JAR) file containing the resource environment entry or resource environment provider to update.
3. Expand the JAR file to view its contents until you locate the resource environment entry or resource environment provider to update. Take one of the following actions:
   - Right-click the resource environment object, and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the displayed fields. For detailed field help, see:
   - Resource environment provider properties
   - Resource environment entry properties
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Example: Configuring Resource Environment settings:

You can configure Resource Environment settings. This topic provides the required fields and an example.

The purpose of this topic is to help you configure Resource Environment settings.

- Required fields:
  - Resource Environment Provider page: **Name**
  - Resource Environment Entry page: **Name, JNDI Name**
- Example:

```
<resources.env:ResourceEnvironmentProvider xmi:id="ResourceEnvironmentProvider_1"
name="resourceEnvProvider:name" description="resourceEnvProvider:description">
<classpath>resourceEnvProvider:classpath</classpath>
<factories xmi:type="resources.env:ResourceEnvEntry" xmi:id="ResourceEnvEntry_1"
name="resourceEnvEntry:name" jndiName="resourceEnvEntry:jndiName"
description="resourceEnvEntry:description">
<propertySet xmi:id="J2EEResourcePropertySet_20">
<resourceProperties xmi:id="J2EEResourceProperty_22"
name="resourceEnvEntry:customName" value="resourceEnvEntry:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_21">
<resourceProperties xmi:id="J2EEResourceProperty_23"
name="resourceEnvProvider:customName" value="resourceEnvProvider:customValue"/>
</propertySet>
</resources.env:ResourceEnvironmentProvider>
```

Example: Configuring resource environment custom settings for application clients:

You can configure resource environment custom settings.

The purpose of this topic is to help you configure resource environment custom settings.

- The custom page applies to every resource type. You can specify as many custom names and values as you need.
- Example:

```
<propertySet xmi:id="J2EEResourcePropertySet_20">
<resourceProperties xmi:id="J2EEResourceProperty_22"
name="resourceEnvEntry:customName" value="resourceEnvEntry:customValue"/>
</propertySet>
```

Removing application client resources

You can remove Java Platform, Enterprise Edition (Java EE) application client resources using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

The option to delete an item does not offer a confirmation dialog. As a safeguard, consider saving your work right before you begin this task. If you change your mind after removing an item, you can close the EAR file without saving your changes, canceling your deletion. Remember to close the EAR file immediately after the deletion, or you also lose any unsaved work that you performed since the deletion.

This task only applies to Java EE application clients.

To remove resources for application clients running on z/OS, run the Application Client Resource Configuration Tool (ACRCT) on Windows, according to the steps below, and then reinstall the application on z/OS.

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the Enterprise Archive (EAR) file from which you want to remove an object. The EAR file contents display in the navigation tree view. If you already have an EAR file open and have made some changes, click **File** > **Save** to save your work before preceding to delete an object.
2. Locate the object that you want to remove in the tree.
3. Right-click the object, and click **Delete**.
4. Click **File** > **Save**.

Installing Application Client for WebSphere Application Server

This topic describes how to install the Application Client for WebSphere Application Server using the installation image on the product CD-ROM.

Before you begin

Running client applications that communicate with a WebSphere Application Server requires that elements of the Application Server are installed on the system on which the client applications run. However, if the system does not have the Application Server installed, you can install Application Client, which provides a stand-alone client run-time environment for your client applications. See the Supported Prerequisites page for more information on supported product platforms.

The steps that follow provide enough detail to guide you through preparing for, choosing, and installing the variety of options and features provided. To prepare for installation and to make yourself familiar with installation options, complete the steps in this topic and read the related topics, before you start to use the installation tools. Specifically, read these topics before installing the product:

- Installing WebSphere Application Client from an i5/OS Qshell command line
- Best practices for installing

It is recommended that you have 350 MB of temporary disk space available before beginning the installation.

In Version 7.0, the Application Client for WebSphere Application Server is installable on a machine with a previous version of Application Client. However, you cannot install a Version 7.0 Application Client on top of a previous version of the Application Client.

1. Install Application Client for WebSphere Application Server using the launchpad.

   Note: The free download Application Client installation is not packaged with the launchPad program.

   a. Click **Launch the installation wizard for Application Client for WebSphere Application Server** from the launchpad tool to launch the InstallShield for MultiPlatforms installation wizard. This action launches the installation wizard.

      The Readme documentation to which the launchpad links is the `readme.html` file in the CD root directory. The readme directory off the root of the CD has more detailed Readme files. The Installation Guide is in the `/docs` directory of the CD root directory.

      Note: Readme file names are based on product offerings.

      When you install application clients, the current working directory must be the directory where the installer binary program is located. This placement is important because the resolution of the class files location is done in the current working directory. For example:

      ```
      cd /app_server_root/AppClient

      ./install
      ```

      or

      ```
       cd <CD mount point>/AppClient

           ./install
      ```

      Failing to use the correct working directory can cause ISMP errors that abort the installation.

      The installation wizard does not upgrade or remove previous Application Clients installation automatically.

   b. As indicated in the previous example, you can start the installation wizard from the product CD-ROM, using the command line.

   c. You can also perform a silent installation using the `-options responsefile -silent` parameter, which causes the installation wizard to read your responses from the options response file, instead of from the interactive graphical user interface. Customize the response file before installing silently. After customizing the file, issue the command to silently install. Silent installation is particularly useful if you install the product often.

      The rest of this procedure assumes that you are using the installation wizard. There are corresponding entries in the response file for every prompt that is described as part of the wizard. Review the description of the response file for more information. Comments in the file describe how to customize its options.

2. Click **Next** to continue when the Welcome panel displays.

   a. Click the radio button beside the **I accept the terms in the license agreement** message if you agree to the license agreement, and click **Next** to continue.

3. The installation wizard checks the system for prerequisites. Click **Next** if you see a success message on the wizard panel. If a warning message displays on the panel, click **Cancel** to exit the installation wizard and install the proper prerequisites to the system.

   Note: Application Client may be fully functional even if some prerequisites are not installed on the system, however it is recommended you update your system to the required level.

   Note: When completing a custom installation in GUI mode, you have installable options such as Applet Client Samples and StandAlone Thin Client Resource Adapters. If you select any of the available options and need to go backwards on the install panels (for instance, to reread the

license agreement), the installable options panel disappears when continuing through the installation panels. To access these panels again, you will have to relaunch the installation wizard.

4. Specify a destination directory. Click **Next** to continue.
   a. Ensure that there is adequate space available in the target directory.
   b. Specify a target directory for the Application Client product.
   c. Enter the required target directory to proceed to the next panel. Deleting the default target location and leaving an installation directory field empty prevents you from continuing the installation process.
5. Choose a type of installation, and click **Next**.

   If you use the GUI you can choose a Typical installation type, which installs Java EE and Thin application client, Samples and IBM Developer Kit, Java 2 Technology Edition, or you can choose the custom installation type. For Windows, there are two custom installation types: Custom Java EE/Thin Client and Custom Pluggable Client. For other platforms, there is only one custom installation type.
   - If you select the **ActiveX to EJB Bridge** feature, then the following message displays in a dialog box: Do you want to add Java runtime to the system path and make it the default JRE? If you answer Yes, then the Java run time is added to the beginning of the system path. If you answer No, then the ActiveX to EJB Bridge does not function from the Active Server Pages (ASP), unless you add the Java run time to the path. To add the Java run time later, see the topic ActiveX application clients or reinstall the Application Client.
   - If you select the **Applet client** feature, then the following message might be displayed: An existing JDK or JRE has been detected on your computer. You chose to install the Applet Client, which will overwrite the registry entries for this JDK or JRE. Do you want to continue and install the Applet Client? If you select Yes, the installation overrides the registry on your machine. If you select No, the Applet client feature is not installed, and you are directed to the back feature dialog box.
   - Install the Software Developer Kit feature, if you need to use any of the utilities that it provides, such as the javaccompiler, the jarutility or the jarsigner utility. The Java 2 Development Kit that IBM provides has two components, Java Runtime Environment (JRE) and a complete Software Developer Kit (SDK). The JRE sub feature is selected by default for the Custom Java EE/Thin Client installation type. The SDK component is optional; however, you must install the SDK component to compile the sample.
   - Install the Pluggable Client samples if you want to make use of the Pluggable Client applications.
   - Install the Software Developer Kit feature, if you need to use any of the utilities that it provides, such as the *javaccompiler*, the *jarutility* or the *jarsigner* utility. The Java 2 Development Kit that IBM provides has two components, Java Runtime Environment (JRE) and a complete Software Developer Kit (SDK). The JRE sub feature is selected by default for the Custom Java EE/Thin Client installation type. The SDK component is optional; however, you must install the SDK component to compile the sample.
6. (Pluggable Client installation type only) Click **Next** to accept the detected Sun JRE, or click Browse to select the location of the installed Sun JRE. The Sun Software Development Kit installation location is optional. However, if the installation location is not provided, the installed Samples do not compile.
   - If Sun JRE has not been installed, the installation cannot be continued. Click **Cancel** to exit the installation. Install the Sun JRE, and restart the Pluggable Custom installation. The Sun JRE panel displays with the JRE path detected, and the Pluggable application client installation continues.
7. Enter the host name of the WebSphere Application Server machine. Click **Next** to continue. The default port number is 2809.
8. Review the summary information, and click **Next** to install the product code or you might also click **Back** to change your specifications.
9. Click **Finish** to exit the wizard, after the Application Client installs.
10. Verify the success of the installer program by examining the Completion panel and the *app_server_root*/`logs/install/log.txt` file for installation status. The installer program records the following indicators of success in the logs:

- `INSTCONFSUCCESS` indicates that the installation is successful and that no further log analysis is required.
- `INSTCONFFAILED` indicates an installation failure that you cannot retry or recover from without reinstalling.

The installation wizard records installation events in the installation log files. Examine messages that the installation program displays. If the Application Client for WebSphere Application Server does not install successfully, read the messages to identify why the installation failed. Correct the problems identified and try installing the product again.

Note: If you are logged in as non-root or non-admin user, you may not get a successful installation. When the installer runs, a log file is created in your home directory. If the installation fails, an attempt is made to move the log to the *app_server_root* directory; however, without the necessary permission to move the file to the *app_server_root* directory, the attempt to move fails and the log remains in your home directory. Look for the following log files in the *user_home*`/waslogs` directory:

- `log.txt`
- `trace.txt.gz`
- `trace.xml.gz`

Results

You successfully installed the Application Client for WebSphere Application Server and the features you selected.

What to do next

Use the installation verification utility to verify a successful installation. If the installation is not successful, fix the error as indicated in the installation error message. For example, if you do not have enough disk space, add more space, and reinstall the Application Client.

Best practices for installing Application Client for WebSphere Application Server

This topic provides tips for installing Application Client on multiple platforms.

All platforms

The following tips pertain to installing Application Client on all platforms:

- Reserve at least 4 to 5MB free space in the target platform temporary directory.
- When specifying a different temporary directory while installing Application Client, the following message is displayed if the target platform default temporary directory does not have enough free space to install Application Client:

```
Error writing file =  There may not be enough
temporary disk space.
Try using -is:tempdir to use a temporary
directory on a partition with more disk
space.
```

  Use the `-is:tempdir` installation option to specify a different temporary directory. For example, the following command uses `/swap` as a temporary directory during installation:

```
./install -is:tempdir /swap
```

- After the installation, when changing the installation settings for the WebSphere Application Server host name and the port number, edit the setupClient file.

Change the DEFAULTSERVERNAME and SERVERPORTNUMBER to the new WebSphere Application Server host name and port number, respectively. If the SERVERPORTNUMBER is not set, then the default is 2809. Review the following example:

```
set DEFAULTSERVERNAME=NDServerName
set SERVERPORTNUMBER=9810
```

The setupClient file is located in the bin subdirectory under the Application Client installation destination.

Specific platforms

The following tips pertain to installing Application Client on the designated platforms:

Installing Application Client for WebSphere Application Server silently

This topic provides the steps necessary to use the installation wizard and perform a silent installation.

About this task

Use these steps to perform a silent installation, which uses the installation wizard to install the product. Instead of displaying a user interface, the silent installation provides interaction between you and the wizard by reading all of your responses from a file that you must customize.

1. Verify that the user ID that you are using to run the silent installation has sufficient authority to perform the task.
2. Customize the option response file.
   a. Locate the sample options response file. The file name is setup.response in the operating system platform directory on the product CD-ROM.
   b. Make a copy to preserve the original response file. For example, copy the file as myoptionsfile.
   c. Edit the copy in your flat file editor of choice, on the target operating system. Read the directions within the response file to choose appropriate values.
   d. Make a non-commented option to have a silent install.
   e. Include custom option responses that reflect parameters for your system.
   f. Follow the instructions in the response file to choose appropriate values.
   g. Save the file.
3. Issue a command to use your custom response file: Install.exe -options myoptionsfile -silent for Windows platforms and install -options ./myoptionsfile -silent for Linux® and UNIX® platforms.

   The sample options response file is located in the AppClient directory on the product CD-ROM.
   a. Issue the following command from a command prompt to update your response file: -OPT silentInstallLicenseAcceptance="true' .

      Issuing this command indicates that you accept all IBM license terms associated with this product, which is necessary for installing application clients.
4. Optional: Restart your machine in response to the prompt that appears on Windows platforms when the installation is complete.

Results

You installed application clients silently by using the response file.

What to do next

To verify the silent install, look for the string INSTCONFSUCCESS in the log.txt file for successful installation and INSTCONFFAILED for a failed installation.

When the InstallShield for MultiPlatforms (ISMP) fails and the log.txt file is not created, the error log file might have been created in one of the following directories:

- When a GUI install fails the logs are in var\tmp\niflogs\log.txt.
- When a silent install fails the logs are in <user_home>\waslogs\log.txt.
- When a silent install for Plugins fails the logs are in <user_home>\plglogs\log.txt.
- When a silent install for HTTP Server fails the logs are in <user_home>\ihslogs\log.txt.

Uninstalling Application Client for WebSphere Application Server

This task describes using the uninstall program to uninstall the Application Client for WebSphere Application Server.

Before you begin

Note: If you have a feature pack installed, uninstalling the Application Client for WebSphere Application Server using the Version 6.1.0 uninstaller program causes the feature pack to stop working because the uninstallation removes the server. However, you can uninstall the feature pack after uninstalling the WebSphere Application Server product.

Note: If you are logged in as a root user and attempt to uninstall an instance of Application Client for WebSphere Application Server that a non-root user installed, the Windows registry will also be wiped clean for the application client instance that the root user installed. If you need to uninstall an instance of application client that a non-root user has installed, be sure that you are logged in as the non-root user.

1. Stop any browsers and any Java processes related to Application Client products.
2. Change directories to the `uninstall` directory before issuing the command to uninstall the application client. The command file is located in the `uninstall` directory.
   - *app_server_root*/`uninstall` directory

     For example, to change directories before uninstalling the product from a Linux platform, issue this command if your installation root is `/opt/IBM/WebSphere/AppClient`:

     ```
     cd /opt/IBM/WebSphere/AppClient/uninstall
     ```
3. Issue the command to uninstall the product.

   The command file is named `uninstall`. Issue the **uninstall** command from the *app_server_root*/`uninstall` directory:

   ```
   ./uninstall
   ```

   The Uninstall wizard begins and displays the Welcome panel.
4. Click **Next** to begin uninstalling the product. The Uninstall wizard displays a Confirmation panel that lists the product and features that you are uninstalling.
5. Click **Next** to continue uninstalling the product. The Uninstall wizard deletes existing profiles first.

   After deleting profiles, the Uninstall wizard deletes core product files by component.
6. Click **Finish** to close the wizard after the wizard removes the product.

Results

Application Client for WebSphere Application Server is uninstalled.

What to do next

Verify the uninstall procedure by viewing the *app_server_root*/`logs/uninstall/log.txt` file for errors. Look for the `INSTCONFSUCCESS`, indicating a successful uninstall in the log file:

```
Uninstall, com.ibm.ws.install.ni.ismp.actions.ISMPLogSuccessMessageAction, msg1,
   INSTCONFSUCCESS
```

The uninstallation wizard records uninstallation events in the uninstallation log files. Examine messages that the uninstallation program displays. If the Application Client for WebSphere Application Server does not uninstall successfully, read the messages to identify why the uninstallation failed. Correct the problems identified and try uninstalling the product again.

Note: If you are logged in as non-root or non-admin user, the uninstall may not be successful. When the uninstaller program runs, a log file is created in your home directory. If the uninstaller program fails, an attempt is made to move the log to the *app_server_root* directory; however, without the necessary permission to move the file to the *app_server_root* directory, the attempt to move fails and the log remains in your home directory. Look for the following log files in the *user_home*/waslogs directory:

- log.txt
- trace.txt.gz
- trace.xml.gz

Running application clients

The Java Platform, Enterprise Edition (Java EE) specification requires support for a client container that runs stand-alone Java applications (known as Java EE application clients) and provides Java EE services to the applications. Java EE services include naming, security, and resource connections.

About this task

You are ready to run your application client using this tool after you:
1. Write the application client program.
2. Assemble and install an application module (.ear file) in the application server run time.
3. Deploy the application using the Application Client Resource Configuration Tool (ACRCT) on Windows.

   Alternatively, you can use the ACRCT scripting tool on z/OS.

This task only applies to Java EE application clients. To launch Java EE application clients using the launchClient script, perform the following steps:

1. Enter the following command to launch Java EE application clients:

   app_server_root/bin/launchClient
2. Pass parameters to the launchClient command or to your application client program as well. The launchClient command allows you to do both. The launchClient command requires that the first parameter is either:
   - An EAR file specifying the application client to launch.
   - A request for launchClient usage information.

   The following example illustrates the command line invocation syntax for the launchClient tool:

   launchClient [-profileName pName | -JVMOptions options | -help | -?] <*userapp*> [-CC<name>=<value>] [app args]

   where
   - *userapp* is the path and the name of the EAR file that contains the application client.
   - *-CC<name>=<value>* is the client container name-value pair parameter. See the client container parameters section, for supported name-value pair arguments.
   - *app args* are arguments that pass to the application client.
   - *-profileName* defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment or in an Application Clients installation.

     The default is **default_profile**.
   - *-JVMOptions* is a valid Java standard or non-standard option string. Insert quotation marks around the string.
   - *-help, -?* prints the usage information.

All other parameters intended for the `launchClient` command must begin with the -CC prefix.

Parameters that are not EAR files, or usage requests, or that do not begin with the -CC prefix, are ignored by the application client run time, and are passed directly to the application client program.

The `launchClient` command retrieves parameters from three places:
- The command line
- A properties file
- System properties

The parameters are resolved in the order listed above, with command line values having the highest priority and system properties the lowest. Using this prioritization you can set and override default values.

3. Specify the server name.

   By default, the **launchClient** command uses the localhost for the `BootstrapHost` property value.

   This setting is effective for testing your application client when it is installed on the same computer as the server. However, in other cases override this value with the name of your server. You can override the `BootstrapHost` value by invoking `launchClient` command with the following parameters:

   ```
   launchClient myapp.ear -CCBootstrapHost=abc.midwest.mycompany.com
   ```

   You can also override the default by specifying the value in a properties file and passing the file name to the launchClient shell.

   Security is controlled by the server. You do not need to configure security on the client because the client assumes that security is enabled. If server security is not enabled, then the server ignores the security request, and the application client functions as expected.

Example

You can store launchClient values in a properties file, which is a good method for distributing default values. You can then override one or more values on the command line. The format of the file is one `launchClient -CC` parameter per line without the `-CC` prefix. For example:

```
 verbose=true classpath=/usr/lpp/mydir/util.jar;/usr/lpp/mydir/harness.jar;/usr/lpp
/production/G19/global.jar BootstrapHost=abc.westcoast.mycompany.com tracefile=/usr
/lpp/WebSphere/mylog.txt
```

launchClient tool

This topic describes the Java Platform, Enterprise Edition (Java EE) command line syntax for the launchClient tool for WebSphere Application Server.

You can use the `launchClient` command from a node within a Network Deployment environment. **However, do not attempt to use the `launchClient` command from the Deployment Manager.**

Note: All users who run commands from a specific profile must have authority to modify files that are created by other users that use the same profile. Otherwise, you might see a permission denied error in the log files. To avoid this issue, consider one of the following policies:
- Use a separate installation for distinct user authorities
- Always use the same user for all of the commands that are run in a given profile
- Ensure that all users of a specific profile belong to the same group. In addition, ensure that each user of a group has the read and write authority to the files that are created by other members in the same profile.

The following example illustrates the command line invocation syntax for the launchClient tool:

```
launchClient [-profileName pName | -JVMOptions options | -help | -?] <userapp> [-CC<name>=<value>] [app args]
```

where

- *userapp* is the path and the name of the EAR file that contains the application client.
- *-CC<name>=<value>* is the client container name-value pair parameter. See the client container parameters section, for supported name-value pair arguments.
- *app args* are arguments that pass to the application client.
- *-profileName* defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment or in an Application Clients installation.

  The default is **default_profile**.
- *-JVMOptions* is a valid Java standard or nonstandard option string, except `-cp` or `-classpath`. Insert quotation marks around the string.
- *-help, -?* prints the usage information.

The first parameter must be `-help`, `-?` or contain no parameter at all. The `-profileName pName` and `-JVMOptions` options are optional parameters. If used, they must appear before the *<userapp>* parameter. All other parameters are optional and can appear in any order after the *<userapp>* parameter. The Java EE Application client run time ignores any optional parameters that do not begin with a `-CC` prefix and passes those parameters to the application client.

Client container parameters

Supported arguments include:

-CCadminConnectorHost
Specifies the host name of the server from which configuration information is retrieved.

The default is the value of the `-CCBootstrapHost` parameter or the value, `localhost`, if the `-CCBootstrapHost` parameter is not specified.

-CCadminConnectorPort
Indicates the port number for the administrative client function to use. The default value is `8880` for SOAP connections and `2809` for Remote Method Invocation (RMI) connections.

-CCadminConnectorType
Specifies how the administrative client connects to the server. Specify `RMI` to use the RMI connection type, or specify `SOAP` to use the SOAP connection type. The default value is `SOAP`.

-CCadminConnectorUser
Administrative clients use this user name when a server requires authentication. If the connection type is SOAP, and security is enabled on the server, this parameter is required.

-CCadminConnectorPassword
The password for the user name that the `-CCadminConnectorUser` parameter specifies.

-CCaltDD
The name of an alternate deployment descriptor file. This parameter is used with the `-CCjar` parameter to specify the deployment descriptor to use. Use this argument when a client JAR file is configured with more than one deployment descriptor. Set the value to `null` to use the client JAR file standard deployment descriptor.

-CCBootstrapHost
The name of the host server you want to connect to initially. The format is:
your_server_of_choice.com

-CCBootstrapPort
The server port number. If you do not specify this argument, the WebSphere Application Server default value is `used`.

-CCclassLoaderMode
Specifies the class loader mode. If PARENT_LAST is specified, the class loader loads classes from the local class path before delegating the class loading to its parent. The classes loaded for the following are affected:

- Classes defined for the Java EE application client
- Resources defined in the Java EE application
- Classes specified on the manifest of the Java EE client JAR file
- Classes specified using the -CCclasspath option

If PARENT_LAST is not specified, then the default mode, PARENT_FIRST, causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path.

-CCclasspath
A class path value. When you launch an application, the system class path is used. If you want to access classes that are not in the EAR file or part of the system class paths, specify the appropriate class path here. Multiple paths can be concatenated.

-CCD
Use this option to have the WebSphere Application Server set the specified system property during initialization. Do not use the equals (=) character after the `-CCD`. For example: `-CCDcom.ibm.test.property=testvalue`. You can specify multiple `-CCD` parameters. The general format of this parameter is `-CCD<property key>=<property value>`. For example, `-CCDI18NService.enable=true`.

-CCdumpJavaNameSpace
Controls generation of a dump of the java: name space for the application that is launched, which can be used for debugging purposes. A value of **true** generates a dump in short format, and includes the name and object type for each binding. A value of **long** generates a dump in long format, and includes additional information for each binding over short format, such as the local object type and string representation of the local object. The default value is **false**, and does not generate a dump.

-CCexitVM
Use this option to have the WebSphere Application Server call the `System.exit()` method after the client application completes. The default is `false`.

-CCinitonly
Use this option to initialize application client run time for ActiveX application clients without launching the client application. The default is `false`.

-CCjar
The name of the client Java Archive (JAR) file that resides within the EAR file for the application you wish to launch. Use this argument when you have multiple client JAR files in the EAR file.

-CCpropfile
Indicates the name of a properties file that contains launchClient properties. Specify the properties without the `-CC` prefix in the file, with the exception of the securityManager, securityMgrClass and securityMgrPolicy properties. See the following example: `verbose=true`.

-CCproviderURL
Provides bootstrap server information that the initial context factory can use to obtain an initial context. WebSphere Application Server initial context factory can use either a Common Object Request Broker Architecture (CORBA) object URL or an Internet Inter-ORB Protocol (IIOP) URL. CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. This value can contain more than one bootstrap server address. This feature can be used when attempting to obtain an initial context from a server cluster. You can specify bootstrap server addresses, for all servers in the cluster, in the URL. The operation will succeed if at least one of the servers is running, eliminating a single point of failure. The address list does not process in a particular order. For naming operations,

this value overrides the -CCBootstrapHost and -CCBootstrapPort parameters. A CORBA object URL specifying multiple systems is illustrated in the following example:

```
-CCproviderURL=corbaloc:iiop:myserver.mycompany.com:9810,:mybackupserver.mycompany.com:2809
```

This value is mapped to the java.naming.provider.url system property.

-CCsecurityManager
Enables and runs the WebSphere Application Server with a security manager. The default is disable.

-CCsecurityMgrClass
Indicates the fully qualified name of a class that implements a security manager. Only use this argument if the -CCsecurityManager parameter is set to enable. The default is java.lang.SecurityManager.

-CCsecurityMgrPolicy
Indicates the name of a security manager policy file. Only use this argument if the -CCsecurityManager parameter is set to enable. When you enable this parameter, the java.security.policy system property is set. The default is *<app_server_root>*/ properties/client.policy.

-CCsoapConnectorPort
The Simple Object Access Protocol (SOAP) connector port. If you do not specify this argument, the WebSphere Application Server default value is used.

-CCtrace
Use this option to obtain debug trace information. You might need this information when reporting a problem to IBM customer support. The default is false. For more information, read the topic ″Enabling trace.″

-CCtracefile
Indicates the name of the file to which trace information is written. The default is to write output to the console.

-CCtraceMode
Specifies the trace format to use for tracing. If the valid value, basic, is not specified the default is advanced. Basic tracing format is a more compact form of tracing.

-CCverbose
This option displays additional information messages. The default is false.

If you are using an EJB client application with security enabled, edit the sas.client.props file, which is located in the *profile_root*/properties directory. Within the file, change the com.ibm.CORBA.loginSource value to *none*.

For more information on the sas.client.props utility, see PropFilePasswordEncoder command reference.

RMI connection with security. Used with the EJB and administrative client application.

Using Jacl:

```
wsadmin.sh -conntype RMI -port  rmiportnumber -user userid
-password password
```

Using Jython:

```
wsadmin.sh -lang jython -conntype RMI -port  rmiportnumber -user userid
-password password
```

rmiportnumber for your connection displays in the administrative console as BOOTSTRAP_ADDRESS.

Note: On the AIX®, HP-UX, Linux, i5/OS, Solaris, and z/OS operating systems, the use of -password option may result in security exposure as the password information becomes visible to the system status program, such as ps command, which can be invoked by other users to display all of the running processes. Do not use this option if security exposure is

a concern. Instead, specify user and password information in the `soap.client.props` file for SOAP connector or `sas.client.props` file for RMI connector. The `soap.client.props` and `sas.client.props` files are located in the properties directory of your WebSphere Application Server profile.

If Kerberos (KRB5) is enabled for administrative authentication, the authentication target supports BasicAuth and KRB5. To use KRB5, update the `sas.client.props`, `soap.client.props`, and `ipc.client.props` files, according to the connector type.

Note: When using Kerberos authentication, the user password does not flow across the wire. A one-way hash of password is used to identify the client.

The following examples demonstrate correct syntax.

Specifying the directory for an expanded EAR file

You can archive the `Manifest.mf` client Java Archive (JAR) files instead of automatically cleaning them up after the application exits.

Before you begin

Each time the launchClient tool is called, it extracts the Enterprise Archive (EAR) file to a random directory name in the temporary directory on your hard drive. Then the tool sets up the thread ClassLoader to use the extracted EAR file directory and JAR files included in the `Manifest.mf` client Java Archive (JAR) file. In a normal J2EE Java client, these files are automatically cleaned up after the application exits. This cleanup occurs when the client container shutdown hook is called. To avoid extracting the EAR file (and removing the temporary directory) each time the launchClient tool is called, complete the following steps:

1. Specify a directory to extract the EAR file by setting the `com.ibm.websphere.client.applicationclient.archivedir` Java system property. If the directory does not exist or is empty, the EAR file is extracted normally. If the EAR file was previously extracted, the launchClient tool reuses the directory.
2. Delete the directory before running the launchClient tool again, if you need to update your EAR file. When you call the `launchClient` command, it extracts the new EAR file to the directory. If you do not delete the directory or change the system property value to point to a different directory, the launchClient tool reuses the currently extracted EAR file and does not use your changed EAR file. When specifying the `com.ibm.websphere.client.applicationclient.archivedir` property, make sure that the directory you specify is unique for each EAR file you use. For example, do not point the `MyEar1.ear` and the `MyEar2.ear` files to the same directory.

Using Java Web Start

Learn about the Java Web Start technology that is provided by the Java Standard Edition runtime environment to deploy Java Enterprise Edition application clients, including Thin application clients, on the remote client machine with a single click from Web browser on the client machine.

Before you begin

The supported client platforms for deploying application clients using the Java Web Start are the same as the IBM Application Client for WebSphere Application Server supported platforms, except Linux on Power and OS/400® operating systems.

Before you begin this task, see the following topics to understand Java Web Start technology and its components:

- “Java Web Start architecture for deploying application clients” on page 382
- “Client application Java Network Launcher Protocol deployment descriptor file” on page 383
- “ClientLauncher class” on page 387

You can use the following resources:

- The IBM implementation of Java Web Start on Java Platform Standard Edition 6 (Java SE 6) that is packaged in the Application Client for WebSphere Application Server.

Note: When using the Sun JRE with the Application Client runtime, application resources cannot be downloaded using the https protocol. This is a Sun Microsystems limitation. Install the IBM JRE on the client machine and use IBM Java Web Start to run the WebSphere Application Client application. Sun Java Web Start is supported, but with a limitation that only the http protocol can be used to download the application resource, including the application jnlp description file.

About this task

To deploy application clients using Java Web Start, the client machine must have at least a Java SE runtime environment installed. The Java SE runtime environment includes the Java Web Start, which implements the JSR 56: Java Network Launching Protocol and API. The application clients Enterprise Archive (EAR) file is a Java archive (JAR) resource in a JNLP descriptor file that resides on a central server. The JNLP descriptor file also specifies the runtime environment requirement for running the application.

WebSphere Application Server provides a launcher class to launch the Java EE application client in the application client container inside of Java Web Start. The client machine might not have the IBM Application Client for WebSphere Application Server installed. If this is the case, create and install an application client container and runtime package as a runtime environment through Java Web Start. The JNLP descriptor file specifies this runtime environment as the required runtime environment for running the Java EE application client.

WebSphere Application Server also provides command-line utility programs to create this application client container and runtime package from an existing IBM Application Client for WebSphere Application Server installation, as well as an installer class to install this package as a runtime environment for the application client container and also the Java Runtime Environment (JRE) in the IBM Application Client for WebSphere Application Server installation. To run the Java EE application client, the EAR file is deployed as a JAR resource that is described in the JNLP descriptor file.

1. Identify the client machine operating system, and install the corresponding IBM Application Client for WebSphere Application Server on a development machine. For example, if the Java EE application clients are targeted to run on Windows operating systems, install the IBM Application Client for WebSphere Application Server for Windows. Follow the instructions for “Installing Application Client for WebSphere Application Server” on page 369.
2. Run the utility programs to create the application client container and runtime package.
   a. Use the “buildClientRuntime tool” on page 390 utility to create the package.
   b. Use the “buildClientLibJars tool” on page 383 utility to create the JAR files containing the launcher and the installer class. This utility also zips up the properties files in the <app_client_root>/ properties directory.
3. Create the runtime installer JNLP descriptor file. The JNLP response must be included in the JNLP version ID to indicate the current runtime version in the response header, for example, `x-java-jnlp-version-id=1.6.0`. Using a servlet of a Java Server Pages (JSP) file to provide a dynamic JNLP response.
4. Create the Java EE application client launch JNLP descriptor file.
5. Package the application client container runtime environments and the Java EE application in an Enterprise Archive (EAR) file. Depending on your preferred deployment strategy, the files can be in two separate Web modules, or combined into one.
6. All JAR resources must be Java signed, including the Java EE application client EAR file.
7. Deploy the Enterprise Archive file on an application server, and start the application. The Java EE application client is ready to be deployed.

Example

A Java Web Start deployment Sample is included in the client samples. This Sample demonstrates the steps to deploy a Java EE application client with an automated ANT script. The Sample has a servlet to generate the runtime installer JNLP response with JNLP version ID, for example, `x-java-jnlp-version-id`.

Note: When the application client initially launches using Java Web Start from Sun Microsystems Java SE Runtime Environment 6.0, it installs the Application Client runtime, which includes the IBM JRE. An null pointer exception (NPE) is thrown from the `com.sun.deploy.services.WPlatformService.getSecureRandom()` method. This is a known bug in Sun Java SE 6 (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6505528). If you experience this exception, relaunch the application. The NPE only occurs on the first launch of the application client.

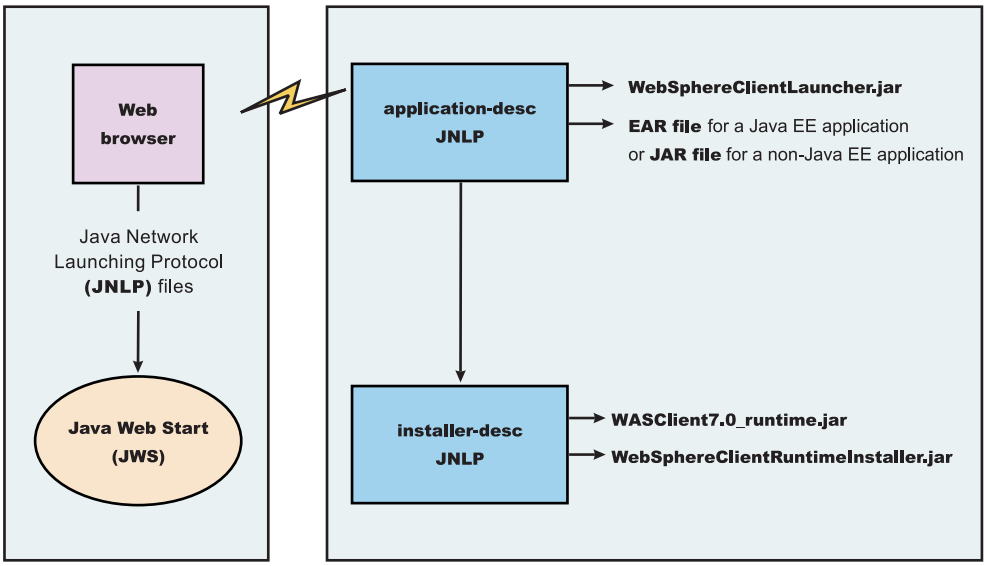
Java Web Start architecture for deploying application clients

Java Web Start is an application-deployment technology that includes the portability of applets, the maintainability of servlets and JavaServer Pages (JSP) file technology, and the simplicity of mark-up languages such as XML and HTML. It is a Java application that allows full-featured Java EE client applications to be launched, deployed and updated from a standard Web server. The Java Web Start client is used with platforms that support a Web browser.

Upon launching Java Web Start for the first time, you might download new client applications from the Web. Each time you launch JWS thereafter, you can initiate applications either through a link on a Web page or (in Windows) from desktop icons or the Start menu. You can deploy applications quickly using Java Web Start, cache applications on the client machine, and launch applications remotely offline. Additionally, because Java Web Start is built from the Java Platform, Enterprise Edition (Java EE) infrastructure, the technology inherits the complete security architecture of the Java EE platform.

The technology underlying Java Web Start is the Java Network Launching Protocol & API (JNLP). Java Web Start is a JNLP client and it reads and parses a JNLP descriptor file (JNLP file). Based on the JNLP descriptor, it downloads appropriate pieces of a client application and any of its dependencies. If any of the pieces of the application are already cached on the client machine, then those components are not downloaded again, unless they have been updated on the server machine. After you download and cache the client application, JWS launches it natively on the client machine.

The following diagram shows an overview of launching a client application, include the Application Client for WebSphere Application Server as a dependent resource, using Java Web Start.



The Web browser running on a client machine connects to a Web application located on a server machine. The client application JNLP descriptor file is downloaded and processed by Java Web Start on the client machine.

In this diagram, there are two JNLP descriptor files:

- Client application JNLP descriptor (application-desc in the diagram)
- Application Clients run-time installer JNLP descriptor (installer-desc in the diagram)

Each of these JNLP descriptor files, the client application (JAR or EAR) and the dependent resource JAR files are packaged as Web applications in an EAR file. This EAR file is deployed to an Application server. The client machine with JWS installed uses a Web browser to connect to the url of the client application JNLP descriptor file to download and run the client application.

Using Java Web Start from Java SE Runtime Environment 6.0 or later is highly recommended. All the platforms supported by the application client for WebSphere Application Server are supported.

You can use the following:

- Java Web Start on the Java Standard Edition Developer Kits that IBM provides, packaged in Application Client for WebSphere Application Server
- Java Web Start on Sun Microsystems Java SE 6 Development Kit or Java SE Runtime Environment 6.0, which you can download from the Sun Microsystems Web site for Windows, Linux and Solaris operating systems
- Java Web Start on HP-UX JDK or JRE for Java Platform, Standard Edition, version 6, which you can download from the HP Web site

buildClientLibJars tool:

For a Java Platform, Enterprise Edition (Java EE) application client application and or Thin application client application to be launched using Java Web Start (JWS), the properties files bundled in Application Client for WebSphere Application Server must be installed in the Java Web Start. Use this tool to create those property JAR files. The Java Web Start client is used with platforms that support a Web browser.

The buildClientLibJars tool copies the JAR files from the Application Client for WebSphere Application Server installation and creates a `properties.jar` file, which contains the properties files from the Application Clients installation properties directory to a specified location. When this property is created, the tool uses the value of `keystore`, `storepass`, `alias` and `storetype` to sign all of the JAR files in the specified location.

```
Windows usage: buildClientLibJars.bat  [-help] [-verbose] destdir keystore storepass alias storetype

Unix usage:    buildClientLibJars.sh  [-help] [-verbose] destdir keystore storepass alias storetype
```

where:

- `-help` will display the message
- `-verbose` will turn on verbose message
- `destdir` will output the destination directory name
- `keystore` is the key store file
- `storepass` is the key store password
- `alias` key is the alias name
- `storetype` is the key store type

Client application Java Network Launcher Protocol deployment descriptor file

The deployment descriptor file is the main Java Network Launcher Protocol (JNLP) descriptor file for the client application.

Location

The client application has an Application Clients run-time dependency that provides the following:

- Java SE Runtime Environment from IBM
- Application Clients run-time properties
- SSL KeyStore and TrustStore file
- Application Clients run-time library JAR files (optional for Thin Application client applications)

If the Application Clients run-time dependency is not met, it is downloaded and installed in Java Web Start (JWS), as described by the Application Clients run-time installer JNLP descriptor file. For example:

```
<j2se version="1.6" href="http://your_server.com/jws/wasappclient/download.jnlp"/>
```

Usage notes

The client application must also include the `WebSphereClientLauncher.jar` file, which contains the launcher class, com.ibm.websphere.client.launcher.ClientLauncher, that completes one of the following actions:

- If it is a Java Platform, Enterprise Edition (Java EE) Application client application (that is the resources for the application contain an EAR file with a client application), the EAR file must be specified as a JAR resource so that it can be downloaded to JWS and specified in the system property, `com.ibm.websphere.client.launcher.ear`. See “JNLP descriptor file for a Java EE Application client application” on page 385 for an example.
- If it is a Thin Application client application, the Thin Application client application JAR file must be specified as a JAR resource so that it can be downloaded to JWS and the name of the class containing main method entry point is specified in the system property, `com.ibm.websphere.launcher.main`. See “JNLP descriptor file for a Thin Application client application” on page 386 for an example.

The JNLP specification requires all the resource (JAR or EAR) files used in a JNLP file to be signed.

You can specify the –CC arguments defined in the launchClient tool for a J2EE Application client application in application arguments section of the JNLP descriptor files. However, only –CCD is supported for a Thin Application client application to define system properties and the JNLP <property> tag can also be used to define system properties. See the following example for details:

```
<property name="java.naming.provider.url" value="corbaloc:iiop:myserver.com:9089"/>
```

For a J2EE Application client application, specify the following application arguments as defined in the JNLP.

1. Specify your target server provider URL, as shown in the following example:

   ```
   <argument> >-CCDjava.naming.provider.url =corbaloc:iiop:myserver.mydomain.com:9080 </argument>
   ```

2. Specify the SSL Key File and SSL Trust File location. These files are expected to be available in the client machine. To use the ones in the Application Clients run-time dependency installed in JWS cache, specify these application arguments:

   ```
   <argument> -CCDcom.ibm.ssl.keyStore=${WAS_ROOT}/etc/key.p12 </argument>
   <argument>-CCDcom.ibm.ssl.trustStore=${WAS_ROOT}/etc/trust.p12 </argument>
   ```

3. Specify the initial naming context factor, as shown in the following example:

   ```
   <argument>-CCDjava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory </argument>
   ```

   For a Thin Application client application, you also need to specify the actual location of the `sas.client.props` and `ssl.client.props` files located in the Application Clients run-time dependency that is installed in the JWS cache.

   ```
   <argument>-CCDcom.ibm.CORBA.ConfigURL=file:${WAS_ROOT}/properties/sas.client.props </argument>
   <argument>-CCDcom.ibm.SSL.ConfigURL=file:${WAS_ROOT}/properties/ssl.client.props </argument>
   ```

   If any of the default settings in the `sas.client.props` and `ssl.client.props` file need modifying, use the –CCD to change the settings through the system properties, as shown in the following example:

```
<argument>-CCDjavacom.ibm.CORBA.securityEnabled=false </argument>
```

Note: The ${WAS_ROOT} token used in the JNLP file is replaced by the launcher class, com.ibm.websphere.client.launcher.ClientLauncher, to the actual location of the Application Clients run-time dependency installation in the JWS cache. If you are using JSP to dynamically create this JNLP description file, you must escape this token because it has a different meaning in JSP 2.0. See the following example for details:

```
<argument>-CCDcom.ibm.ssl.keyStore=\${WAS_ROOT}/etc/key.p12 </argument>
<argument>-CCDcom.ibm.ssl.trustStore=\${WAS_ROOT}/etc/trust.p12 </argument>
```

JNLP descriptor file for a Java EE Application client application:

The deployment descriptor file is the main Java Network Launcher Protocol (JNLP) descriptor file for the client application.

Here is an example of the client application JNLP descriptor file for a Java EE Application client application:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This sample program is provided AS IS and may be used, executed, copied and modified
without royalty payment by customer (a) for its own instruction and study, (b) in order
to develop applications designed to run with an IBM WebSphere product, either for customer's
own internal use or for redistribution by customer, as part of such an application, in
customer's own products.

Licensed Materials - Property of IBM

5724-I63, 5724-H88, 5724-H89, 5655-N02, 5724-J08

Copyright IBM Corp. 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with
IBM Corp.
-->

<jnlp spec="1.0+" codebase="http://your_server:port_number/jws/wasappclient/apps/">
 <information>
  <title>Java EE Client Example</title>
  <vendor>IBM</vendor>
  <homepage href="null"/>
  <description>Java WebStart example: Launching Java EE Application Client</description>
  <description kind="short">Java EE Applicaiton Client</description>
  <description kind="tooltip">Java EE Application Client</description>
 </information>

 <security>
  <all-permissions/>
 </security>

 <resources>
  <j2se href="http://your_server:port_number/jws/wasappclient/JREDownload.xjnlp" version="1.6"/>
  <jar href="../lib/WebSphereClientLauncher.jar" download="eager" main="false"/>
  <jar href="../lib/properties.jar" download="eager" main="false"/>
  <jar href="SwingCalculator.ear" download="eager" main="false"/>

  <property name="com.ibm.websphere.client.launcher.ear" value="SwingCalculator.ear"/>
 </resources>

 <application-desc main-class="com.ibm.websphere.client.launcher.ClientLauncher">
  <argument>-CCproviderURL=corbaloc:iiop:tiu03.torolab.ibm.com:2809</argument>
 </application-desc>

</jnlp>
```

JNLP descriptor file for a Thin Application client application:

The deployment descriptor file is the main Java Network Launcher Protocol (JNLP) descriptor file for the client application. If it is a Thin Application client application, then the launcher class uses the current JVM from the Application Clients run-time dependency and invokes the Thin Application client application main method.

Here is an example of the JNLP descriptor file for a Thin Application client application.

```
This sample program is provided AS IS and may be used, executed, copied and modified
without royalty payment by customer (a) for its own instruction and study, (b) in order
to develop applications designed to run with an IBM WebSphere product, either for customer's
own internal use or for redistribution by customer, as part of such an application, in
customer's own products.

Licensed Materials - Property of IBM

5724-I63, 5724-H88, 5724-H89, 5655-N02, 5724-J08

Copyright IBM Corp. 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with
IBM Corp.

Licensed Materials - Property of IBM

5724-I63, 5724-H88, 5724-H89, 5655-N02, 5724-J08

Copyright IBM Corp. 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with
IBM Corp.
-->

<!-
============================================================================
->
<!-- TODO: change "codebase" to the actual URL location of the jnlp file ->
============================================================================
->
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+"
codebase="http://your_server:port_number/jws/wasappclient/apps">
  <information>
  <title>Thin Base Calculator Client Samples</title>
  <vendor>IBM</vendor>
  <description>Thin Base Calculator Client Samples</description>
  <offline-allowed/>
 </information>

 <security>
  <all-permissions/>
 </security>
  <resources>
   <j2se version="1.6" href="http://your_server:port_number/jws/wasappclient/JREDownload.xjnlp"/>

   <jar href="/jws/wasappclient/lib/WebSphereClientLauncher.jar" main="true"/>
   <jar href="BasicCalculatorClientCommon.jar"/>
   <jar href="BasicCalculatorEJB.jar"/>
   <jar href="BasicCalculatorThinClient.jar"/>

   <property name="com.ibm.websphere.client.launcher.main"
      value="com.ibm.websphere.samples.technologysamples.basiccalcthinclient.BasicCalculatorClientThinMain"/>
```

```
   <property name="java.naming.factory.initial"
              value="com.ibm.websphere.naming.WsnInitialContextFactory" />
   <property name="java.naming.provider.url"
               value="corbaloc:iiop:tiu03:2809"/>

   </resources>

   <add</argument>
    <argument>1</argument>
    <argument>2</argument>
   </application-desc>
</jnlp>
```

ClientLauncher class:

The class, com.ibm.websphere.client.installer.ClientLauncher, contains a main() method that is called by Java Web Start (JWS) to launch the client application. The Java Web Start client is used with platforms that support a Web browser.

This client is packaged in the `WebSphereClientLauncher.jar` file that is located in the Application Client for WebSphere Application Server installation under the <*app_client_root*>/ lib/webstart directory.

The launcher class requires that the following properties are defined. These properties are not defined in a separate properties file. Instead, the properties are defined as part of the Java Network Launching Protocol (JNLP) files.

com.ibm.websphere.client.launcher.main
: If the client application is a Thin Application client, then this property should be specified. It specifies the class where the main entry point of the client application resides.

com.ibm.websphere.client.launcher.ear
: If the client application is a Java Platform, Enterprise Edition (Java EE) Application client, then this property should be specified. It specifies the name of the EAR file to be executed. This property takes precedence over com.ibm.websphere.client.launcher.main. However, only one of the two properties should be specified.

Application client launcher for Java Web Start:

The application client launcher for Java Web Start is a Java class, com.ibm.websphere.client.installer.ClientLauncher, which has a main() method that Java Web Start calls to start the application client container and to invoke the application client's main() method. It provides similar functions as the **lauchClient** command line tool to start application clients from the command line.

The com.ibm.websphere.client.launcher.ClientLauncher class is packaged in the WebSphereClientLauncher.jar file under the <app_client_root>/lib/webstart directory.

The launcher tool requires that the following properties are defined.

com.ibm.websphere.client.launcher.main
: If the client that is to be run is a thin client, then this property should be specified. It specifies the class where the main entry point of the application resides. It is the main class name for a Thin application client. If it is set, the launcher will not start the client container, it will rather invoke the main method for the application directly. However, if `com.ibm.websphere.client.launcher.ear` is also set, it will be ignored.

com.ibm.websphere.client.launcher.ear
: If the client that is to run is the Java Platform, Enterprise Edition (Java EE) client, then this property should be specified. It specifies the name of the ear file to be executed. This property takes precedence over `com.ibm.websphere.client.launcher.main` although only one of the two properties should be specified.

These properties are not defined in a separate properties file. Instead, they are defined as part of the Java Network Launching Protocol files.

When `com.ibm.websphere.client.launcher.ear` is set, the application client launcher for JWS supports almost all of the –CC arguments as the **lauchClient** command line tool supports. However, if only `com.ibm.websphere.client.launcher.main` is set, the launcher will only support the –CCD argument. The following table shows the comparison of the supported –CC arguments for the **launchClient** command line tool and the application client launcher for JWS:

-CC argument	launchClient	Application client launcher for JWS
-CCverbose	Yes	Yes
-CCjar	Yes	Yes
-CCclasspath	Yes	N/A
-CCadminConnectorHost	Yes	Yes
-CCadminConnectorPort	Yes	Yes
-CCadminConnectorType	Yes	Yes
-CCadminConnectorUser	Yes	Yes
-CCaltDD	Yes	Yes
-CCBootstrapHost	Yes	Yes
-CCBootstrapPort	Yes	Yes
-CCproviderURL	Yes	Yes
-CCinitonly	Yes	N/A
-CCtrace	Yes	Yes
-CCtracefile	Yes	Yes
-CCsecurityManager	Yes	N/A
-CCsecurityMgrClass	Yes	N/A
-CCsecurityMgrPolicy	Yes	N/A
-CCD	Yes	Yes
-CCexitVM	Yes	Yes
-CCdumpJavaNameSpace	Yes	Yes
-CCsoapConnectorPort	Yes	Yes
-CCtraceMode	Yes	Yes
-CCclassLoaderMode	Yes	Yes

Macro expansion is supported for the –CCD argument by the application client launcher for JWS. The launcher will automatically substitute certain macro keys (enclosed with ${...}) with the calculated value at runtime. For example, if a macro key is used in the –CCD argument in the application client JNLP manifest file,

```
<argument>-CCDcom.ibm.ssl.keyStore= ${WAS_ROOT}/etc/key.p12</argument>
```

it will be expanded to the JWS cache installation root location and the argument will become:

```
-CCDcom.ibm.ssl.keyStore=/home/tiu/.java/deployment/cache/javaws/ext/E1134532441112/etc/key12.p12
```

The following table shows the 3 macro keys that are currently supported and will be substituted by the launcher:

Table 9.

Macro key	Value
${WAS_ROOT}	Installation root location within the JWS cache that is used by the application client container and runtime installer for JWS.
${JAVA_HOME}	Location of Java home. The return value of System.getProperty("java.home").
${USER_HOME}	Location of user home. The return value of System.getProperty("user.home").

Preparing the application client run time dependency component for Java Web Start

To launch a Java Platform, Enterprise Edition (Java EE) application client application, a Thin application client application, or both using Java Web Start (JWS), a Java Runtime Environment implementation Java archive (JAR) that IBM provides, the library JAR files and properties files bundled in Application Client for WebSphere Application Server must be installed in the JWS. Learn the steps to build the application client run time dependency component from an application client installation. It is packaged as a Web Archive Resource (WAR) file that can be installed in an Application Server.

Before you begin

Install the Application Client for WebSphere Application Server for the operating system to which the client application deploys. If there is a requirement to deploy the client application to multiple operating systems, the application client run time dependency component must be built separately for each operating system that client application supports.

1. Install the Application Client for WebSphere Application Server for the client application supported operating systems.
2. Change the directory to the installation bin directory.
3. Run the "buildClientRuntime tool" on page 390 to generate the application client run time JAR file, which contains the Java Standard Edition Runtime Environment, the run time library JAR files, properties files, and the SSL KeyStore and TrustStore files from the application client installation.
4. Run the buildClientLibJars tools to package up the properties files in the properties directory of the application client installation into a properties.jar file in the specified location. The buildClientLibJars tools will also copy the `WebSphereClientLauncher.jar` file and `WebSphereClientRuntimeInstaller.jar` file from the application client installation to the specified location. All jar files in the specified location will be signed by the provided certificate.

   For example, if you are using Version 7.0 and using the test certificate that is included in the application client installation:

   ```
   buildClientLibJars C:\Temp\webstart ..\etc\DummyClientKeyFilejar WebAS "websphere dummy client" JKS
   ```
5. Create a JavaServer Pages (JSP) file or use a servlet to generate the application client run time installer Java Network Launching Protocol (JNLP) descriptor to respond to Java Web Start request. See the Java Web Start Deployment sample in the application client installation.
6. Package the two signed JAR files, `WASClient7.0_windows.jar` and `WebSphereClientRuntimeInstaller.jar`, and the JSP file or servlet for generating the Application Client run time installer JNLP descriptor into a Web archive (WAR) file. This WAR file is packaged into an EAR file that can be deployed to an application server. See the Java Web Start Deployment sample in the application client installation.

Results

Your Web application is ready to serve the application client run time and the JRE environment.

buildClientRuntime tool:

For a Java Platform, Enterprise Edition (Java EE) application client application and or Thin application client application to be launched using Java Web Start (JWS), the library JAR files bundled in Application Client for WebSphere Application Server must be installed in the Java Web Start. Use this tool to build those JAR files. The Java Web Start client is used with platforms that support a Web browser.

The buildClientRuntime tool builds the required components from the WebSphere Application Server clients installation into the JAR file specified on the command. This JAR file contains:

- License files
- Java SE Runtime Environment 6 (JRE 6) that IBM provides
- Application Clients runtime properties and configuration
- SSL KeyStore and TrustStore files
- Runtime library JAR files

In the case of building an Application Clients runtime JAR file only for serving Thin Application client applications and not for Java EE Application client applications, the runtime library JAR files and the Application Clients runtime properties files are not included, except the configuration files, `sas.client.props`, `ssl.client.props` and `soap.client.props`, located in the `WAS_ROOT/properties` directory. The Java Web Start client is used with platforms that support a Web browser.

The command-line invocation syntax for the buildClientRuntime tool is shown in the following example:

```
Windows Usage:    buildClientRuntime.bat  [-help] [-verbose] outfile keystore storepass alias storetype

Unix Usage:         buildClientRuntime.sh    [-help] [-verbose] outfile keystore storepass alias storetype
```

where:

- -help will display the message
- -verbose will turn on verbose message
- outfile is the output file name
- keystore is the key store file
- storepass is the key store password
- alias is the key alias name
- storetype is the key store type

ClientRuntimeInstaller class:

This section provides information on the ClientRuntimeInstaller class.

This class, com.ibm.websphere.client.installer.ClientRuntimeInstaller, contains a main() method that Java Web Start (JWS) calls to install the Application Client for WebSphere Application Server run-time dependency component in JWS cache. It is packaged in `WebSphereClientRuntimeInstaller.jar` file located in the Application Client for WebSphere Application Server installation in the *<app_server_root>*/`JWS` directory.

Specify the `WebSphereClientRuntimeInstaller.jar` file and the Application Client run-time dependency component JAR file as JAR resources in the Application Client run-time installer Java Network Launcher Protocol (JNLP) descriptor file. See the following example for details:

```
<jar href="Launcher/WebSphereClientRuntimeInstall.jar" main="true"/>
<jar href="Launcher/WASClient6.1_windows.jarRuntimeInstall.jar" main="true"/>
```

The ClientRuntimeInstaller class main method requires the following properties to be set in the JNLP file:

com.ibm.websphere.client.jre.version
Specifies a Java Runtime Environment (JRE) version name that is to be used when referring to the Application Client run-time dependency component.

com.ibm.websphere.client.jre.launch.java
Specifies the relative location of the javaw.exe program in the Application Client run-time dependency component JAR file.

The previously mentioned properties, JRE version name and the location of the javaw.exe program are registered to the Java Web Start Application Manager, as shown in the following example:

```
<property name="com.ibm.websphere.client.jre.version" value="WASclient6.1"/>
<property name="com.ibm.websphere.client.jre.launch.java" value="java\jre\bin\javaw.exe"/>
```

Using the Java Web Start sample

The EAR file, `WebSphereClientRuntime.ear`, is provided in the JWS directory of the Client Application for WebSphere Application Server installation. This EAR file provides a sample Application Clients run-time installer JNLP descriptor file and a sample Application Clients run-time library component JNLP descriptor file. Follow the steps in this task to build the Application Clients run-time dependency component and the Application Clients run-time library component. Add these components to the `WebSphereClientRuntime.ear` file, and then install the EAR file in an Application Server to be used by the client application.

About this task

There is a new Java Web Start sample available in the client sample gallery for WebSphere Application Server V7.0. Refer to the client sample gallery in the Application Client for WebSphere Application Server product. The name of the new sample is ″Java Web Start Deployment Sample″.

Installing Java Web Start

Learn about the steps that are necessary to install Java Web Start (JWS) on the AIX platform. Java Web Start technology is provided by the Java SE runtime environment to deploy Java EE application clients (including Thin application clients) on the remote client machine with a single click from Web browser on the client machine.

Before you begin

Note: This topic applies only to the AIX operating system.

Before you begin this task, see the “Preparing the application client run time dependency component for Java Web Start” on page 389 topic to understand Java Web Start (JWS) technology and components.

Note:

You can use the following resources:

- The IBM implementation of Java Web Start on Java Platform Standard Edition 6 (Java SE 6) that is packaged in the Application Client for WebSphere Application Server

Note: You can use Java Web Start on Java Platform, Standard Edition Developer Kit 6 that IBM provides, packaged in the Application Client for WebSphere Application Server, Version 7.0; Java Web Start on Sun Microsystems Java Standard Edition Software Development Kit 6 or Java SE Runtime Environment 6.0, which you can download from the Sun Microsystems Web site for Windows, Linux and Solaris operating systems, or the Java Web Start on HP Software Development Kit (SDK) or RTE for Java 2 version 5.0, which you can download from the HP Web site.

About this task

Use the following steps to install JWS on the AIX platform.

1. Install IBM Application Client for WebSphere Application Server.
2. Change your directory to the client_install_root/java/jre/lib/javaws path.
3. Run the updateSetting.sh script from the path mentioned in the previous step.
4. Change your path to the JWS installed path. For example, enter: `client_install_root/java/jre/javaws`.
5. Run `./javaws` from the path mentioned in the previous step.

Using a static JNLP file with Java Web Start for Application clients

Do not use JSP to dynamically generate a JNLP file, otherwise the JNLP jsp page cannot be opened in some IE browsers.

About this task

To use a static JNLP file, you will need to add the following mime type mapping in the web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
       <display-name>
       WAS Client runtime for Java Web Start</display-name>
       <welcome-file-list>
              <welcome-file>index.html</welcome-file>
              <welcome-file>index.htm</welcome-file>
              <welcome-file>index.jsp</welcome-file>
              <welcome-file>default.html</welcome-file>
              <welcome-file>default.htm</welcome-file>
              <welcome-file>default.jsp</welcome-file>
       </welcome-file-list>
       <mime-mapping>
       <extension>jnlp</extension>
       <mime-type>application/x-java-jnlp-file</mime-type>
       </mime-mapping>
</web-app>
```

Writing command interfaces

The command package can be used by distributed applications to reduce the number of remote invocations that a client makes. The base interface for all commands is the Command interface.

About this task

Distributed applications are defined by the ability to utilize remote resources as if they were local, but this remote work affects the performance of distributed applications. Distributed applications can improve performance by using remote calls sparingly. For example, if a server does several tasks for a client, the application can run more quickly if the client bundles requests together, reducing the number of individual remote calls. The command package provides a mechanism for collecting sets of requests to be submitted as a unit.

In addition, the command package provides a generic way of making requests. A client instantiates the command, sets its input data, and tells it to run. The command infrastructure determines the target server and passes a copy of the command to it. The server runs the command, sets any output data, and copies it back to the client. The package provides a common way to issue a command, locally or remotely, and independently of the server's implementation. Any server (an enterprise bean, a Java Database

Connectivity (JDBC) server, a servlet, and so on) can be a target of a command if the server supports Java access to its resources and provides a way to copy the command between the client's Java Virtual Machine (JVM) and its own JVM.

The command facility is implemented in the com.ibm.websphere.command Java package. The classes and interfaces in the command package fall into four general categories:

- Interfaces for creating commands
- Classes and interfaces for implementing commands
- Classes and interfaces for determining where the command is run
- Classes defining package-specific exceptions
- Write a command interface. Extend one or more of the three interfaces that are included in the command package. The command interface provides only the client-side interface for generic commands and declares three basic methods:
  - isReadyToCallExecute - This method is called on the client side before the command is passed to the server for execution.
  - execute - This method passes the command to the target and returns any data.
  - reset - This method reverts any output properties to the values they had before the execute method was called so that the object can be reused.
- Implement the command interface. The implementation class for your interface must contain implementations for the isReadyToCallExecute and reset methods. The execute method is implemented for you elsewhere. Most commands do not extend the Command interface directly but use one of the provided extensions: the "TargetableCommand interface" and the "CompensableCommand interface" on page 394.
- Use the command.

TargetableCommand interface

The TargetableCommand interface extends the Command interface and provides for remote execution of commands.

Most commands will be targetable commands. The TargetableCommand interface declares several additional methods:

- setCommandTarget

  This method allows you to specify the target object to a command.
- setCommandTargetName

  This method allows you to specify the target by name to a command
- getCommandTarget

  This method returns the target object of the command.
- getCommandTargetName

  This method returns the name of the target object of the command.
- hasOutputProperties

  This method indicates whether the command has output that must be copied back to the client. (The implementation class also provides a method, setHasOutputProperties, for setting the output of this method. By default, hasOutputProperties returns true.)
- setOutputProperties

  This method saves output values from the command for return to the client.
- performExecute

  This method encapsulates the application-specific work. It is called for you by the execute method declared in the Command interface.

With the exception of the performExecute method, which you must implement, all of these methods are implemented in the TargetableCommandImpl class. This class also implements the execute method declared in the Command interface.

Command interface example application

ModifyCheckingAccountCmd command interface

This example uses an entity bean with container-managed persistence (CMP), called CheckingAccountBean, which enables a client to deposit money, withdraw money, set a balance, get a balance, and retrieve the name on the account. This entity bean also accepts commands from the client. The code examples illustrate the command-related programming. For a servlet-based example, see “Example: Writing a command target (client-side adapter)” on page 409

This command is both targetable and compensable, so the interface extends both TargetableCommand and CompensableCommand interfaces.

```
...
import com.ibm.websphere.exception.*;
import com.ibm.websphere.command.*;
public interface ModifyCheckingAccountCmd
extends TargetableCommand, CompensableCommand {
float getAmount();
float getBalance();
float getOldBalance(); // Used for compensating
float setBalance(float amount);
float setBalance(int amount);
CheckingAccount getCheckingAccount();
void setCheckingAccount(CheckingAccount newCheckingAccount);
TargetPolicy getCmdTargetPolicy();
...
}
```

CompensableCommand interface

The CompensableCommand interface extends the Command interface.

A compensable command is one that has another command (a compensator) associated with it, so that the work of the first can be undone by the compensator. For example, a command that attempts to make an airline reservation followed by a hotel reservation can offer a compensating command that allows the user to cancel the airline reservation if the hotel reservation cannot be made.

The CompensableCommand interface declares one method:

- getCompensatingCommand

  This method returns the command that can be used to undo the effects of the original command.

To create a compensable command, you write an interface that extends the CompensableCommand interface. Such interfaces typically extend the TargetableCommand interface as well. You must implement the getCompensatingCommand method in the implementation class for your interface. You must also implement the compensating command.

Implementing command interfaces

The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface.

About this task

- To implement an application command interface, write a class that extends the TargetableCommandImpl class and implements your command interface. The structure of the ModifyCheckingAccountCmdImpl class is as follows:

```
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
// Variables
...
// Methods
...
}
```

- The class must declare any variables and implement the following methods:
  - Any methods you defined in your command interface.
  - The isReadyToCallExecute and reset methods from the Command interface.
  - The performExecute method from the TargetableCommand interface.
  - The getCompensatingCommand method from the CompensableCommand interface, if your command is compensable. You must also implement the compensating command.

  You can also override the nonfinal implementations provided in the TargetableCommandImpl class. The most likely candidate for reimplementation is the setOutputProperties method, since the default implementation does not save final, transient, or static fields.

Instance and class variables

The ModifyCheckingAccountCmdImpl class declares the variables used by the methods in the class, including the remote interface of the CheckingAccount entity bean, the variables used to capture operations on the checking account (balances and amounts), and a compensating command.

Variables that are used by the ModifyCheckingAccountCmd command

The following code example shows the variables in the ModifyCheckingAccountCmdImpl class:

```
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
// Variables
public float balance;
public float amount;
public float oldBalance;
public CheckingAccount checkingAccount;
public ModifyCheckingAccountCompensatorCmd
modifyCheckingAccountCompensatorCmd;
...
}
```

Related tasks

“Implementing command interfaces” on page 394
The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface.

Command-specific methods

The ModifyCheckingAccountCmd interface defines several command-specific methods in addition to extending other interfaces in the command package. These command-specific methods are implemented in the ModifyCheckingAccountCmdImpl class.

Code example: Constructors in the ModifyCheckingAccountCmdImpl class

You must provide a way to instantiate the command. The command package does not specify the mechanism, so you can choose the technique most appropriate for your application. The fastest and most efficient technique is to use constructors. The most flexible technique is to use a factory. Also, since commands are implemented internally as JavaBeans components, you can use the standard Beans.instantiate method. The ModifyCheckingAccountCmd command uses constructors.

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
// Variables
...
// Constructors
// First constructor: relies on the default target policy
public ModifyCheckingAccountCmdImpl(CommandTarget target,
float newAmount)
{
amount = newAmount;
checkingAccount = (CheckingAccount)target;
setCommandTarget(target);
}
// Second constructor: allows you to specify a custom target policy
public ModifyCheckingAccountCmdImpl(CommandTarget target,
float newAmount,
TargetPolicy targetPolicy)
{
setTargetPolicy(targetPolicy);
amount = newAmount;
checkingAccount = (CheckingAccount)target;
setCommandTarget(target);
}
...
}
```

This code example shows the two constructors for the command. The difference between them is that the first uses the default target policy for determining the target of the command and the second allows you to specify a custom policy. For more information on targets and target policies, see “Targets and target policies”.

Both constructors take a CommandTarget object as an argument and cast it to the CheckingAccount type. The CheckingAccount interface extends both the CommandTarget interface and the EJBObject (see Figure 80 on page 160). The resulting checkingAccount object routes the command to the desired server by using the bean’s remote interface. (For more information on CommandTarget objects, see “Writing a command target (server)” on page 159.)

Code example: Command-specific methods in the ModifyCheckingAccountCmdImpl class

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
// Variables
...
// Constructors
...
// Methods in ModifyCheckingAccountCmd interface
public float getAmount() {
return amount;
}
public float getBalance() {
return balance;
}
```

```
public float getOldBalance() {
return oldBalance;
}
public float setBalance(float amount) {
balance = balance + amount;
return balance;
}
public float setBalance(int amount) {
balance += amount ;
return balance;
}
public TargetPolicy getCmdTargetPolicy() {
return getTargetPolicy();
}
public void setCheckingAccount(CheckingAccount newCheckingAccount) {
if (checkingAccount == null) {
checkingAccount = newCheckingAccount;
}
else
System.out.println("Incorrect Checking Account (" +
newCheckingAccount + ") specified");
}
public CheckingAccount getCheckingAccount() {
return checkingAccount;
}
...
}
```

This code example shows the implementation of the following command-specific methods:

- setBalance - sets the balance of the account.
- getAmount - returns the amount of a deposit or withdrawal.
- getOldBalance, getBalance - capture the balance before and after an operation.
- getCmdTargetPolicy - retrieves the current target policy.
- setCheckingAccount, getCheckingAccount - set and retrieve the current checking account.

The ModifyCheckingAccountCmd command operates on a checking account. Because commands are implemented as JavaBeans components, you manage input and output properties of commands using the standard JavaBeans techniques. For example, initialize input properties with set methods (like setCheckingAccount) and retrieve output properties with get methods (like getCheckingAccount). The get methods do not work until after the execute method for the command has been called.

Related tasks

“Implementing command interfaces” on page 394
The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface.

Example: Implementing methods from the TargetableCommand interface

The TargetableCommand interface declares one method, performExecute, that application programmer must implement.

Methods from the TargetableCommand interface in the ModifyCheckingAccountCmdImpl class

The following code example shows the implementations for the **ModifyCheckingAccountCmd** command. The implementation of the **performExecute** method is as follows:

- Saves the current balance (so the command can be undone by a compensator command)
- Calculates the new balance
- Sets the current balance to the new balance

- Ensures that the hasOutputProperties method returns true so that the values are returned to the client

In addition, the ModifyCheckingAccountCmdImpl class overrides the default implementation of the setOutputProperties method.

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
...
// Method from the TargetableCommand interface
public void performExecute() throws Exception {
CheckingAccount checkingAccount = getCheckingAccount();
oldBalance = checkingAccount.getBalance();
balance = oldBalance+amount;
checkingAccount.setBalance(balance);
setHasOutputProperties(true);
}
public void setOutputProperties(TargetableCommand fromCommand) {
try {
if (fromCommand != null) {
ModifyCheckingAccountCmd modifyCheckingAccountCmd =
(ModifyCheckingAccountCmd) fromCommand;
this.oldBalance = modifyCheckingAccountCmd.getOldBalance();
this.balance = modifyCheckingAccountCmd.getBalance();
this.checkingAccount =
modifyCheckingAccountCmd.getCheckingAccount();
this.amount = modifyCheckingAccountCmd.getAmount();
}
}
catch (Exception ex) {
System.out.println("Error in setOutputProperties.");
}
}
...
}
```

Related tasks

"Implementing command interfaces" on page 394
The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface.

Setting and determining targets

The object that is the target of a TargetableCommand must implement the CommandTarget interface. This object can be an actual server-side object, such as an entity bean, or it can be a client-side adapter for a server.

About this task

The implementor of the CommandTarget interface is responsible for ensuring the proper execution of a command in the desired target server environment. This typically requires the following steps:

1. Copying the command to the target server by using a server-specific protocol.
2. Running the command in the server.
3. Copying the executed command from the target server to the client by using a server-specific protocol.

Example: Implementing methods from the Command interface

The Command interface declares two methods, isReadyToCallExecute and reset, that the application programmer must implement.

Methods from the Command interface in the ModifyCheckingAccountCmdImpl class

The following code example shows the implementations for the **ModifyCheckingAccountCmd** command. The implementation of the **isReadyToCallExecute** method ensures that the checkingAccount variable is set. The **reset** method sets all of the variables back to starting values.

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
...
// Methods from the Command interface
public boolean isReadyToCallExecute() {
if (checkingAccount != null)
return true;
else
return false;
}
public void reset() {
amount = 0;
balance = 0;
oldBalance = 0;
checkingAccount = null;
targetPolicy = new TargetPolicyDefault();
}
...
}
```

Related tasks

"Implementing command interfaces" on page 394
The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface.

Example: Implementing methods from the Compensable interface

The CompensableCommand interface declares the **getCompensatingCommand** method that the application programmer must implement.

Method from the CompensableCommand interface in the ModifyCheckingAccountCmdImpl class

The following code example shows the implementation for the **ModifyCheckingAccountCmd** command. The implementation simply returns an instance of the **ModifyCheckingAccountCompensatorCmd** command that is associated with the current command.

```
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
...
// Method from CompensableCommand interface
public Command getCompensatingCommand() throws CommandException {
modifyCheckingAccountCompensatorCmd =
new ModifyCheckingAccountCompensatorCmd(this);
return (Command)modifyCheckingAccountCompensatorCmd;
}
}
```

Writing the compensating command

An application that uses a compensable command requires two separate commands: the primary command (declared as a CompensableCommand) and the compensating command. In the example application, the primary command is declared in the ModifyCheckingAccountCmd interface and implemented in the ModifyCheckingAccountCmdImpl class. Because this command is also a compensable

command, there is a second command associated with it that is designed to undo its work. When you create a compensable command, you also have to write the compensating command.

Writing a compensating command can require exactly the same steps as writing the original command: writing the interface and providing an implementation class. In some cases, it may be simpler. For example, the command to compensate for the ModifyCheckingAccountCmd does not require any methods beyond those defined for the original command, so it does not need an interface. The compensating command, called ModifyCheckingAccountCompensatorCmd, simply needs to be implemented in a class that extends the TargetableCommandImpl class. This class must:

- Provide a way to instantiate the command; the example uses a constructor.
- Implement the three required methods: isReadyToCallExecute and reset—both from the Command interface and performExecute—from the TargetableCommand interface.

The following code example shows the structure of the implementation class, its variables (references to the original command and to the relevant checking account), and the constructor. The constructor simply instantiates the references to the primary command and account.

```
...
public class ModifyCheckingAccountCompensatorCmd extends TargetableCommandImpl
{
public ModifyCheckingAccountCmdImpl modifyCheckingAccountCmdImpl;
public CheckingAccount checkingAccount;
public ModifyCheckingAccountCompensatorCmd(
ModifyCheckingAccountCmdImpl originalCmd)
{
// Get an instance of the original command
modifyCheckingAccountCmdImpl = originalCmd;
// Get the relevant account
checkingAccount = originalCmd.getCheckingAccount();
}
// Methods from the Command and Targetable Command interfaces
....
}
```

The performExecute method verifies that the actual checking-account balance is consistent with what the original command returns. If so, it replaces the current balance with the previously stored balance by using the ModifyCheckingAccountCmd command. Finally, it saves the most-recent balances in case the compensating command needs to be undone. The reset method has no work to do.

The following code example shows the implementation of the inherited methods. The implementation of the isReadyToCallExecute method ensures that the checkingAccount variable has been instantiated.

```
...
public class ModifyCheckingAccountCompensatorCmd extends TargetableCommandImpl
{
// Variables and constructor
....
// Methods from the Command and TargetableCommand interfaces
public boolean isReadyToCallExecute() {
if (checkingAccount != null)
return true;
else
return false;
}
public void performExecute() throws CommandException
{
try {
ModifyCheckingAccountCmdImpl originalCmd =
modifyCheckingAccountCmdImpl;
// Retrieve the checking account modified by the original command
CheckingAccount checkingAccount = originalCmd.getCheckingAccount();
if (modifyCheckingAccountCmdImpl.balance ==
checkingAccount.getBalance()) {
```

```
// Reset the values on the original command
checkingAccount.setBalance(originalCmd.oldBalance);
float temp = modifyCheckingAccountCmdImpl.balance;
originalCmd.balance = originalCmd.oldBalance;
originalCmd.oldBalance = temp;
}
else {
// Balances are inconsistent, so we cannot compensate
throw new CommandException(
"Object modified since this command ran.");
}
}
catch (Exception e) {
System.out.println(e.getMessage());
}
}
public void reset() {}
}
```

Related tasks

"Implementing command interfaces" on page 394
The command package provides a class, TargetableCommandImpl, that implements all of the methods in the TargetableCommand interface except the performExecute method. It also implements the execute method from the Command interface.

Interfaces for creating commands

The Command interface specifies the most basic aspects of a command. This interface is extended by both the TargetableCommand interface and the CompensableCommand interface, which offer additional features.

To create commands for applications, you must:

- Define an interface that extends one or more of interfaces in the command package.
- Provide an implementation class for your interface.

In practice, most commands implement the TargetableCommand interface, which allows the command to be executed remotely. The following example shows the structure of a command interface for a targetable command:

```
... import com.ibm.websphere.command.*;
public interface MySimpleCommand extends TargetableCommand { // Declare application
methods here }
```

The CompensableCommand interface enables the association of one command with another that can undo the work of the first. Compensable commands also typically implement the TargetableCommand interface. The following example shows the structure of a command interface for a targetable, compensable command:

```
... import com.ibm.websphere.command.*;
public interface MyCommand extends TargetableCommand, CompensableCommand {
// Declare application methods here }
```

Facilities for implementing commands

Commands are implemented by extending the class TargetableCommandImpl, which implements the TargetableCommand interface. The TargetableCommandImpl class is an abstract class that provides some implementations for some of the methods in the TargetableCommand interface (for example, setting return values) and declares additional methods that the application itself must implement (for example, how to execute the command).

You implement your command interface by writing a class that extends the TargetableCommandImpl class and implements your command interface. This class contains the code for the methods in your interface, the methods inherited from extended interfaces (the TargetableCommand and CompensableCommand

interfaces), and the required (abstract) methods in the TargetableCommandImpl class. You can also override the default implementations of other methods provided in the TargetableCommandImpl class. The following example shows the structure of an implementation class for an interface:

```
... import java.lang.reflect.*; import com.ibm.websphere.command.*;
public class MyCommandImpl extends TargetableCommandImpl implements MyCommand
{ // Set instance variables here ... // Implement methods in the MyCommand
interface ... // Implement methods in the CompensableCommand interface ...
// Implement abstract methods in the TargetableCommandImpl class ... }
```

Exceptions in the command package

The command package defines a set of exception classes.

The CommandException class extends the DistributedException class and acts as the base class for the additional command-related exceptions:

- UnauthorizedAccessException
- UnsetInputPropertiesException
- UnavailableCompensableCommandException

Applications can extend the CommandException class to define additional exceptions.

Although the CommandException class extends the DistributedException class, you do not have to import the distributed-exception package, com.ibm.websphere.exception, unless you need to use the features of the DistributedException class in your application.

Targets and target policies

A targetable command extends the TargetableCommand interface, which allows the client to direct a command to a particular server. The TargetableCommand interface (and the TargetableCommandImpl class) provide two ways for a client to specify a target: the setCommandTarget and setCommandTargetName methods.

The setCommandTarget methods allows the client to set the target object directly on the command. The setCommandTargetName method allows the client to refer to the server by name; this approach is useful when the client is not directly aware of server objects. A targetable command also has corresponding getCommandTarget and getCommandTargetName methods.

The command package needs to be able to identify the target of a command. Because there is more than one way to specify the target and because different applications can have different requirements, the command package does not specify a selection algorithm. Instead, it provides a TargetPolicy interface with one method, getCommandTarget, and a default implementation. This enables applications to devise custom algorithms for determining the target of a command when appropriate.

The default target policy

The command package provides a default implementation of the TargetPolicy interface in the TargetPolicyDefault class.

Relevant variables and the methods in the TargetPolicyDefault class

If you use this default implementation, the command determines the target by looking through an ordered sequence of four options:

- The CommandTarget value
- The CommandTargetName value
- A registered mapping of a target for a specific command
- A defined default target

If the command finds no target, it returns null.

The TargetPolicyDefault class provides methods for managing the assignment of commands with targets (registerCommand, unregisterCommand, and listMappings), and a method for setting a default name for the target (setDefaultTargetName). The default target name is com.ibm.websphere.command.LocalTarget, where LocalTarget is a class that runs the command's performExecute method locally.

```
...
public class TargetPolicyDefault implements TargetPolicy, Serializable
{
...
protected String defaultTargetName = "com.ibm.websphere.command.LocalTarget";
public CommandTarget getCommandTarget(TargetableCommand command) {
... }
public Dictionary listMappings() {
... }
public void registerCommand(String commandName, String targetName) {
... }
public void unregisterCommand(String commandName) {
... }
public void seDefaultTargetName(String defaultTargetName) {
... }
}
```

Setting the command target:

The ModifyCheckingAccountImpl class provides two command constructors. One of them takes a command target as an argument and implicitly uses the default target policy to locate the target. The constructor passes a null target, so that the default target policy traverses its choices and eventually finds the default target name, LocalTarget.

Identifying a target with CommandTarget

If you use this default implementation, the command determines the target by looking through an ordered sequence of four options:

- The CommandTarget value
- The CommandTargetName value
- A registered mapping of a target for a specific command
- A defined default target

If the command finds no target, it returns null.

This example uses the same constructor to set the target explicitly. This example differs from the example in "Using a command" on page 408 as follows:

- The command target is set to the checking account rather than null. The default target policy starts to traverse its choices and finds the target in the first place it looks.
- It does not have to call the setCheckingAccount method to indicate the account on which the command should operate; the constructor uses the target variable as both the target and the account.

```
{
...
CheckingAccount checkingAccount
....
try {
ModifyCheckingAccountCmd cmd =
new ModifyCheckingAccountCmdImpl(checkingAccount, 1000);
cmd.execute();
}
catch (Exception e) {
System.out.println(e.getMessage());
}
...
}
```

Setting the command target name:

If a client needs to set the target of the command by name, it can use the setCommandTargetName method for the command.

Identifying a target with CommandTargetName

This example compares with the example in “Using a command” on page 408 as follows:

- Both explicitly set the command target in the constructor to null.
- Both use the setCheckingAccount method to indicate the account on which the command should operate.
- This example sets the target name explicitly by using the setCommandTargetName method. When the default target policy traverses its choices, it finds a null for the first choice and a name for the second.

```
{
...
CheckingAccount checkingAccount
....
try {
ModifyCheckingAccountCmd cmd =
new ModifyCheckingAccountCmdImpl(null, 1000);
cmd.setCheckingAccount(checkingAccount);
cmd.setCommandTargetName("com.ibm.sfc.cmd.test.CheckingAccountBean");
cmd.execute();
}
catch (Exception e) {
System.out.println(e.getMessage());
}
...
}
```

Example: Mapping the command to a target name:

The default target policy also permits commands to be registered with targets. Mapping a command to a target is an administrative task that most appropriately done through a configuration tool.

Mapping a command to a target in an external application

The WebSphere Application Server administrative console does not yet support the configuration of mappings between commands and targets. Applications that require support for the registration of commands with targets must supply the tools to manage the mappings. These tools can be visual interfaces or command-line tools.

The following example shows the registration of a command with a target. The names of the command class and the target are explicit in the code, but in practice, these values would come from fields in a user interface or arguments to a command-line tool. If a program creates a command as shown in the example in “Using a command” on page 408, with a null for the target, when the default target policy traverses its choices, it finds a null for the first and second choices and a mapping for the third.

```
{
...
targetPolicy.registerCommand(
"com.ibm.sfc.cmd.test.ModifyCheckingAccountImpl",
"com.ibm.sfc.cmd.test.CheckingAccountBean");
...
}
```

Example: Customizing target policies

You can define custom target policies by implementing the TargetPolicy interface and providing a getCommandTarget method appropriate for your application. The TargetableCommandImpl class provides setTargetPolicy and getTargetPolicy methods for managing custom target policies.

Custom target policy

So far, the target of all the commands has been a checking-account entity bean. Suppose that someone introduces a session enterprise bean (MySessionBean) that can also act as a command target. The following code example shows a simple custom policy that sets the target of every command to MySessionBean.

```
import java.io.*;
import java.util.*;
import java.beans.*;
import com.ibm.websphere.command.*;
public class CustomTargetPolicy implements TargetPolicy, Serializable {
public CustomTargetPolicy {
super();
}
public CommandTarget getCommandTarget(TargetableCommand command) {
CommandTarget = null;
try {
target = (CommandTarget)Beans.instantiate(null,
"com.ibm.sfc.cmd.test.MySessionBean");
}
catch (Exception e) {
e.printStackTrace();
}
}
}
```

Because commands are implemented as JavaBeans components, using custom target policies requires importing the java.beans package and writing some elementary JavaBeans code. Also, your custom target-policy class must also implement the java.io.Serializable interface.

Example: Using a custom target policy:

The ModifyCheckingAccountImpl class provides two command constructors. One of them implicitly uses the default target policy; the other takes a target policy object as an argument, which enables you to use a custom target policy.

Custom target policy example

The following example uses the second constructor, passing a null target and a custom target policy, so that the custom policy is used to determine the target. After the command is executed, the code uses the reset method to return the target policy to the default.

```
{
...
CheckingAccount checkingAccount
....
try {
CustomTargetPolicy customPolicy = new CustomTargetPolicy();
ModifyCheckingAccountCmd cmd =
new ModifyCheckingAccountCmdImpl(null, 1000, customPolicy);
cmd.setCheckingAccount(checkingAccount);
cmd.execute();
cmd.reset();
}
catch (Exception e) {
System.out.println(e.getMessage());
}
}
```

Writing a command target (server)

A server must implement the CommandTarget interface and its single method, executeCommand, to accept commands.

About this task

The example application implements the CommandTarget interface in an enterprise bean. The target enterprise bean can be a session bean or an entity bean. You can write a target enterprise bean that forwards commands to a specific server, such as another entity bean. In this case, all commands directed at a specific target go through the target enterprise bean. You can also write a target enterprise bean that does the work of the command locally.

Make an enterprise bean the target of a command, as follows:

- Extending the CommandTarget interface when you define the remote interface for the bean, which must also extend the EJBObject interface.
- Implementing the CommandTarget interface when you implement the bean class, which must also implement either the SessionBean or EntityBean interface. The target of the example application is an enterprise bean called CheckingAccountBean. The remote interface for the bean, CheckingAccount, extends the CommandTarget interface in addition to the EJBObject interface. The methods that are declared in the remote interface are independent of those that are used by the command. The executeCommand is declared in neither the home for the bean, nor remote interfaces. The following code example displays the CheckingAccount interface.

```
...
import com.ibm.websphere.command.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface CheckingAccount extends CommandTarget, EJBObject {
float deposit (float amount) throws RemoteException;
float deposit (int amount) throws RemoteException;
String getAccountName() throws RemoteException;
float getBalance() throws RemoteException;
float setBalance(float amount) throws RemoteException;
float withdrawal (float amount) throws RemoteException, Exception;
float withdrawal (int amount) throws RemoteException, Exception;
}
```

  The enterprise bean class, CheckingAccountBean, implements the EntityBean interface as well as the CommandTarget interface. The class contains the business logic for the methods in the remote interface, the necessary life-cycle methods (ejbActivate, ejbStore, and so on), and the executeCommand declared by the CommandTarget interface. The executeCommand method is the only command-specific code in the enterprise bean class. It attempts to run the performExecute method on the command and throws a CommandException if an error occurs. If the performExecute method runs successfully, the executeCommand method uses the hasOutputProperties method to determine if there are output properties that must be returned. If the command has output properties, the method returns the command object to the client. The following code example displays the relevant parts of the CheckingAccountBean class.

```
...
public class CheckingAccountBean implements EntityBean, CommandTarget {
// Bean variables
...
// Business methods from remote interface
...
// Life-cycle methods for CMP entity beans
...
// Method from the CommandTarget interface
public TargetableCommand executeCommand(TargetableCommand command)
throws RemoteException, CommandException
{
try {
command.performExecute();
}
catch (Exception ex) {
if (ex instanceof RemoteException) {
RemoveException remoteException = (RemoteException)ex;
if (remoteException.detail != null) {
```

```
throw new CommandException(remoteException.detail);
}
throw new CommandException(ex);
}
}
if (command.hasOutputProperties()) {
return command;
}
return null;
}
}
```

Example: Compensating command example

To use a compensating command, you must retrieve the compensator that is associated with the primary command and call its execute method.

ModifyCheckingAccountCompensator command

The following code example shows the code used to run the original command and to give the user the option of undoing the work by running the compensating command.

```
{
...
CheckingAccount checkingAccount
....
try {
ModifyCheckingAccountCmd cmd =
new ModifyCheckingAccountCmdImpl(null, 1000);
cmd.setCheckingAccount(checkingAccount);
cmd.execute();
...
System.out.println("Would you like to undo this work? Enter Y or N");
try {
// Retrieve and validate user's response
...
}
...
if (answer.equalsIgnoreCase(Y)) {
Command compensatingCommand = cmd.getCompensatingCommand();
compensatingCommand.execute();
}
}
catch (Exception e) {
System.out.println(e.getMessage());
}
...
}
```

Helper class for running commands

WebSphere Application Server ships a CommandTarget enterprise bean to enable administrators to run a command in a designated server without providing their own implementation of CommandTarget. The EJBCommandTarget class, along with the EJBCommandTarget bean (CommandServerSessionBean), are located in the `EJBCommandTarget.jar` file in the `lib` directory under the WebSphere Application Server installation directory. This is a deployed jar file. You can use this JAR file in a new application or add it into an existing application.

The EJBCommandTarget class serves as a wrapper for a CommandTarget bean. CommandServerSessionBean is the WebSphere Application Server implementation of this CommandTarget bean. A command developer can set this EJBCommandTarget object into the Command. The following is a code example of the EJBCommandTarget bean:

```
EJBCommandTarget target = new EJBCommandTarget();
MyCommand cmd = new MyCommandImpl(Arguments...);
cmd.setCommandTarget(target);
cmd.execute();
```

In the code example, the client creates a MyCommand object. It is then executed in the application server. When the execute method is run, the target (EJBCommandTarget) looks up the CommandServerSessionHome from the InitialContext and executes the executeCommand method on the CommandServerSessionBean. The EJBCommandTarget object ensures that there is only one CommandServerSessionBean per object to avoid extra naming lookup.

An EJBCommandTarget object can be created using four different constructors:

- EJBCommandTarget(⌂MyNamingServerName⌂, ⌂PortNumber⌂, ⌂JNDIName⌂)
- EJBCommandTarget(InitialContext,⌂JNDIName⌂)
- EJBCommandTarget(⌂JNDIName⌂)
- EJBCommandTarget()

The first constructor enables the application to specify the naming server name and the port. The JNDI name of the CommandServerSessionBean can also be specified. The EJBCommandTarget constructs a `iiop://MyNamingServerName:PortNumber` provider URL and looks up the CommandServerSessionBean with the given JNDI name. If null values are passed in for any of the parameters, WebSphere Application Server defaults for server and port and a default JNDI name of CommandServerSession are used.

The second constructor enables the application to specify its own initial context. The EJBCommandTarget object then uses this initial context to look up the CommandServerSession bean with the specified JNDI name.

The third constructor enables the application to set up the naming server (the provider URL) in property files.

The default constructor uses the default values for the provider URL and default JNDI name for the CommandServerSession bean (CommandServerSession).

You do not need to use the EJBCommandTarget class. You can instead create your own custom target policy that uses the EJBCommandTarget bean (CommandServerSessionBean). The EJBCommandTarget object is a convenience class and attempts to address most usage scenarios.

Using a command

To use a command, the client creates an instance of the command and calls the execute method for the command. Depending on the command, calling other methods can be necessary. The specifics will vary with the application.

About this task

In the example application, the server is the CheckingAccountBean, an entity enterprise bean. In order to use this enterprise bean, the client gets a reference to the bean's home interface. The client then uses the reference to the home interface and one of the finder methods for the bean to obtain a reference to the remote interface for the bean. If there is no appropriate bean, the client can create one using a create method on the home interface.

The following code example illustrates the use of the **ModifyCheckingAccountCmd** command. This work takes place after an appropriate CheckingAccount bean has been found or created. The code instantiates a command, setting the input values by using one of the constructors defined for the command. The null argument indicates that the command should look up the server using the default target policy, and 1000

is the amount the command attempts to add to the balance of the checking account. After the command is instantiated, the code calls the setCheckingAccount method to identify the account to be modified. Finally, the execute method on the command is called.

```
{
...
CheckingAccount checkingAccount
...
try {
ModifyCheckingAccountCmd cmd =
new ModifyCheckingAccountCmdImpl(null, 1000);
cmd.setCheckingAccount(checkingAccount);
cmd.execute();
}
catch (Exception e) {
System.out.println(e.getMessage());
}
...
}
```

Example: Writing a command target (client-side adapter)

Commands can be used with any Java application, but the means of sending the command from the client to the server varies. The example in this topic shows how you can send a command to a servlet over the HTTP protocol. The client implements the CommandTarget interface locally.

The structure of a client-side adapter for a target

This example shows the structure of the client-side class; it implements the CommandTarget interface by implementing the executeCommand method.

```
...
import java.io.*;
import java.rmi.*;
import com.ibm.websphere.command.*;
public class ServletCommandTarget implements CommandTarget, Serializable
{
protected String hostName = "localhost";
public static void main(String args[]) throws Exception
{
....
}
public TargetableCommand executeCommand(TargetableCommand command)
throws CommandException
{
....
}
public static final byte[] serialize(Serializable serializable)
throws IOException {
... }
public String getHostName() {
... }
public void setHostName(String hostName) {
... }
private static void showHelp() {
... }
}
```

Instantiating the client-side adapter

The main method in the client-side adapter constructs and intializes the CommandTarget object, as follows:

```
public static void main(String args[]) throws Exception
{
String hostName = InetAddress.getLocalHost().getHostName();
String fileName = "MyServletCommandTarget.ser";
```

```
// Parse the command line
...
// Create and initialize the client-side CommandTarget adapter
ServletCommandTarget servletCommandTarget = new ServletCommandTarget();
servletCommandTarget.setHostName(hostName);
...
// Flush and close output streams
...
}
```

Example: Implementing a client-side adapter:

The CommandTarget interface declares one method, executeCommand, which the client implements. The executeCommand method takes a TargetableCommand object as input; it also returns a TargetableCommand.

A client-side implementation of the executeCommand method

This example shows the implementation of the method used in the client-side adapter. This implementation does the following:

- Serializes the command it receives
- Creates an HTTP connection to the servlet
- Creates input and output streams, to handle the command as it is sent to the server and returned
- Places the command on the output stream
- Sends the command to the server
- Retrieves the returned command from the input stream
- Returns the returned command to the caller of the executeCommand method

```
public TargetableCommand executeCommand(TargetableCommand command)
throws CommandException
{
try {
// Serialize the command
byte[] array = serialize(command);
// Create a connection to the servlet
URL url = new URL
("http://" + hostName +
"/servlet/com.ibm.websphere.command.servlet.CommandServlet");
HttpURLConnection httpURLConnection =
(HttpURLConnection) url.openConnection();
// Set the properties of the connection
...
// Put the serialized command on the output stream
OutputStream outputStream = httpURLConnection.getOutputStream();
outputStream.write(array);
// Create a return stream
InputStream inputStream = httpURLConnection.getInputStream();
// Send the command to the servlet
httpURLConnection.connect();
ObjectInputStream objectInputStream =
new ObjectInputStream(inputStream);
// Retrieve the command returned from the servlet
Object object = objectInputStream.readObject();
if (object instanceof CommandException) {
throw ((CommandException) object);
}
// Pass the returned command back to the calling method
return (TargetableCommand) object;
}
// Handle exceptions
....
}
```

Example: Running the command in the servlet:

The CommandTarget interface declares one method, executeCommand, which the client implements. The executeCommand method takes a TargetableCommand object as input; it also returns a TargetableCommand.

Running the command in the servlet

The servlet that runs the command is shown in the following example. The service method retrieves the command from the input stream and runs the performExecute method on the command. The resulting object, with any output properties that must be returned to the client, is placed on the output stream and sent back to the client.

```
...
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.websphere.command.*;
public class CommandServlet extends HttpServlet {
...
public void service(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
try {
...
// Create input and output streams
InputStream inputStream = request.getInputStream();
OutputStream outputStream = response.getOutputStream();
// Retrieve the command from the input stream
ObjectInputStream objectInputStream =
new ObjectInputStream(inputStream);
TargetableCommand command = (TargetableCommand)
objectInputStream.readObject();
// Create the command for the return stream
Object returnObject = command;
// Try to run the command's performExecute method
try {
command.performExecute();
}
// Handle exceptions from the performExecute method
...
// Return the command with any output properties
ObjectOutputStream objectOutputStream =
new ObjectOutputStream(outputStream);
objectOutputStream.writeObject(returnObject);
// Flush and close output streams
...
}
catch (Exception ex) {
ex.printStackTrace();
}
}
}
```

The target invokes the performExecute method on the command, but this is not always necessary. In some applications, it can be preferable to implement the work of the command locally. For example, the command can be used only to send input data, so that the target retrieves the data from the command and runs a local database procedure based on the input. You must decide the appropriate way to use commands in your application.

Running the IBM Thin Client for Enterprise JavaBeans (EJB)

An EJB Client is a Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) Java Platform, Standard Edition (Java SE) application that accesses remote Enterprise Java Beans from a server through Java Naming and Directory Interface (JNDI) look up. IBM Thin Client for EJB offers a smaller footprint and is easy to deploy to a Java SE environment and an Eclipse Rich Client Platform (RCP) environment. You can bundle the IBM Thin Client for EJB library using the WebSphere Application Server installation or the Application Client for WebSphere Application Server installation with your application. The IBM Thin Client for EJB also extends the choice of Java SE runtime. It can be run in the Java Runtime Environment (JRE) that is packaged with the WebSphere Application Server product, the Sun Microsystems JRE that is downloaded from the Sun Microsystems Web site, or the JRE that is downloaded from the HP Web site.

Before you begin

The IBM ORB implementation library is required if the IBM Thin Client for EJB is running with a non-IBM product JRE on a non-IBM product platform. For example, running the IBM Thin Client for EJB with Sun Microsystems JRE on Windows, Linux, or Solaris, and with the HP JRE on HPUX. The IBM-provided Solaris hybrid and HP hybrid JRE are not considered non-IBM product JRE environments. The IBM Thin Client for EJB can access version 2.x and version 3.0 EJB on the WebSphere Application Server using the JNDI lookup, but it cannot access version 3.0 EJB through resource injection. Resource injection is supported if the client application is a Java Platform, Enterprise Edition (Java EE) Application Client running within the Java Platform, Enterprise Edition (Java EE) Application Client Container.

Before you set up an EJB Thin Client environment, obtain the Java archive (JAR) file for the EJB Thin Client for WebSphere Application Server. To obtain the EJB Thin Client for WebSphere Application Server, install WebSphere Application Server or Application Client. The EJB Thin Client for WebSphere Application Server file, `com.ibm.ws.ejb.thinclient.zos_7.0.0.jar`, is located in the *app_server_root*`\runtimes` directory.

Copy the Java archive (JAR) file for the IBM Thin Client for EJB with WebSphere Application Server product, `com.ibm.ws.ejb.thinclient.zos_7.0.0.jar` , to other machines to create a lightweight client environment that enables communications with the products. Copies of the IBM Thin Client for EJB are subject to the same terms and conditions of the license agreement for Version 7.0 WebSphere Application Server where you obtained the Thin Client for EJB. Refer to the license agreements for correct usage and other limitations.

The IBM Thin Client for EJB with WebSphere Application Server runs on distributed operating systems with JDK support, including both Version 5 and Version 6. When using the IBM Thin Client for EJB as a standalone Java SE application with a non-IBM product JRE, you must override the default ORB implementation for the JRE through one of following methods:

- Include the com.ibm.ws.orb_7.0.0.jar file in the Java system classpath.
- Override the default ORB implementation in the JRE, using Java Endorsed Standards Override Mechanism.
- Set the java.endorsed.dirs path to a directory that contains the com.ibm.ws.orb_7.0.0.jar file.

When running the IBM Thin Client for EJB as an Eclipse RCP application, it is recommended to use method two, to override the default JRE ORB implementation.

Note: The Pluggable Application Client feature in the application client installer is deprecated. It is replaced by the IBM Thin Client for EJB.

About this task

Run the IBM Thin Client for EJB, by completing the following steps.

1. Invoke the client application. Run the following Java command:

Add the following system properties to the Java command if you want authentication and SSL enabled:

```
export LIBPATH=<app_server_root>/lib:$LIBPATH
<java_install_root>/bin/java
-classpath com.ibm.ws.ejb.thinclient.zos_7.0.0.jar:<list_of_your_application_jars_and_classes>
-Djava.naming.provider.url=iiop://<your_application_server_machine_name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
-Dcom.ibm.SSL.ConfigURL=file:///home/user1/ssl.client.props
-Dcom.ibm.CORBA.ConfigURL=file:///home/user1/sas.client.props
<fully_qualified_class_name_to_run>
```

2. Provide IIOP authentication configuration and Client SSL Configuration. Add the following system properties to the Java command:

   ```
   -Dcom.ibm.SSL.ConfigURL=file:///home/user1/ssl.client.props
   -Dcom.ibm.CORBA.ConfigURL=file:///home/user1/sas.client.props
   ```

   You can obtain the `ssl.client.props` file and `sas.client.props` file from the WebSphere Application Server installation and modify the file to suit your environment. You must, at a minimum, update the location of the key files in the `ssl.client.props` file to the match location of your target environment. For example,

   ```
   -Dcom.ibm.ssl.keyStore=/home/user1/etc/key.p12
   -Dcom.ibm.ssl.trustStore=/home/user1/etc/trust.p12
   ```

   Recommended SSL configuration settings when running the application with a non-IBM product JRE:

   ```
   com.ibm.ssl.protocol=SSL
   com.ibm.ssl.trustManager=SunX509
   com.ibm.ssl.keyManager=SunX509
   com.ibm.ssl.contextProvider=SunJSSE

   com.ibm.ssl.keyStoreType=JKS
   com.ibm.ssl.keyStoreProvider=SUN
   com.ibm.ssl.keyStore=/home/user1/etc/key.jks

   com.ibm.ssl.trustStoreType=JKS
   com.ibm.ssl.trustStoreProvider=SUN
   com.ibm.ssl.trustStore=/home/user1/etc/trust.jks
   ```

   The key store file and trust store file must be created using the Java keytool utility before the application runs. The automatic key file generation is not supported with a non-IBM product JRE.

   You must override the default ORB implementation of the non-IBM product JRE with the `com.ibm.ws.orb_7.0.0.jar` file, or add it to the classpath.

What to do next

Enable trace for the IBM Thin Client for EJB by adding the following to the Java command.

```
-Dcom.ibm.ejs.ras.lite.traceSpecification=*=all
```

Related tasks

Using JMS to connect to a WebSphere Application Server default messaging provider messaging engine

Related reference

ssl.client.props client configuration file

Use the `ssl.client.props` file to configure Secure Sockets Layer (SSL) for clients. In previous releases of WebSphere Application Server, SSL properties were specified in the `sas.client.props` or `soap.client.props` files or as system properties. By consolidating the configurations, WebSphere Application Server enables you to manage security in a manner that is comparable to server-side configuration management. You can configure the `ssl.client.props` file with multiple SSL configurations.

Chapter 6. Web services

Task overview: Implementing Web services applications

Use this topic as an introduction to using Web services. WebSphere Application Server supports Web services that are developed and implemented based on a variety of Java programming models. Use Web services when operating across a variety of platforms, including Java Platform, Enterprise Edition (Java EE) and non-Java EE platforms.

Before you begin

Decide if a Web services implementation benefits your business process.

About this task

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Implementing Web services applications is an easy way to integrate application systems together within or outside your business infrastructure that function as stand-alone systems. For example, your customer information database is a stand-alone application, but you want your accounting application to access the customer data. You can create a Web service for the customer database and then enable the accounting application as a Web service client. The accounting application can now access the customer information. By implementing a Web service, these two applications can share information in an efficient way.

Because Web services are easily applied to existing applications and information technology assets, you can develop, deploy and recompose new solutions quickly to address new opportunities. As Web services become more popular, the pool of services grows, promoting development of more robust models of just-in-time application and business integration over the Internet.

You can use Web services applications with the application server by following the steps provided:

1. Plan to use Web services. Review all of the components of Web services to learn how you can make your Web services plan more robust.
2. (Optional) Migrate existing Web services.

   Because Java EE environments emphasize compatibility, most application servers that offer support for the newer JAX-WS and JAXB specifications continue to support the older JAX-RPC specification. A consequence of this is that, existing Web services are likely to remain JAX-RPC based while new ones are developed using JAX-WS and JAXB.

   However, as time passes and applications are revised and rewritten, there might be times when the best strategy is to migrate a JAX-RPC based Web service to one based on JAX-WS and JAXB. This might result from a vendor choosing to provide enhancements to qualities of service that are only available in the new programming models. For example, SOAP 1.2 and SOAP Message Transmission Optimization Mechanism (MTOM) support are only available within the JAX-WS 2.x and JAXB 2.x

programming models and not JAX-RPC. Read about Web services migration best practices to learn more about best practices and examples when migrating JAX-RPC Web services to JAX-WS and JAXB Web services.

Note: Existing JAX-RPC applications wanting to use JAX-WS features must be rewritten using the JAX-WS programming model.

If you have used Web services based on Apache SOAP and now want to develop and implement Web Services for Java EE specification, you need to migrate client applications developed with all versions of 4.0, and versions of 5.0 prior to 5.0.2. See the topic, Migrating Apache SOAP Web services to JAX-RPC Web Services based on Java EE standards.

3. Develop Web services applications. You can develop Web services in one of the following ways:
   a. Develop Web services from existing WSDL files using JAX-WS.

      You can create a JAX-WS Web service by starting with an existing Web Services Description Language (WSDL) file describing the service interface for a JavaBeans or enterprise beans application. Typically, the WSDL file is defined as part of the application modeling process. Using an existing service definition or WSDL file to generate a new application is called a top-down approach to developing Web services.

   b. Develop Web services applications using JAX-WS.

      You can use the Java API for XML-Based Web Services (JAX-WS) programming model to develop Web services. JAX-WS simplifies application development through a standard, annotation-based model to develop Web services applications and clients. A common set of binding rules for XML and Java objects make it easy to incorporate XML data and process functions in Java applications. A further set of enhancements help you optimally send binary attachments, such as images or files, with the Web services requests.

      When developing a JAX-WS Web service starting from existing JavaBeans or stateless session enterprise beans, you can expose the bean as a JAX-WS Web service by using annotations. Adding the @WebService or @WebServiceProvider annotation to the bean defines the bean as a JAX-WS Web service. Enterprise beans that are exposed as JAX-WS Web services must be packaged in EJB 3.0 or higher modules.

      Transforming an existing application into Web services is called a bottoms-up approach to developing Web services. This process is called bottoms-up because you are starting with the implementation rather than starting with an existing service or Web Services Description Language (WSDL) file.

   c. Develop and deploy JAX-WS Web services clients Web services clients that can both access and invoke JAX-WS Web services are developed based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. The application server supports Enterprise JavaBeans™ (EJB) clients, Java EE application clients, JavaServer Pages (JSP) files and servlets that are based on the JAX-WS programming model.

   d. Develop Web services applications from existing WSDL files with JAX-RPC.

      You can create a JAX-RPC Web service by starting with an existing WSDL file describing the service interface of an enterprise bean implementation using a top-down approach to developing Web services.

   e. Develop Web services applications with JAX-RPC.

      You can use the Java API for XML-based RPC (JAX-RPC) programming model to develop Web services. When developing a JAX-RPC Web service starting from existing JavaBeans or enterprise beans, you need develop a WSDL file. You can use existing JavaBeans or enterprise beans and then enable the implementation for Web services.

   f. Develop and deploy JAX-RPC Web services clients You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-based remote procedure call (JAX-RPC) specification. The application server supports Enterprise JavaBeans™ (EJB) clients, Java EE application clients, JavaServer Pages (JSP) files and servlets that are based on the JAX-RPC programming model.

g. Enable Web services through service integration technologies

   You can use the Web services enablement of the service integration bus (SIBus) to achieve the following goals:

   - Make an internally hosted service that is available at a bus destination available as a Web service.
   - Make an external Web service available internally at a bus destination.
   - Use the Web services gateway to map an existing service, either an internally-hosted service or an external Web service, to a new Web service that is provided by the gateway.

You can develop Web services to take advantage of Web Services Addressing (WS-Addressing), Web Services Resource Framework (WSRF), and Web Services Transaction (WS-Transaction) support.

- Use the WS-Addressing SPI: Performing more advanced Web Service Addressing tasks.

  You can develop Web services to take advantage of Web Services Addressing (WS-Addressing), which aids interoperability between Web services using a standard way to address Web services and providing addressing information in messages.
- Create stateful Web services using the Web Services Resource Framework.

  With the Web Services Resource Framework (WSRF) support in the application server, you can implement a stateful Web service as a WS-Resource, and reference that service it using a WS-Addressing endpoint reference.
- Use WS-Transaction policy to coordinate transactions or business activities for Web services.

  WS-Transaction is an interoperability standard that includes the WS-AtomicTransaction, WS-BusinessActivity, and WS-Coordination specifications. The Web Services Atomic Transaction (WS-AT) support in the application server provides transactional quality of service to the Web services environment. Distributed Web services applications, and the resources they use, can take part in distributed global transactions. With Web Services Business Activity (WS-BA) support in the application server, Web services on different systems can coordinate activities that are more loosely coupled than atomic transactions. Such activities can be difficult or impossible to roll back atomically, and therefore require a compensation process if an error occurs.
- Use WS-Policy to exchange policies in a standard format.

  WS-Policy is an interoperability standard that is used to describe and communicate the policies of a Web service so that service providers can export policy requirements in a standard format. Clients can combine the service provider requirements with their own capabilities to establish the policies required for a specific interaction.

4. Assemble Web services.

   Read about what you need to assemble a Web service and the order in which to assemble the parts, for example an enterprise archive (EAR) file.
5. Deploy Web services.

   Read about the steps necessary to deploy the EAR file that has been configured and enabled for Web services.
6. Administer deployed Web services.

   Once your Web services application is deployed, you can configure security settings, view deployment descriptors and WSDL documents, set the scope of a Web service port, and manage policy sets and service providers. These tasks can be done using the administrative console or with command-line tools.
7. Secure Web services.

   - Manage policy sets using the administrative console.

     Read about creating policy sets that you can use to simplify the security configuration of your Web services applications. Policy sets are only supported by JAX-WS applications.
   - Secure Web services applications using message level security.

Consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, delegation, and auditing across a spectrum of application and business topologies.

8. Publish the WSDL file.

   After installing a Web services application, and optionally modifying the endpoint information, you might need Web Services Description Language (WSDL) files containing the updated endpoint information. Read about the steps necessary to publish the WSDL files so that this information is available.

9. Monitor the performance of Web services applications.

   Read about using the Performance Monitoring Infrastructure (PMI) to measure the time required to process Web services requests.

10. Troubleshoot Web services.

    Read about troubleshooting different processes used to develop, implement and use Web services, including command-line tools, Java compiling errors, client runtime errors and exceptions, serialization and deserialization errors, and authentication challenges and authorization failures with Web services security.

Example

The following example illustrates how a business might use Web services.

The owner of a flower shop wants to start receiving orders from customers through the Web. This owner starts the process by finding wholesale flower suppliers, pricing the product, and completing contracts for future flower orders.

Using Web services, the flower shop owner can find wholesale flower suppliers. One way to find new suppliers is to use a Universal Description, Discovery and Integration (UDDI) registry to search for potential suppliers. When the suppliers are chosen, the registry sends back information on how to contact the flower distributors that meet the criteria of the flower shop owner.

The flower shop owner can request price lists from each of the suppliers by obtaining a WSDL file for each potential supplier. The WSDL can be downloaded from the Web page of the supplier, received through e-mail, or retrieved from the UDDI registry entry of the supplier.

The WSDL describes the procedure call. When using the application server, the procedure call is a JAX-RPC or a JAX-WS procedure call. Either of these procedure call types retrieves the price list. The WSDL file also specifies the Universal Resource Locator (URL), where the request is sent.

The flower shop owner now has to compare the prices received from each supplier, decide which suppliers to do business with, and make arrangements for future orders to fill. The flower shop can now sell merchandise through the Web by using Web services to communicate with suppliers for the best prices and complete the ordering processes. The merchandise price lists need publishing to the Web site and a mechanism is needed for customers to order flowers.

The Web services clients of the flower supplier are deployed on the flower shop server. When a customer makes a transaction to purchase flowers through the Web, the order is sent to the supplier through the procedure call. The supplier responds by sending a confirmation with the order number and shipping date. The suppliers maintain the inventory and the flower shop owner handles billing and customer order management.

Similarly, the flower shop catalog can be composed automatically from the catalogs of each supplier. If the supplier delivers directly to the customer, then the order tracking inquiries can pass directly to the order tracking system of the supplier. The supplier can also use Web services to send invoices for orders by the

flower shop. Processes that previously required forms to fill manually, and fax or mail, can now be done automatically, saving labor costs for both the flower shop and the supplier.

Using Web services is beneficial because a much larger inventory is made available to the flower shop. No merchandise maintenance overhead exists, and the flower shop can offer their customers products that they otherwise might not have. Selling flowers through the Web increases capital for the flower shop without overhead of another store or resources invested into additional products.

For a more detailed scenario, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer, Plants by WebSphere, and how they incorporated the Web services concept.

Refer to the Samples Gallery for additional Samples that demonstrate JAX-WS and JAX-RPC Web services.

Service-oriented architecture

A *service-oriented architecture* (SOA) is a collection of services that communicate with each other, for example, passing data from one service to another or coordinating an activity between one or more services.

Companies want to integrate existing systems to implement Information Technology (IT) support for business processes that cover the entire business value chain. A variety of designs are used, ranging from rigid point-to-point electronic data interchange (EDI) to Web auctions. By using the Internet, companies can make their IT systems available to internal departments or external customers, but the interactions are not flexible and are without standardized architecture.

Because of this increasing demand for technologies that support connecting and sharing resources and data, a need exists for a flexible, standardized architecture. SOA is a flexible architecture that unifies business processes by structuring large applications into building blocks, or small modular functional units or services, for different groups of people to use inside and outside the company. The building blocks can be one of three roles: service provider, service broker, or service requestor. See Web services approach to a service-oriented architecture to learn more about these roles.

Requirements for an SOA

To efficiently use an SOA, follow these requirements:

- **Interoperability between different systems and programming languages**.

  The most important basis for a simple integration between applications on different platforms is to provide a communication protocol. This protocol is available for most systems and programming languages.
- **Clear and unambiguous description language**.

  To use a service offered by a provider, it is not only necessary to be able to access the provider system, but the syntax of the service interface must also be clearly defined in a platform-independent fashion.
- **Retrieval of the service**.

  To support a convenient integration at design time or even system run time, a search mechanism is required to retrieve suitable services. Classify these services as *computer-accessible*, *hierarchical* or *taxonomies* based on what the services in each category do and how they can be invoked.

Web services approach to a service-oriented architecture

You can use Web services in a service-oriented architecture (SOA) environment.

You can use Web services to implement a SOA. A major focus of Web services is to make functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages. These services can be new applications or just wrapped around existing legacy systems to make them network-enabled. A service can rely on another service to achieve its goals.

Each SOA building block can assume one or more of three roles:

- **Service provider**

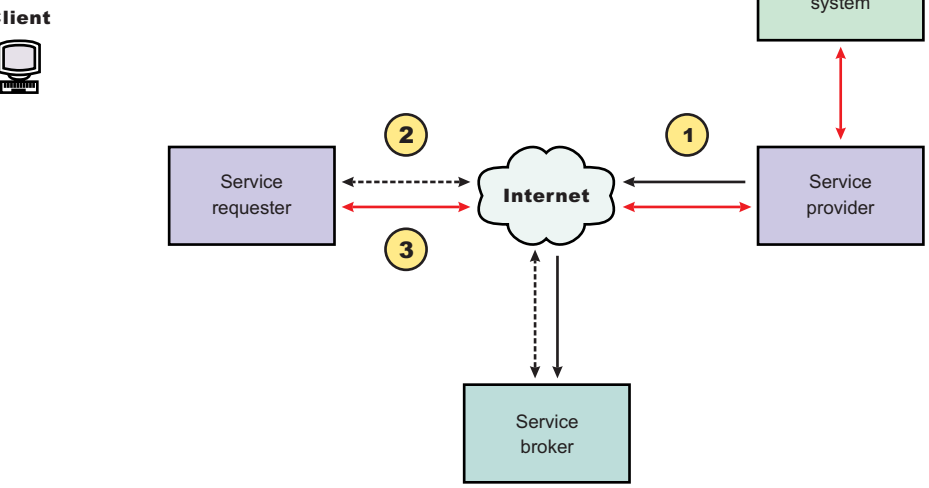
  The service provider creates a Web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or how to exploit free services for other value. The provider also has to decide which category to list the service in for a given broker service and what sort of trading partner agreements are required to use the service.

- **Service broker**

  The service broker, also known as *service registry*, is responsible for making the Web service interface and implementation access information available to any potential service requestor. The implementer of the broker decides the scope of the broker. Public brokers are available through the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, some decisions need to be made about the amount of the offered information. Some brokers specialize in many listings. Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. Some brokers catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, the number of listings or the accuracy of the listings. The Universal Description, Discovery and Integration (UDDI) specification defines a way to publish and discover information about Web services.

- **Service requester**

  The service requestor or Web service client locates entries in the broker registry using various find operations and then binds to the service provider to invoke one of its Web services.



Characteristics of the SOA

The presented SOA illustrates a loose coupling between the participants, which provides greater flexibility in the following ways:

- A client is coupled to a service. Therefore, the integration of the server takes place outside the scope of the client application programs.
- Old and new functional blocks or applications and systems, are encapsulated into components that work as services.
- Functional components and their interfaces are separate so that new interfaces can be plugged in more easily.
- Within complex applications, the control of business processes can be isolated. A business rule engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.

- Services can be incorporated dynamically during run time.
- Bindings are specified using configuration files and can be easily adapted to new needs.

Properties of a service-oriented architecture

The service-oriented architecture offers the following properties:

- **Web services are self-contained**

  On the client side, no additional software is required. A programming language with Extensible Markup Language (XML) and HTTP client support is enough to get you started. On the server side, a Web server and a SOAP server are required. It is possible to enable an existing application for Web services without writing a single line of code.
- **Web services are self-describing**

  Neither the client nor the server knows or cares about anything besides the format and content of the request and response messages (loosely coupled application integration). The definition of the message format travels with the message; no external metadata repositories or code generation tool are required.
- **Web services can be published, located, and invoked across the Internet**

  This technology uses established lightweight Internet standards such as HTTP and it leverages the existing infrastructure. Some other standards that are required include, SOAP, Web Services Description Language (WSDL), and UDDI.
- **Web services are language-independent and interoperable**

  The client and server can be implemented in different environments. Existing code does not have to change in order to be Web services-enabled.
- **Web services are inherently open and standard-based**

  XML and HTTP are the major technical foundation for Web services. A large part of the Web service technology has been built using open-source projects.
- **Web services are dynamic**

  Dynamic e-business can become reality using Web services because with UDDI and WSDL you can automate the Web service description and discovery.
- **Web services are composable**

  Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This chaining shortens development time and enables best-of-breed implementations.
- **Web services are loosely coupled**

  Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that supports a more flexible reconfiguration for an integration of the services.
- **Web services provide programmatic access**

  The approach provides no graphical user interface; it operates at the code level. Service consumers need to know the interfaces to Web services, but do not need to know the implementation details of services.
- **Web services provide the ability to wrap existing applications**

  Already existing stand-alone applications can easily integrate into the SOA by implementing a Web service as an interface.

Web services business models supported

This article explains the concept and business models that can be implemented by using Web services in a service-oriented architecture (SOA).

The properties and benefits of using a SOA such as Web services is well suited for binding small modules that perform independent tasks within a highly heterogeneous e-business model. Web services can be easily wrapped around existing applications in your business model and plugged into different business processes.

For connecting to a large monolithic system that does not support the implementation of different flexible business processes, other approaches might be better suited, for example, to satisfy specialized features, such as performance or security.

The following business models are easily implemented by using an architecture including Web services:

- **Business information**

  Sharing of information with consumers or other businesses. Web services can be used to expand the reach through such services as news streams, local weather reports, integrated travel planning, and intelligent agents.
- **Business integration**

  Providing transactional, fee-based services for customers. A global network of suppliers can be easily created. Web services can be implemented in auctions, e-marketplaces, and reservation systems.
- **Business process externalization**

  Web services can be used to model value chains by dynamically integrating processes to a new solution within an organizational unit or even with those of other e-businesses. This modeling can be achieved by dynamically linking internal applications to new partners and suppliers, to offer their services to complement internal services.

To see how these models are implemented using all aspects of Web services, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how this retailer incorporates the Web services concept.

Web services

Web services are self-contained, modular applications that you can describe, publish, locate, and invoke over a network.

The application server supports Web services that are developed and implemented based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. The application server supports the Java API for XML Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. The JAX-WS is a strategic programming model that simplifies application development through support of a standard, annotation-based model to develop Web services applications and clients.

A typical Web services scenario is a business application requesting a service from another existing application. The request is processed through a given Web address using SOAP messages over a HTTP, Java Message Service (JMS) transport or invoked directly as Enterprise JavaBeans (EJB). The service receives the request, processes it, and returns a response. Examples of a simple Web service include weather reports or getting stock quotes. The method call is synchronous, that is, the method waits until the result is available. Transaction Web services, supporting quotes, business-to-business (B2B) or business-to-client (B2C) operations include airline reservations and purchase orders.

Web services can include the actual service or the client that accesses the service.

Web services are Web applications that help improve the flexibility of your business processes by integrating with applications that otherwise do not communicate. The inner-library loan program at your local library is a good example of the Web services concept and its evolution. The Web service concept existed even before the term; the concept became widely accepted with the creation of the Internet. Before the Internet was created, you visited your library, searched the collections and checked out your books. If you did not find the book that you wanted, the librarian ran a search for you by computer or phone and

located the book at a nearby library. The librarian ordered the book for you and you picked it up after it was delivered to your local library. By incorporating Web services applications, you can streamline your library visit.

Now, you can search the local library collection and other local libraries at the same time. When other libraries provide your library with a Web service to search their collection (the service might have been provided through Universal Description Discovery and Integration (UDDI), your results yield their resources. You might use another Web service application to check out and send the book to your home. Using Web services applications saves time and provides a convenience for you, as well as freeing the librarian to do other business tasks.

Web services reflect the service-oriented architecture (SOA) approach to programming. This approach is based on the idea of building applications by discovering and implementing network-available services, or by invoking the available applications to accomplish a task. Web services deliver interoperability, for example, Web services applications provide components created in different programming languages to work together as if they were created using the same language. Web services rely on existing transport technologies, such as HTTP, and standard data encoding techniques, such as Extensible Markup Language (XML), for invoking the implementation.

The key components of Web services include:

- Web Services Description Language (WSDL)

  WSDL is the XML-based file that describes the Web service. The Web service request uses this file to bind to the service.
- SOAP

  SOAP is the XML-based protocol that the Web service request uses to invoke the service.
- Universal Description, Discovery and Integration Protocol (UDDI)

  UDDI is the registry that hosts the service broker. UDDI is similar to the Yellow Pages in a phone book.

For a more detailed scenario, see Web services scenario: Overview, which tells the story of a fictional online garden supply retailer named Plants by WebSphere, and how this retailer incorporated the Web services concept.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Web Services for Java EE specification

The *Web Services for Java Platform, Enterprise Edition (Java EE)* specification defines the programming model and runtime architecture for implementing Web services based on the Java language. Another name for the Web Services for Java EE specification is the Java Specification Requirements (JSR) 109. The specification includes open standards for developing and implementing Web services.

The Web Services for Java EE specification is based on the Java EE technology and supports the Java API for XML Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) programming model for Web services and clients in a manner that is interoperable and portable across application servers within environments that are scalable and secure. This specification is based on industry standards for Web services, including Web Services Description Language (WSDL) and SOAP, and it describes the development and deployment of Web services.

The application server supports the Web Services for Java EE specification, Version 1.2. This specification supports WSDL Version 1.1, SOAP Version 1.1 and SOAP Version 1.2. Prior to Web Services for Java EE Version 1.2, the JSR 109 specification name was Web Services for Java 2 Platform, Enterprise Edition (J2EE).

You can integrate the Java EE technology with Web services in a variety of ways. You can expose Java EE components as Web services, for example, JavaBeans and enterprise beans. When you expose Java

EE components as Web services, clients that are written in Java code or existing Web service clients that are not written in Java code can access these services. Java EE components can also act as Web service clients.

The Web Services for Java EE specification is the preferred platform for Web-based programming because it provides open standards permitting different types of languages, operating systems and software to communicate seamlessly through the Internet.

For a Java application to act as Web service client, a mapping between the WSDL file and the Java application must exist. For JAX-WS applications, the mapping is defined using annotations. You can optionally use the webservices.xml deployment descriptor to specify the location of the WSDL file and override the value defined in the @WebServices annotation. For JAX-RPC applications, you must define the JAX-RPC mapping file. To learn more about the mapping that is defined between the WSDL file and your Web service application, see the JAX-WS specification or the JAX-RPC specification in the Web services specifications and API documentation depending on the programming model used.

You can use a Java component to implement a Web service by specifying the component interface and binding information in the WSDL file and designing the application server infrastructure to accept the service request.

This entire process encompassed is based on the Web Services for Java EE specification.

The specification defines the webservices.xml deployment descriptor specifically for Web services. The webservices.xml deployment descriptor file defines the set of Web services that you can deploy in a Web Services for Java EE enabled container.

For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

For example, if your service implementation class for your JAX-WS Web service includes the @WebService annotation as follows:

```
@WebService(wsdlLocation="http://myhost.com/location/of/the/wsdl/ExampleService.wsdl")
```

and the webservices.xml specifies a different filename for the WSDL document as follows:

```
<webservices>
<webservice-description>
<webservice-description-name>ExampleService</webservice-description-name>
<wsdl-file>META-INF/wsdl/ExampleService.wsdl</wsdl-file>
...
</webservice-description>
</webservices>
```

then the value that is specified in the deployment descriptor, META-INF/wsdl/ExampleService.wsdl overrides the annotation value.

See section 5 of the Web Services for Java EE specification for details regarding the correlation between values specified in the Web services deployment descriptor file and the attributes of the @WebService and the @WebServiceProvider annotations.

For JAX-RPC Web services, you must define the deployment characteristics in the webservices.xml deployment descriptor file.

You are responsible for providing various elements to the deployment descriptor, including:

- Port name
- Port service implementation
- Port service endpoint interface
- Port WSDL definition
- Port QName
- MTOM/XOP support for JAX-WS Web services
- Protocol binding for JAX-WS Web services
- JAX-RPC mapping
- Handlers (optional)
- Servlet mapping (optional)

The Enterprise JavaBeans (EJB) 2.1 specification also states that for a Web service developed from a session bean, the EJB deployment descriptor, `ejb-jar.xml`, must contain the service-endpoint element. The service-endpoint value must be the same as that stated in the webservices.xml deployment descriptor.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Artifacts used to develop Web services

With *development artifacts* you can develop an enterprise bean or a JavaBeans module into a Web service. This topic describes artifacts used to develop Web services that are based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification.

To create a Web service from an enterprise bean or from a JavaBeans module, the following files are added to the respective Java archive (JAR) file or Web archive (WAR) modules at assembly time:

- **Web Services Description Language (WSDL) Extensible Markup Language (XML) file**

  The WSDL XML file describes the Web service that is implemented.
- **Service Endpoint Interface**

  A Service Endpoint Interface is the Java interface corresponding to the Web service port type implemented. The Service Endpoint Interface is defined by the Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) Web services run time that you are using.
- **webservices.xml**

  The webservices.xml file contains the Java EE deployment descriptor of the Web service specifying how the Web service is implemented. The webservices.xml file is defined in the Web Services for Java EE specification.

  For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

  For JAX-RPC applications, deployment descriptors are required to specify how the Web service is implemented.
- **ibm-webservices-bnd.xmi (JAX-RPC applications only)**

  This file contains WebSphere product-specific deployment information and is defined in ibm-webservices-bnd.xmi assembly properties.
- **Java API for XML-based remote procedure call (JAX-RPC) mapping file**

  The JAX-RPC mapping deployment descriptor specifies how Java elements are mapped to and from WSDL file elements.

The following files are added to an application client, enterprise beans or Web module to permit a Web Services for Java Platform, Enterprise Edition (Java EE) client access to Web services:

- **WSDL file**

The WSDL file is provided by the Web service implementer.

- **Java interfaces for the Web service**

  The Java interfaces are generated from the WSDL file as specified by the JAX-WS or JAX-RPC specification. These bindings are the Service Endpoint Interface based on the WSDL port type, or the service interface, which is based on the WSDL service.
- **ibm-webservicesclient-bnd.xmi (JAX-RPC applications only)**

  This file contains WebSphere product-specific deployment information, such as security information for JAX-RPC applications. For JAX-WS applications, deployment descriptors are not supported and have been replaced by the use of annotations.
- **Other JAX-RPC binding files**

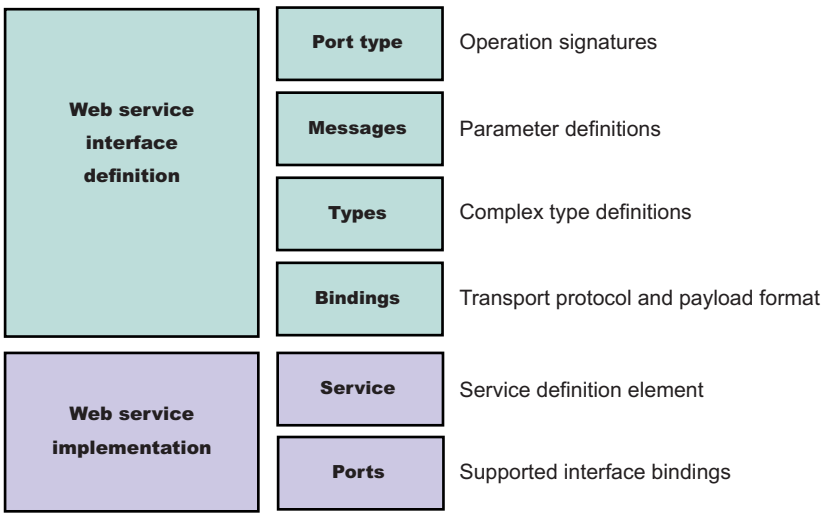
  Additional JAX-RPC binding files that support the client application in mapping SOAP to the Java language are generated from WSDL by the WSDL2Java command tool.

WSDL

Web Services Description Language (WSDL) is an Extensible Markup Language (XML)-based description language. This language was submitted to the World-Wide Web Consortium (W3C) as the industry standard for describing Web services. The power of WSDL is derived from two main architectural principles: the ability to describe a set of business operations and the ability to separate the description into two basic units. These units are a description of the operations and the details of how the operation and the information associated with it are packaged.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definitions of endpoints and messages are separated from their concrete network deployment or data format bindings. This separation supports the reuse of abstract definitions: messages, which are abstract descriptions of exchanged data, and port types, which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Therefore, a WSDL document is composed of several elements.

The following is the structure of the information in a WSDL file:



A WSDL file contains the following parts:

- **Web service interface definition**

  This part contains the elements and the namespaces.
- **Web service implementation**

  This part contains the definition of the service and ports.

A WSDL file describes a Web service with the following elements:

portType

The description of the operations and associated messages. The portType element defines abstract operations.

```
<portType name="EightBall">
 <operation name="getAnswer">
  <input message="ebs:IngetAnswerRequest"/>
  <output message="ebs:OutgetAnswerResponse"/>
 </operation>
</portType>
```

message

The description of input and output parameters and return values.

```
<message name="IngetAnswerRequest">
 <part name="meth1_inType" type="ebs:questionType"/>
</message>
<message name="OutgetAnswerResponse">
 <part name="meth1_outType" type="ebs:answerType"/>
</message>
```

types

The schema for describing XML types used in the messages.

```
<types>
 <xsd:schema targetNamespace="...">
  <xsd:complexType name="questionType">
   <xsd:element name="question" type="string"/>
  </xsd:complexType>
  <xsd:complexType name="answerType">
  ...
</types>
```

binding

The bindings describe the protocol that is used to access a portType, as well as the data formats for the messages that are defined by a particular portType element.

```
<binding name="EightBallBinding" type="ebs:EightBall">
 <soap:binding style="rpc" transport="schemas.xmlsoap.org/soap/http">
 <operation name="ebs:getAnswer">
 <soap:operation soapAction="urn:EightBall"/>
  <input>
   <soap:body namespace="urn:EightBall" ... />
  ...
```

Service

The services and ports define the location of the Web service.

The service contains the Web service name and a list of ports.

Ports

The ports contain the location of the Web service and the binding used for service access.

```
<service name="EightBall">
 <port binding="ebs:EightBallBinding" name="EightBallPort">
  <soap:address location="localhost:8080/axis/EightBall"/>
 </port>
</service>
```

When creating Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) Web services, you can use a bottom-up development approach when you start from JavaBeans or an enterprise bean, or you can use a top-down development approach when you start with an existing Web Services Description Language (WSDL) file.

When creating JAX-WS Web services for this product, you can start with either a WSDL or an implementation bean class. If you start with an implementation bean class, then use the wsgen command line tool to generate all the Web services server artifacts, including a WSDL if requested. If you start with a WSDL, then use the wsimport command line tool to generate all the Web services artifacts for either the server or client side.

When creating JAX-RPC Web services for this product, you must first have an implementation bean that includes a service endpoint interface. Then, you use the Java2WSDL command-line tool to create a WSDL file that defines the Web services. If you are starting with the WSDL to generate the implementation bean class, run the WSDL2Java command line tool against the WSDL file to create Java APIs and deployment descriptor templates.

Multipart WSDL and WSDL publication

The product supports deployment of Web services using a multipart Web Services Description Language (WSDL) file. In multipart WSDL files, an implementation WSDL file contains the wsdl:service. This implementation WSDL file imports an interface WSDL file, which contains the other WSDL constructs. This supports multiple Web services using the same WSDL interface definition.

The <wsdl:import> element indicates a reference to another WSDL file. If the <wsdl:import> element location attribute does not contain a URL, that is, it contains only a file name, and does not begin with http://, https:// or file://, the imported file must be located in the same directory and must not contain a relative path component. For example, if META-INF/wsdl/A_Impl.wsdl is in your module and contains the <wsdl:import=″A.wsdl″ namespace=″...″/> import statement, the A.wsdl file must also be located in the module META-INF/wsdl directory.

It is recommended that you place all WSDL files in either the META-INF/wsdl directory, if you are using Enterprise JavaBeans (EJB), or the WEB-INF/wsdl directory, if you are using JavaBeans components, even if relative imports are located within the WSDL files. Otherwise, implications exist with the WSDL publication when you use a path like <location=″../interfaces/A_Interface.wsdl″namespace=″...″/>. Using a path like this example fails because the presence of the relative path, regardless of whether the file is located at that path or not. If the location is a Web address, it must be readable at both deployment and server startup.

You can publish the files located in the META-INF/wsdl or the WEB-INF/wsdl directory through either a URL address or file, including WSDL or XML Schema Definition (XSD) files. For example, if the file referenced in the <wsdl-file> element of the webservices.xml deployment descriptor is located in the META-INF/wsdl or the WEB-INF/wsdl directory, it is publishable. If the files imported by the <wsdl-file> are located in the wsdl/ directory or its subdirectory, they are publishable.

If the WSDL file referenced by the <wsdl-file> element is located in a directory other than wsdl, or its subdirectories, the file and its imported files, either WSDL or XSD files, which are in the same directory, are copied to the wsdl directory without modification when the application is installed. These types of files can also be published.

If the <wsdl-file> imports a file located in a different directory (a directory that is not -INF/wsdl or a subdirectory), the file is not copied to the wsdl directory and not available for publishing.

For JAX-WS Web services, you can use an annotation to specify the location of the WSDL. Use the @WebService annotation with the WSDLLocation attribute. The WSDLLocation attribute is optional. If this attribute is not specified, then WSDL is generated and published from the information that is found in the Web service classes. You can optionally specify the location of the WSDL file in the webservices.xml deployment descriptor. However, any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

SOAP

SOAP is a specification for the exchange of structured information in a decentralized, distributed environment. As such, it represents the main way of communication between the three key actors in a service oriented architecture (SOA): service provider, service requestor and service broker. The main goal of its design is to be simple and extensible. A SOAP message is used to request a Web service.

SOAP 1.1

WebSphere Application Server follows the standards outlined in SOAP 1.1.

SOAP was submitted to the World Wide Web Consortium (W3C) as the basis of the Extensible Markup Language (XML) Protocol Working Group by several companies, including IBM and Lotus®. This protocol consists of three parts:

- An *envelope* that defines a framework for describing message content and processing instructions.
- A set of *encoding rules* for expressing instances of application-defined data types.
- A *convention* for representing remote procedure calls and responses.

SOAP 1.1 is a protocol-independent transport and can be used in combination with a variety of protocols. In Web services that are developed and implemented with WebSphere Application Server, SOAP is used in combination with HTTP, HTTP extension framework, and Java Message Service (JMS). SOAP is also operating-system independent and not tied to any programming language or component technology.

As long as the client can issue XML messages, it does not matter what technology is used to implement the client. Similarly, the service can be implemented in any language, as long as the service can process SOAP messages. Also, both server and client sides can reside on any suitable platform.

SOAP 1.2

The SOAP 1.2 specification is also a W3C recommendation, and WebSphere Application Server follows the standards that are outlined in SOAP 1.2. The SOAP 1.2 specification comes in three parts plus some assertions and a test collection:

- Part 0: Primer
- Part 1: Messaging Framework
- Part 2: Adjuncts
- Specification Assertions and Test Collection

SOAP 1.2 provides a more specific definition of the SOAP processing model, which removes many of the ambiguities that sometimes led to interoperability problems in the absence of the Web Services-Interoperability (WS-I) profiles. SOAP 1.2 should reduce the chances of interoperability issues with SOAP 1.2 implementations between different vendors.

Some of the more significant changes in the SOAP 1.2 specification include:

- The ability to now officially define other transport protocols other than the HTTP protocol as long as vendors conform to the binding framework that is defined in SOAP 1.2. While HTTP is ubiquitous, it is not as reliable of a transport as other things such as TCP/IP, MQ, and so forth.
- The fact that SOAP 1.2 is based on the XML Information Set (XML Infoset). The information set provides a way to describe the XML document using the XSD schema but does not necessarily serialize the document by using XML 1.0 serialization. SOAP 1.1 is based upon XML 1.0 serialization. The information set will make it easier to use other serialization formats such as a binary protocol format. You can use a binary protocol format shrink the message into a much more compact format where some of the verbose tagging information might not be required.

The Java API for XML Web Services (JAX-WS) standard introduces the ability to support both SOAP 1.1 as well as SOAP 1.2.

See “Differences in SOAP versions” on page 440 for additional differences between SOAP 1.1 and SOAP 1.2.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

SOAP with Attachments API for Java interface:

The *SOAP with Attachments API for Java* (SAAJ) interface is used for SOAP messaging that provides a standard way to send XML documents over the Internet from a Java programming model. SAAJ is used to manipulate the SOAP message to the appropriate context as it traverses through the runtime environment.

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

The Java API for XML-Based RPC (JAX-RPC) programming model supports SAAJ 1.2 to manipulate the XML.

The JAX-WS programming model supports SAAJ 1.2 and 1.3. SAAJ 1.3 includes support for SOAP 1.2 messages.

The differences in the SAAJ 1.2 and SAAJ 1.3 specification can be reviewed in the topic ″Differences in SAAJ versions.″

How are messages used in Web services?

Web services use XML technology to exchange messages. These messages conform to XML schema. When developing Web services applications, there are limited XML APIs to work with, for example, Document Object Model (DOM). It is more efficient to manipulate the Java objects and have the serialization and deserialization completed during run time.

Web services uses SOAP messages to represent remote procedure calls between the client and the server. Typically, the SOAP message is deserialized into a series of Java value-type business objects that represent the parameters and return values. In addition, the Java programming model provides APIs that support applications and handlers to manipulate the SOAP message directly. Because there are a limited number of XML schema types that are supported by the programming models, the specification provides the SAAJ data model as an extension to manipulate the message.

To manipulate the XML schema types, you need to map the XML schema types to Java types with a *custom data binder*.

The SAAJ interface

The SAAJ-related classes are located in the javax.xml.soap package. SAAJ builds on the interfaces and abstract classes and many of the classes begin by invoking factory methods to create a factory such as SOAPConnectionFactory and SOAPFactory.

The most commonly used classes are:

- SOAPMessage: Contains the message, both the XML and non-XML parts
- SOAPHeader: Represents the SOAP header XML element
- SOAPBody: Represents the SOAP body XML element
- SOAPElement: Represents the other elements in the SOAP message

Other parts of the SAAJ interface include:

- MessageContext: Contains a SOAP message and related properties
- AttachmentPart: Represents a binary attachment
- SOAPPart: Represents the XML part of the message
- SOAPEnvelope: Represents the SOAP envelope XML element
- SOAPFault: Represents the SOAP fault XML element

The primary interface in the SAAJ model is javax.xml.soap.SOAPElement, also referred to as SOAPElement. Using this model, applications can process an SAAJ model that uses pre-existing DOM code. It is also easier to convert pre-existing DOM objects to SAAJ objects.

Messages created using the SAAJ interface follow SOAP standards. A SOAP message is represented in the SAAJ model as a javax.xml.soap.SOAPMessage object. The XML content of the message is represented by a javax.xml.soap.SOAPPart object. Each SOAP part has a SOAP envelope. This envelope is represented by the SAAJ javax.xml.SOAPEnvelope object. The SOAP specification defines various elements that reside in the SOAP envelope; SAAJ defines objects for the various elements in the SOAP envelope.

The SOAP message can also contain non-XML data that is called attachments. These attachments are represented by SAAJ AttachmentPart objects that are accessible from the SOAPMessage object.

A number of reasons exist as to why handlers and applications use the generic SOAPElement API instead of a tightly bound mapping:

- The Web service might be a conduit to another Web service. In this case, the SOAP message is only forwarded.
- The Web service might manipulate the message using a different data model, for example a Service Data Object (SDO). It is easier to convert the message from a SAAJ DOM to a different data model.
- A handler, for example, a digital signature validation handler, might want to manipulate the message generically.

You might need to go a step further to map your XML schema types, because the SOAPElement interface is not always the best alternative for legacy systems. In this case you might want to use a generic programming model, such as SDO, which is more appropriate for data-centric applications.

The XML schema can be configured to include a custom data binding that pairs the SDO or data object with the Java object. For example, the run time renders an incoming SOAP message into a SOAPElement interface and passes it to the customer data binder for more processing. If the incoming message contains

an SDO, the run time recognizes the data object code, queries its type mapping to locate a custom binder, and builds the SOAPElement interface that represents the SDO code. The SOAPElement is passed to the SDOCustomBinder.

See Custom data binders for more information about the process of developing applications with SOAPElement, SDO and custom binders.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Related concepts

“SOAP” on page 429
SOAP is a specification for the exchange of structured information in a decentralized, distributed environment. As such, it represents the main way of communication between the three key actors in a service oriented architecture (SOA): service provider, service requestor and service broker. The main goal of its design is to be simple and extensible. A SOAP message is used to request a Web service.

“Differences in SOAP versions” on page 440
Both SOAP Version 1.1 and SOAP Version 1.2 are World Wide Web Consortium (W3C) standards. Web services can be deployed that support not only SOAP 1.1 but also support SOAP 1.2. Some changes from SOAP 1.1 that were made to the SOAP 1.2 specification are significant, while other changes are minor.

“Differences in SAAJ versions”
The SOAP with Attachments API for Java (SAAJ) interface Version 1.3 expands the support of SOAP 1.2 messages in a Web services environment. There are several differences between SAAJ 1.2 and SAAJ 1.3 that are presented in this topic.

Related reference

Web services specifications and APIs

Differences in SAAJ versions:

The SOAP with Attachments API for Java (SAAJ) interface Version 1.3 expands the support of SOAP 1.2 messages in a Web services environment. There are several differences between SAAJ 1.2 and SAAJ 1.3 that are presented in this topic.

In a typical Web services environment, you rely on the underlying code that is based on Java standards to translate a set of Java objects. The SAAJ interface provides APIs to read, write, send and receive SOAP messages, and advocates binary content sent as an attachment to a SOAP message.

SAAJ 1.3 aligns with SOAP 1.1 and SOAP 1.2 messages and is supported by the Java API for XML Web Services (JAX-WS) programming model and the Java API for XML-Based RPC (JAX-RPC) programming model. SAAJ 1.2 works with SOAP 1.1 messages only.

If you migrate your code from SOAP 1.1 to SOAP 1.2, you can continue to use your existing SOAP 1.1 code, if the message is a SOAP 1.2 message. If you upgrade your base code to use SAAJ 1.3, then you can continue to use the existing code that operates on a SOAP 1.1 message. An example of these differences is in SOAP 1.1, where the human readable text of a fault is stored in the faultString element. In SOAP 1.2, the human readable text is stored in the Reason element. Your code might look like the following example:

```
String text = soapFault.getFaultString();
```

The getFaultString () returns the faultString value if the message is based on SOAP 1.1. If you are using SOAP 1.2, the getFaultString () returns the Reason value. In addition, the SAAJ 1.3 interface provides a new method, getReasonText (Locale), that gets a specific Reason value. The getReasonText (Locale) method returns a documented exception if the message is based on SOAP 1.1. The SAAJ 1.3 interface supports existing code to process both SOAP 1.1 and SOAP 1.2 messages.

Other differences between SAAJ 1.2 and SAAJ 1.3 are in the following list:

- SAAJMetaFactory interface

  The SAAJMetaFactory SPI is introduced to support creating SOAP factory classes in a single place.
- SAAJResult class

  The SAAJResult object acts as a holder for the results of a Java API for XML Processing (JAX-P) transformation or a Java Architecture for XML Binding (JAXB) marshalling, in the SAAJ tree. The SAAJResult class is introduced for improved usability when transformation results are expected to be a valid SAAJ tree.
- Overloaded methods that accept a QName instead of a Name

  It is preferred that a QName represents an XML-qualified name. Therefore, overloaded methods are introduced in all of the SAAJ APIs, where a corresponding method accepts a javax.xml.soap.Name name as an argument.
- New methods in AttachmentPart, SOAPBody and SOAPElement interfaces and classes

  Use these new methods to assist you when you are working with the new SOAP features.
- SOAPPart is now a javax.xml.soap.Node method.

  The SOAPPart object is now also considered to be a SOAP node method.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Message Transmission Optimization Mechanism:

SOAP Message Transmission Optimization Mechanism (MTOM) is a standard that is developed by the World Wide Web Consortium (W3C). MTOM describes a mechanism for optimizing the transmission or wire format of a SOAP message by selectively re-encoding portions of the message while still presenting an XML Information Set (Infoset) to the SOAP application.

There are many reasons why you might want to send binary attachments, such as images or files, along with a Web services request. There are ways to accomplish this, such as:

- Encoding with base64 inline in the SOAP payload. However, encoding inline tends to enlarge the size of the SOAP message. Note that base64 encoding might double the size of the binary data.
- Encoding the messages by using SOAP with Attachments (SwA) and to follow the Web Services Interoperability Organization (WS-I) Attachments Profile. WebSphere Application Server currently supports this method.
- Providing optimization of binary message transportation by using XML-binary Optimized Packaging (XOP). Optimization is available only for binary data or content. MTOM uses XOP in the context of SOAP and MIME over HTTP.

XOP defines a serialization mechanism for the XML Infoset with binary content that is not only applicable to SOAP and MIME packaging, but to any XML Infoset and any packaging mechanism. It is an alternate serialization of XML that just happens to look like a MIME multipart/related package, with an XML documents as the root part. That root part is very similar to the XML serialization of the document, except that base64-encoded data is replaced by a reference to one of the MIME parts, which is not base64 encoded. This reference enables you to avoid the bulk and overhead in processing that is associated with encoding. Encoding is the only way a binary data can fit directly into an XML world.

If MTOM mapping generation is disabled, then XOP is disabled. If XOP is disabled, the binary data are not sent by using MIME attachments. Instead, the binary data is base64 encoded as usual.

The MTOM specification is defined in three different parts:

- An abstract feature for optimized transmission or wire format for SOAP messages. This feature is abstract in the sense that the description of the optimization technique as well as the behavior of the SOAP processors at sender, receiver and intermediaries is generic and does not include any references

to technologies such as MIME, HTTP, and so forth. The optimization technique centers around ensuring a SOAP envelope Infoset view for the SOAP processors while encoding selectively certain contents of the SOAP Envelope Infoset that are accessible as canonical lexical representation of the xs:base64Binary data type.

Implementing these abstract features requires concrete specification of two aspects: the optimized wire format and how the optimized wire format flows on a particular transport

- The second part of the MTOM specification addresses the serialization aspect and depends normatively upon MIME Multipart/Related XOP packaging. The serialization aspect is where MTOM relates to XOP.
- As a concrete SOAP HTTP binding level feature, MTOM expands upon the serialization. This part describes how HTTP binding can be used to transport the XOP packages that are holding the SOAP MTOM messages. This part also puts some restrictions on the possible serializations of the SOAP MTOM messages as XOP packages, such as use of XML 1.0 only.

The Java API for XML Web Services (JAX-WS) adds support for sending binary data attachments using MTOM. JAX-WS is the centerpiece of a newly re-architected API stack for Web service that includes JAX-WS 2.0, JAXB 2.0, and SAAJ 1.3. The API stack is sometimes referred to as the integrated stack. JAX-WS is designed to take the place of JAX-RPC in Web services and Web applications.

Attachment approach

Attachment by value or by reference has been the widely accepted technique for handling opaque data in XML-formatted messages.

- **By value** is when the opaque data content is embedded as an element or as an attribute by using either base64 or hexadecimal text encoding approach, which is codified in the XML schema as data types xs:base64Binary and xs:hexBinary, respectively.
- **By reference** is when the opaque data content is referenced externally as element or as attribute by using a URI, which is codified in the XML schema as data type xs:anyURI.

The use of either of these two techniques has its advantages and disadvantages. MTOM is the specification that is focused on resolving these inherent attachments problems.

A different standard is defined by World Wide Web Consortium (W3C) and is called SOAP with Attachments (SwA). SwA was developed as a way to package SOAP messages with attachments. Because some vendors do not support SwA, SwA can be replaced by the more powerful MTOM and XOP mechanisms. SwA and MTOM are conceptually similar, and both encode binary data as a MIME attachment in a MIME document. Using MIME attachments improves the performance of large binary payloads transport.

Additional differences between SwA and MTOM include:

- MTOM uses a standard called XOP, which defines a XOP reference that exists within the SOAP payload. This reference points to the MIME attachment that contains the binary data.
- With MTOM, the XOP reference logically includes the binary data into the XML Information Set (Infoset). With SwA, the href points to data that is not only physically outside the XML document but is not logically included within its Infoset.
- With MTOM, binary attachments can be logically signed as if they were part of the SOAP XML document.
- In additional to IBM, Microsoft .NET supports MTOM, which eliminates some of the interoperability problems found with SwA. Interoperability was treated as the main goal when the co-submitters discussed the suggested modifications.

The MTOM attachment approach takes advantage of the SOAP infrastructure while also gaining transport efficiencies that are provided by SOAP with Attachment (SwA) solution.

SOAP 1.2 and SOAP 1.1

SOAP 1.1 is based on the XML specification. Likely, the SOAP 1.1 implementation will continue to exist for a few years. For those who are still running SOAP 1.1, there is now an interoperable way to use MTOM for attachments support. SAP, Oracle, Microsoft, and IBM have submitted a SOAP 1.1 Binding for MTOM 1.0 specification to W3C, which defines how MTOM can be used with SOAP 1.1 payloads. The specification details the necessary modifications to the SOAP MTOM and XOP specifications that are necessary to successfully use these technologies with SOAP 1.1. See the information at this Web site:http://schemas.xmlsoap.org/soap/mtom/SOAP11MTOM10.pdf

MTOM is a SOAP Version 1.2 feature, which is based on the Infoset. For more information, see “XML information set” on page 438.

Without MTOM, the data is encoded in whatever format is described in the schema (base64 or hex) and then is contained in the XML document. The following example shows a SOAP message with an <xsd:base64Binary> element:

```
... other transport headers ...
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Header/>
      <soapenv:Body>
         <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
            <input>
               <imageData>R0lGODl  ... more base64 encoded data ...  KTJk8giAAA7</imageData>
            </input>
         </sendImage>
      </soapenv:Body>
   </soapenv:Envelope>
```

When MTOM is enabled, the binary data that represents the attachment is included as a MIME attachment to the SOAP message. The following example shows an MTOM-enabled SOAP message with attachment data:

```
... other transport headers ...
Content-Type: multipart/related; boundary=MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812;
type="application/xop+xml"; start="<0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>";
start-info="text/xml"; charset=UTF-8

--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: application/xop+xml; charset=UTF-8; type="text/xml";
content-transfer-encoding: binary
content-id:
   <0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>

<?xml version="1.0" encoding="UTF-8"?>
         <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
            <soapenv:Header/>
            <soapenv:Body>
               <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
                  <input>
                     <imageData>
                        <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
                             href="cid:1.urn:uuid:0FE43E4D025F0BF3DC11582467646811@apache.org"/>
                     </imageData>
                  </input>
               </sendImage>
            </soapenv:Body>
         </soapenv:Envelope>
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: text/plain
content-transfer-encoding: binary
```

```
content-id:
         <1.urn:uuid:0FE43E4D025F0BF3DC115824676468l1@apache.org>

... binary data goes here ...
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812--
```

XML-binary Optimized Packaging:

XML-binary Optimized Packaging (XOP) specification was standardized by the World Wide Web (W3C) on January 25, 2005. SOAP Message Transmission Optimization Mechanism (MTOM) uses XOP in the context of SOAP and MIME over HTTP.

XML is widely used for data transfer. XML is a popular format for exchanging well-formed documents because it is plain text, human-readable, and structured. For example, SOAP messaging in Web services is based on XML (or is based on XML Infoset with SOAP 1.2). People want to leverage legacy formats like PDF, GIF, JPEG and similar things, while still using an XML model. The desire to integrate XML with pre-existing data formats has been a long-standing and persistent issue for the XML community. Users often want to leverage the structured, extensible markup conventions of XML without abandoning existing data formats that do not readily adhere to XML 1.0 syntax.

As SOAP messaging in Web services becomes more widespread, the next step is how to send non-text based data, such as images and workflow data, along with your message. For example, you might want to send a scanned document in .jpeg format between two applications. The question becomes whether this data can be understood between the various applications.

Much of the value of XML and Web services resides in the ability to use generic XML tools to work with content. Many XML tools and standards for describing and manipulating XML (such as parsers, XPath, XQuery, XSLT, XML encryption and digital signature and XML schema) are not designed to work with non-text data, such as images. These XML tools do not work with non-XML content; these tools require text. The challenge is how non-text data (also called *binary data*) can be embedded or attached with XML. In other words, a way to attach a binary file to a SOAP message is needed.

Encoding is the only way binary data can fit directly into an XML world. Normally, you can embed binary data in an XML document by encoding it as text using Base 64. Base 64 is a serialization that has existed for some time, can be easily implemented out of the box, and has interoperability across platforms. The xsi:base64binary datatype supports this serialization in the XML Schema. Base 64 encodes your binary data into a textual representation that can squeeze into an XML document. Base 64 takes your binary data and translates it into a series of ASCII characters by encoding three octets at a time. Because each octet consists of eight bits, representing them as four printable characters in the ASCII standard, it uses 64 ASCII characters to represent the binary. All platforms can decode and encode using this convention. 6-bit ASCII is widely supported, and no special characters need to be dealt with. However, there is a performance impact for larger messages.

For applications that require speedy operation, Base 64 might not be the solution. If you want to index into such content, query it, transform it, encrypt it, sign it, or describe it, you need to use a different mechanism.

The first attachment specification known as SOAP with Attachments (SwA) was developed. The basic idea of SwA is that the binary message part (the attachment) is thought of as a Multipurpose Internet Mail Extensions (MIME) attachment. MIME is a widely implemented specification for formatting non-ASCII mail message attachments. SwA specifies that the SOAP body can contain a reference to the MIME message part (the attachment) simply by using a URI. The binary part is attached by a reference.

A few disadvantages of SwA include:

- SwA fails in its usability or interoperability. The SOAP infrastructure was created around the SOAP envelope, which didn't work well for attachments. An attachment using SwA means that two data models are used in one message. These two data models do not operate with existing XML technology.

- SwA does not work with the composable character of SOAP. Basically standards, such as WS-Security, were not written to work with attachments. WS-Security needs to work on all the data that needs to be digitally signed or encrypted, and that means all the data in the attachment also. But if it cannot access it, then it will not work and the signature is effectively invalid.

Often, users want to leave their existing non-XML formats as is, to be treated as opaque sequences of octets by XML tools and infrastructure. Such an approach permits widely used formats such as .jpeg and .wav to peacefully coexist with XML. XOP makes it a bit more realistic to use base64-encoded data. At the current time, XOP only permits base64-encoded data to be optimized.

Using XOP means that instances of XML-type base64Binary, if enabled, are transported by using MIME attachments. If XOP is in use, the implementation can automatically encode it. XOP maintains the data model of the XML message because the attachment is treated as base64-encoded data. If an XML stack understands XOP encoding, your application does not need to be changed at all. For example, when it wants to access a .jpeg picture, it can get the character value of the content as a base64-encoded string.

XOP gives people a way to think about MIME messages in a data exchange that they are comfortable with and already use for a lot of other data. The XOP format uses multipart MIME to enable raw binary data to be included into an XML 1.0 document without resorting to base64 encoding.

A companion specification, SOAP Message Transmission Optimization Method (MTOM) then specifies how to bind this format to SOAP. The XOP and MTOM standards should enhance SOAP 1.2 performance. XOP and MTOM together provide the preferred approach for mixing binary data with text-based XML. Coupled together, MTOM and XOP enables us to select what parts of the message need to be sent over the wire as binary while still maintaining the Infoset. These standards enable the attachment of binary data outside of the SOAP envelope as a message part. However, unlike SwA, the binary data is treated very much as it was within the SOAP envelope, meaning one Infoset.

XOP defines a serialization mechanism for XML Infoset with binary content that is not only applicable to SOAP and MIME packaging, but applicable to any XML Infoset and any packaging mechanism. On the other hand, XML is not a good general-purpose packaging mechanism.

An XOP package is created by placing a serialization of the XML Infoset inside of an extensible packaging format (such a MIME). Note that XOP does reuse MIME for the actual packaging on the wire. Then, selected portions of its content that are base64-encoded binary data are extracted and re-encoded, meaning the data is decoded from base64 and placed into the package. The locations of those selected portions are marked in the XML with a special element that links to the packaged data by using URIs.

The SOAP processing engines performs a temporary Base 64 encoding of the binary data just before the message hits the wire. This temporary encoding enables the SOAP processor to work on the Base 64 data; for example, enabling a WS-Signature of the data to be taken and placed into the header. There is no need for expensive decoding at the other end, and the process works in reverse.

Implementations of MTOM and XOP are available in Java (JAX-WS).

This example shows an XML Infoset prior to XOP processing (SOAP):

```
<soap:Envelope
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
    xmlns:xmlmime='http://www.w3.org/2004/11/xmlmime'>
  <soap:Body>
    <m:data xmlns:m='http://example.org/stuff'>
      <m:photo
  xmlmime:contentType='image/png'>/aWKKapGGyQ=</m:photo>
      <m:sig
  xmlmime:contentType='application/pkcs7-signature'>Faa7vROi2VQ=</m:sig>
    </m:data>
  </soap:Body>
</soap:Envelope>
```

This example shows an XML Infoset that is serialized as a XOP package (SOAP)

```
MIME-Version: 1.0
Content-Type: Multipart/Related;boundary=MIME_boundary;
    type="application/xop+xml";
    start="<mymessage.xml@example.org>";
    startinfo="application/soap+xml; action=\"ProcessData\""
Content-Description: A SOAP message with my pic and sig in it

--MIME_boundary
Content-Type: application/xop+xml;
    charset=UTF-8;
    type="application/soap+xml; action=\"ProcessData\""
Content-Transfer-Encoding: 8bit
Content-ID: <mymessage.xml@example.org>

<soap:Envelope
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
    xmlns:xmlmime='http://www.w3.org/2004/11/xmlmime'>
  <soap:Body>
    <m:data xmlns:m='http://example.org/stuff'>
      <m:photo
  xmlmime:contentType='image/png'><xop:Include
    xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:http://example.org/me.png'/></m:photo>
      <m:sig
  xmlmime:contentType='application/pkcs7-signature'><xop:Include
    xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:http://example.org/my.hsh'/></m:sig>
    </m:data>
  </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/png
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/me.png>

// binary octets for png

--MIME_boundary
Content-Type: application/pkcs7-signature
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/my.hsh>

// binary octets for signature

--MIME_boundary--
```

XML information set:

XML Information Set (Infoset) is a World Wide Web Consortium (W3C) specification, dated February 4, 2004. An XML information set is an abstract model of the information that is stored in an XML document. The information set establishes a separation between data and information in a way that suits most common uses of XML. Several of the concrete XML data models are defined by referring to XML information set items and their properties.

Whereas an *XML information set* is an abstract model of the information that is stored in an XML document, an *information item* is an abstract representation of some component of an XML document. SOAP Version 1.2 makes use of this abstraction to define the information in a SOAP message without ever referring to XML Version 1.*x*. The SOAP HTTP binding specifically permits alternative media types that provide for, as a minimum, the transfer of the SOAP XML Infoset.

SOAP Message Transmission Optimization Mechanism (MTOM) describes SOAP 1.2 constructs in terms of information items whereas SOAP 1.1 is defined in terms of XML elements. MTOM enables SOAP bindings to optimize the transmission or wire format (or both) of a SOAP message by selectively encoding portions of the message while still presenting an XML information set to the SOAP application. The SOAP 1.2 attribute is now in the SOAP namespace. The XML information sets require the support of XML namespaces. The core XML recommendation does not require the support of XML namespaces; however namespaces are required to support the XML schema.

The XML information set does not require or favor a specific interface or class of interfaces. The XML information set specification presents the information set as a tree for the sake of clarity and simplicity, but there is no requirement that the XML information set be made available through a tree structure. Other types of interfaces, including but not limited to event-based and query-based interfaces, are also capable of providing information conforming to the information set. As long as the information in the information set is made available to XML applications in one way or another, the requirements of the XML information set are satisfied.

The XML information set provides a set of definitions to be used in other specifications that refer to the information in a well-formed XML document. For any given XML document, there are a number of corresponding information sets.

- A unique minimal information set consisting of the core properties of the core items and nothing else.
- A unique maximal information set consisting of all the core and all the peripheral items with all the peripheral properties, and one for every combination of present and absent peripheral items and properties in between. The in-between information sets must be fully consistent with the maximal information set.

Information set items

The XML information set is a description of the information that is available in a well-formed XML document, and it describes an abstract data model of an XML document in terms of a set of information set items. An information item is an abstract description of some part of an XML document, and each information item has a set of associated named properties. All other information items are accessible from the properties of the document information item, either directly or indirectly through the properties of other information items.

Guidelines for using information set items include:

- There is no requirement for an XML document to be valid in order to have an information set.
- An XML document has an information set if it satisfies the namespace constraints.
- An XML document has an information set if it is well-formed
- Only one document information item is permited in the information set.
- An information set for an XML document consists of two or more information items.
- The information set for any well-formed XML document will contain at least the minimum information items: one document information item and one element information item.
- Each information item has a set of associated properties, some of which are core and some of which are peripheral.

An information set can contain up to eleven different types of information items:

- Document information item
- Element information items
- Attribute information items
- Processing instruction information items
- Unexpanded entity reference information items
- Character information items
- Comment information items
- The Document Type Declaration (DTD) information item

- Unparsed entity information items
- Notation information items
- Namespace information items

Note that the information set of the XML document might not be a complete list of all information items.

Certain kinds of invalidity affect the values assigned to some properties. Entities, notations, elements and attributes can be undeclared. You can have multiple declarations for notations and elements. Multiple declarations are valid for entities and attributes. An ID can be undefined or multiply defined. Such cases are noted where relevant in the information item definitions in the XML Information Set specification.

Syntax

The XML information set uses a square-bracket syntax, meaning the property names are shown in square brackets. For example, the document information item has the following properties:

Table 10.

Property	Description
[children]	An ordered list of child information items, in document order.
[document element]	The element information item corresponding to the document element.
[notations]	An unordered set of notation information items, one for each notation declared in the DTD. If any notation is multiply declared, this property has no value.
[unparsed entities]	An unordered set of unparsed entity information items, one for each unparsed entity declared in the DTD.
[base URI]	The base URI of the document entity.
[character encoding scheme]	The name of the character encoding scheme in which the document entity is expressed.
[standalone]	An indication of the standalone status of the document, either yes or no. This property is derived from the optional standalone document declaration in the XML declaration at the beginning of the document entity, and has no value if there is no standalone document declaration.
[version]	A string representing the XML version of the document. This property is derived from the XML declaration optionally present at the beginning of the document entity, and has no value if there is no XML declaration.
[all declarations processed]	This property is not strictly speaking part of the information set of the document. Rather it is an indication of whether the processor has read the complete DTD. Its value is a boolean. If it is false, then certain properties (indicated in their descriptions below) might be unknown. If it is true, those properties are never unknown.

All information sets are understood to describe the XML document with all entity references already expanded; that is, represented by the information items corresponding to their replacement text. In the case that an entity reference cannot be expanded, because an XML processor has not read its declaration or its value, explicit provision is made for representing such a reference in the information set.

Differences in SOAP versions:

Both SOAP Version 1.1 and SOAP Version 1.2 are World Wide Web Consortium (W3C) standards. Web services can be deployed that support not only SOAP 1.1 but also support SOAP 1.2. Some changes from SOAP 1.1 that were made to the SOAP 1.2 specification are significant, while other changes are minor.

The SOAP 1.2 specification introduces several changes to SOAP 1.1. This information is not intended to be an in-depth description of all the new or changed features for SOAP 1.1 and SOAP 1.2. Instead, this information highlights some of the more important differences between the current versions of SOAP.

The changes to the SOAP 1.2 specification that are significant include the following updates:

- SOAP 1.1 is based on XML 1.0. SOAP 1.2 is based on XML Information Set (XML Infoset).

  The XML information set (infoset) provides a way to describe the XML document with XSD schema. However, the infoset does not necessarily serialize the document with XML 1.0 serialization on which SOAP 1.1 is based.. This new way to describe the XML document helps reveal other serialization formats, such as a binary protocol format. You can use the binary protocol format to compact the message into a compact format, where some of the verbose tagging information might not be required.

  In SOAP 1.2 , you can use the specification of a binding to an underlying protocol to determine which XML serialization is used in the underlying protocol data units. The HTTP binding that is specified in SOAP 1.2 - Part 2 uses XML 1.0 as the serialization of the SOAP message infoset.
- SOAP 1.2 provides the ability to officially define transport protocols, other than using HTTP, as long as the vendor conforms to the binding framework that is defined in SOAP 1.2. While HTTP is ubiquitous, it is not as reliable as other transports including TCP/IP and MQ.
- SOAP 1.2 provides a more specific definition of the SOAP processing model that removes many of the ambiguities that might lead to interoperability errors in the absence of the Web Services-Interoperability (WS-I) profiles. The goal is to significantly reduce the chances of interoperability issues between different vendors that use SOAP 1.2 implementations.
- SOAP with Attachments API for Java (SAAJ) can also stand alone as a simple mechanism to issue SOAP requests. A major change to the SAAJ specification is the ability to represent SOAP 1.1 messages and the additional SOAP 1.2 formatted messages. For example, SAAJ Version 1.3 introduces a new set of constants and methods that are more conducive to SOAP 1.2 (such as getRole(), getRelay()) on SOAP header elements. There are also additional methods on the factories for SAAJ to create appropriate SOAP 1.1 or SOAP 1.2 messages.
- The XML namespaces for the envelope and encoding schemas have changed for SOAP 1.2. These changes distinguish SOAP processors from SOAP 1.1 and SOAP 1.2 messages and supports changes in the SOAP schema, without affecting existing implementations.
- Java Architecture for XML Web Services (JAX-WS) introduces the ability to support both SOAP 1.1 and SOAP 1.2. Because JAX-RPC introduced a requirement to manipulate a SOAP message as it traversed through the run time, there became a need to represent this message in its appropriate SOAP context. In JAX-WS, a number of additional enhancements result from the support for SAAJ 1.3.
- The Web Services Description Language (WSDL) Version 1.1 specification does not discuss SOAP 1.2. SOAP 1.2 is discussed in the draft versions of WSDL 2.0. WSDL Version 1.1 only defines how to render a SOAP 1.1 payload in a WSDL 1.1 document. To resolve how to represent SOAP 1.2-based services, there is another W3C document that defines how to define a SOAP 1.2 payload within a WSDL 1.1 document. Read about WSDL 1.1 binding extensions for SOAP 1.2.
- SOAP 1.1 is a single document. The SOAP 1.2 specification is organized in the following parts:
  - Part 0 is a non-normative introduction to SOAP.
  - Part 1 describes the structure of SOAP messages, the SOAP processing model and a framework for binding SOAP to underlying protocols. Conformant SOAP implementations must implement everything in Part 1.
  - Part 2 describes optional add-ins to the core of SOAP including a data model and encoding, an RPC convention and a binding to HTTP. Conformant SOAP implementations might implement any of the add-ins in Part 2. However, if add-ins are implemented, they must conform to the relevant parts of the specification.

  A fourth document is the Specification Assertions and Test Collection

SOAP 1.2 has a number of changes in syntax and provides additional, clarified semantics from those that are described in SOAP 1.1. The SOAP 1.2 Primer document lists and describes these syntax changes.

JAX-WS

Java API for XML-Based Web Services (JAX-WS) is the next generation Web services programming model complimenting the foundation provided by the Java API for XML-based RPC (JAX-RPC)

programming model. Using JAX-WS, development of Web services and clients is simplified with more platform independence for Java applications by the use of dynamic proxies and Java annotations.

JAX-WS is a programming model that simplifies application development through support of a standard, annotation-based model to develop Web Service applications and clients. The JAX-WS technology strategically aligns itself with the current industry trend towards a more document-centric messaging model and replaces the remote procedure call programming model as defined by JAX-RPC. While the JAX-RPC programming model and applications are still supported by this product, JAX-RPC has limitations and does not support various complex document-centric services. JAX-WS is the strategic programming model for developing Web services and is a required part of the Java Platform, Enterprise Edition 5 (Java EE 5). JAX-WS is also known as JSR 224.

Note: WebSphere Application Server Version 7.0 supports the JAX-WS 2.1 specification. JAX-WS 2.1 extends the functionality of JAX-WS 2.0 to provide support for the WS-Addressing in a standardized API. Using this function, you can create, transmit and use endpoint references to target a Web service endpoint. You can use this API to specify the action uniform resource identifiers (URIs) that are associated with the Web Services Description Language (WSDL) operations of your Web service. JAX-WS 2.1 introduces the concept of *features* as a way to programmatically control specific functions and behaviors. There are three standard features: the AddressingFeature for WS-Addressing, the MTOMFeature when optimizing the transmission of binary attachments, and the RespectBindingFeature for wsdl:binding extensions. JAX-WS 2.1 requires Java Architecture for XML Binding (JAXB) Version 2.1 for data binding.

The implementation of the JAX-WS programming standard provides the following enhancements for developing Web services and clients:

- **Better platform independence for Java applications.**

  Using JAX-WS APIs, development of Web services and clients is simplified with better platform independence for Java applications. JAX-WS takes advantage of the dynamic proxy mechanism to provide a formal delegation model with a pluggable provider. This is an enhancement over JAX-RPC, which relies on the generation of vendor-specific stubs for invocation.

- **Annotations**

  JAX-WS introduces support for annotating Java classes with metadata to indicate that the Java class is a Web service. JAX-WS supports the use of annotations based on the Metadata Facility for the Java Programming Language (JSR 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.1 specification. Using annotations within the Java source and within the Java class simplifies development of Web services. Use annotations to define information that is typically specified in deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL files into the source artifacts.

  For example, you can embed a simple @WebService tag in the Java source to expose the bean as a Web service.

  ```
  @WebService

  public class QuoteBean implements StockQuote {

         public float getQuote(String sym) { ... }

  }
  ```

  The @WebService annotation tells the server runtime environment to expose all public methods on that bean as a Web service. Additional levels of granularity can be controlled by adding additional annotations on individual methods or parameters. Using annotations makes it much easier to expose Java artifacts as Web services. In addition, as artifacts are created from using some of the top-down mapping tools starting from a WSDL file, annotations are included within the source and Java classes as a way of capturing the metadata along with the source files.

Using annotations also improves the development of Web services within a team structure because you do not need to define every Web service in a single or common deployment descriptor as required with JAX-RPC Web services. Taking advantage of annotations with JAX-WS Web services enables parallel development of the service and the required metadata.

For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

For example, if your service implementation class for your JAX-WS Web service includes the following:

– the @WebService annotation:

```
@WebService(wsdlLocation="http://myhost.com/location/of/the/wsdl/ExampleService.wsdl")
```

– the webservices.xml file specifies a different file name for the WSDL document as follows:

```
<webservices>
<webservice-description>
<webservice-description-name>ExampleService</webservice-description-name>
<wsdl-file>META-INF/wsdl/ExampleService.wsdl</wsdl-file>
...
</webservice-description>
</webservices>
```

In this case, the value that is specified in the deployment descriptor, META-INF/wsdl/ExampleService.wsdl overrides the annotation value.

- **Invoking Web services asynchronously**

  With JAX-WS, Web services are called both synchronously and asynchronously. JAX-WS adds support for both a polling and callback mechanism when calling Web services asynchronously. Using a polling model, a client can issue a request, get a response object back, which is polled to determine if the server has responded. When the server responds, the actual response is retrieved. Using the callback model, the client provides a callback handler to accept and process the inbound response object. Both the polling and callback models enable the client to focus on continuing to process work without waiting for a response to return, while providing for a more dynamic and efficient model to invoke Web services.

  For example, a Web service interface might have methods for both synchronous and asynchronous requests. Asynchronous requests are identified in bold in the following example:

```
@WebService
public interface CreditRatingService {
      // sync operation
      Score      getCreditScore(Customer customer);
      // async operation with polling
      Response<Score> getCreditScoreAsync(Customer customer);
      // async operation with callback
      Future<?> getCreditScoreAsync(Customer customer,
         AsyncHandler<Score> handler);
}
```

  The asynchronous invocation that uses the callback mechanism requires an additional input by the client programmer. The callback is an object that contains the application code that is run when an asynchronous response is received. Use the following code example to invoke an asynchronous callback handler:

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerFred,
 new AsyncHandler<Score>() {
    public void handleResponse (
        Response<Score> response)
      {
        Score score = response.get();
```

```
        // do work here...
      }
    }
);
```

Use the following code example to invoke an asynchronous polling client:

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerFred);

while (!response.isDone()) {
  // Complete an action while we wait.
}

// No cast needed, because of generics.
Score score = response.get();
```

- **Using resource injection**

  JAX-WS supports resource injection to further simplify development of Web services. JAX-WS uses this key feature of Java EE 5 to shift the burden of creating and initializing common resources in a Java runtime environment from your Web service application to the application container environment, itself. JAX-WS provides support for a subset of annotations that are defined in JSR-250 for resource injection and application life cycle in its runtime environment.

  The application server also supports the usage of the @Resource or @WebServiceRef annotation to declare JAX-WS managed clients and to request injection of JAX-WS services and ports. When either of these annotations are used on a field or method, they result in injection of a JAX-WS service or port instance. The usage of these annotations also results in the type specified by the annotation being bound into the JNDI namespace.

  The @Resource annotation is defined by the JSR-250, Common Annotations specification that is included in Java Platform, Enterprise Edition 5 (Java EE 5). By placing the @Resource annotation on a variable of type javax.xml.ws.WebServiceContext within a service endpoint implementation class, you can request a resource injection and collect the javax.xml.ws.WebServiceContext interface related to that particular endpoint invocation. From the WebServiceContext interface, you can collect the MessageContext for the request associated with the particular method call using the getMessageContext() method.

  The @WebServiceRef annotation is defined by the JAX-WS specification.

  The following example illustrates using the @Resource and @WebServiceRef annotations for resource injection:

```
@WebService
public class MyService {

    @Resource
    private WebServiceContext ctx;

    @Resource
    private SampleService svc;

    @WebServiceRef
    private SamplePort port;

    public String echo (String input) {
        ...
    }

}
```

  Refer to sections 5.2.1 and 5.3 of the JAX-WS specification for more information on resource injection.

- **Data binding with JAXB 2.1**

  JAX-WS leverages the Java Architecture for XML Binding (JAXB) 2.1 API and tools as the binding technology for mappings between Java objects and XML documents. JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents. JAXB data binding replaces the data binding described by the JAX-RPC specification.

The JAXB 2.1 specification provides enhancements such as improved compilation support and introduces support for the @XMLSeeAlso annotation. With the improved compilation support, you now have the flexibility to control the whether a new schema file is generated when using the schemagen schema generator and you can configure the xjc schema compiler so that it does not automatically generate new classes for a particular schema. You can use the @XMLSeeAlso annotation to ensure that JAXB is aware of all the classes included in an inheritance hierarchy for a service endpoint interface.**Dynamic and static clients**

The dynamic client API for JAX-WS is called the dispatch client (javax.xml.ws.Dispatch). The dispatch client is an XML messaging oriented client. The data is sent in either PAYLOAD or MESSAGE mode. When using the PAYLOAD mode, the dispatch client is only responsible for providing the contents of the <soap:Body> and JAX-WS adds the <soap:Envelope> and <soap:Header> elements. When using the MESSAGE mode, the dispatch client is responsible for providing the entire SOAP envelope including the <soap:Envelope>, <soap:Header>, and <soap:Body> elements. JAX-WS does not add anything additional to the message. The dispatch client supports asynchronous invocations using a callback or polling mechanism.

The static client programming model for JAX-WS is the called the proxy client. The proxy client invokes a Web service based on a Service Endpoint interface (SEI), which must be provided.

- **Support for MTOM**

  Using JAX-WS, you can send binary attachments such as images or files along with Web services requests. JAX-WS adds support for optimized transmission of binary data as specified by Message Transmission Optimization Mechanism (MTOM).

- **Multiple data binding technologies**

  JAX-WS exposes the following binding technologies to the end user: XML Source, SOAP Attachments API for Java (SAAJ) 1.3, and Java Architecture for XML Binding (JAXB) 2.1. XML Source enables a user to pass a javax.xml.transform.Source into the runtime environment which represents the data in a Source object to be processed. SAAJ 1.3 now has the ability to pass an entire SOAP document across the interface rather than just the payload itself. This action is done by the client passing the SAAJ SOAPMessage object across the interface. JAX-WS leverages the JAXB 2.1 support as the data binding technology of choice between Java and XML.

- **Support for SOAP 1.2**

  Support for SOAP 1.2 has been added to JAX-WS 2.0. JAX-WS supports both SOAP 1.1 and SOAP 1.2 so that you can send binary attachments such as images or files along with Web services requests. JAX-WS adds support for optimized transmission of binary data as specified by MTOM.

- **Development tools**

  JAX-WS provides the wsgen and wsimport command-line tools for generating portable artifacts for JAX-WS Web services. When creating JAX-WS Web services, you can start with either a WSDL file or an implementation bean class. If you start with an implementation bean class, use the wsgen command-line tool to generate all the Web services server artifacts, including a WSDL file if requested. If you start with a WSDL file, use the wsimport command-line tool to generate all the Web services artifacts for either the server or the client. The wsimport command-line tool processes the WSDL file with schema definitions to generate the portable artifacts, which include the service class, the service endpoint interface class, and the JAXB 2.1 classes for the corresponding XML schema.

- **Support for WS-Addressing (JAX-WS 2.1)**

  JAX-WS 2.1 integrates support for the WS-Addressing standard in to the API. The new API enables you to create, transmit and use endpoint references to target a specific Web service endpoint. You can also explicitly specify the action URIs associated with the Web Services Description Language (WSDL) operations of your Web service.

- **Support for JAX-WS 2.1 features**

  JAX-WS 2.1 introduces the concept of features as a way to programmatically control certain functions or behaviors. There are three standard features: the AddressingFeature , the MTOMFeature, and the RespectBindingFeature. The AddressingFeature is used to enable or disable support for the WS-Addressing Version 1.0 specification. The MTOMFeature is used to enable or disable support for

Message Transmission Optimized Mechanism (MTOM) when sending binary attachments. The RespectBindingFeature is used to enable or disable support for wsdl:binding extensions. The application server supports an additional feature, the SubmissionAddressingFeature, which is used to enable or disable support for WS-Addressing Member Submission specification (prior to the WS-Addressing Version 1.0 level of specification).

JAX-WS client programming model:

The Java API for XML-Based Web Services (JAX-WS) Web service client programming model supports both the Dispatch client API and the Dynamic Proxy client API. The Dispatch client API is a dynamic client programming model, whereas the static client programming model for JAX-WS is the Dynamic Proxy client. The Dispatch and Dynamic Proxy clients enable both synchronous and asynchronous invocation of JAX-WS Web services.

- Dispatch client: Use this client when you want to work at the XML message level or when you want to work without any generated artifacts at the JAX-WS level.
- Dynamic Proxy client: Use this client when you want to invoke a Web service based on a service endpoint interface.

Dispatch client

XML-based Web services use XML messages for communications between Web services and Web services clients. The JAX-WS APIs provide high-level methods to simplify and hide the details of converting between Java method invocations and their associated XML messages. However, in some cases, you might desire to work at the XML message level. Support for invoking services at the XML message level is provided by the Dispatch client API. The Dispatch client API, `javax.xml.ws.Dispatch`, is a dynamic JAX-WS client programming interface. To write a Dispatch client, you must have expertise with the Dispatch client APIs, the supported object types, and knowledge of the message representations for the associated Web Services Description Language (WSDL) file. The Dispatch client can send data in either `MESSAGE` or `PAYLOAD` mode. When using the `javax.xml.ws.Service.Mode.MESSAGE` mode, the Dispatch client is responsible for providing the entire SOAP envelope including the `<soap:Envelope>`, `<soap:Header>`, and `<soap:Body>` elements. When using the `javax.xml.ws.Service.Mode.PAYLOAD` mode, the Dispatch client is only responsible for providing the contents of the `<soap:Body>` and JAX-WS includes the payload in a `<soap:Envelope>` element.

The Dispatch client API requires application clients to construct messages or payloads as XML which requires a detailed knowledge of the message or message payload. The Dispatch client supports the following types of objects:

- `javax.xml.transform.Source`: Use `Source` objects to enable clients to use XML APIs directly. You can use `Source` objects with SOAP or HTTP bindings.
- JAXB objects: Use JAXB objects so that clients can use JAXB objects that are generated from an XML schema to create and manipulate XML with JAX-WS applications. JAXB objects can only be used with SOAP or HTTP bindings.
- `javax.xml.soap.SOAPMessage`: Use `SOAPMessage` objects so that clients can work with SOAP messages. You can only use `SOAPMessage` objects with SOAP bindings.
- `javax.activation.DataSource`: Use `DataSource` objects so that clients can work with Multipurpose Internet Mail Extension (MIME) messages. Use `DataSource` only with HTTP bindings.

For example, if the input parameter type is `javax.xml.transform.Source`, the call to the Dispatch client API is similar to the following code example:

```
Dispatch<Source> dispatch = ... create a Dispatch<Source>
Source request = ... create a Source object
Source response = dispatch.invoke(request);
```

The Dispatch parameter value determines the return type of the `invoke()` method.

The Dispatch client is invoked in one of three ways:

- Synchronous invocation for requests and responses using the `invoke` method
- Asynchronous invocation for requests and responses using the `invokeAsync` method with a callback or polling object
- One-way invocation using the `invokeOneWay` methods

Refer to Chapter 4, section 3 of the JAX-WS specification for more information on using a Dispatch client.

Dynamic Proxy client

The static client programming model for JAX-WS is the called the Dynamic Proxy client. The Dynamic Proxy client invokes a Web service based on a Service Endpoint Interface (SEI) which must be provided. The Dynamic Proxy client is similar to the stub client in the Java API for XML-based RPC (JAX-RPC) programming model. Although the JAX-WS Dynamic Proxy client and the JAX-RPC stub client are both based on the Service Endpoint Interface (SEI) that is generated from a WSDL file , there is a major difference. The Dynamic Proxy client is dynamically generated at run time using the Java 5 Dynamic Proxy functionality, while the JAX-RPC-based stub client is a non-portable Java file that is generated by tooling. Unlike the JAX-RPC stub clients, the Dynamic Proxy client does not require you to regenerate a stub prior to running the client on an application server for a different vendor because the generated interface does not require the specific vendor information.

The Dynamic Proxy instances extend the `java.lang.reflect.Proxy` class and leverage the Dynamic Proxy function in the base Java SE Runtime Environment (JRE) 6. The client application can then provide an interface that is used to create the proxy instance while the runtime is responsible for dynamically creating a Java object that represents the SEI.

The Dynamic Proxy client is invoked in one of three ways:

- Synchronous invocation for requests and responses using the `invoke` method
- Asynchronous invocation for requests and responses using the `invokeAsync` method with a callback or polling object
- One-way invocation using the `invokeOneWay` methods

Refer to Chapter 4 of the JAX-WS specification for more information on using Dynamic Proxy clients.

Related concepts

"JAX-WS" on page 441
Java API for XML-Based Web Services (JAX-WS) is the next generation Web services programming model complimenting the foundation provided by the Java API for XML-based RPC (JAX-RPC) programming model. Using JAX-WS, development of Web services and clients is simplified with more platform independence for Java applications by the use of dynamic proxies and Java annotations.

Related tasks

"Developing and deploying JAX-WS Web services clients" on page 530
You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-Based Web Services (JAX-WS) programming model.

"Developing a JAX-WS client from a WSDL file" on page 534
Java API for XML-Based Web Services (JAX-WS) tooling supports generating Java artifacts you need to develop static JAX-WS Web services clients when starting with a Web Services Description Language (WSDL) file.

"Invoking JAX-WS Web services asynchronously" on page 540
Java API for XML-Based Web Services (JAX-WS) provides support for invoking Web services using an asynchronous client invocation. JAX-WS provides support for both a callback and polling model when calling Web services asynchronously. Both the callback model and the polling model are available on the Dispatch client and the Dynamic Proxy client.

Related reference

Web services specifications and APIs

Related information

Java API for XML Web Services (JAX-WS) API documentation

Java API for XML Web Services (JAX-WS) API User's Guide documentation

JAX-WS annotations:

Java API for XML-Based Web Services (JAX-WS) relies on the use of annotations to specify metadata associated with Web services implementations and to simplify the development of Web services. Annotations describe how a server-side service implementation is accessed as a Web service or how a client-side Java class accesses Web services.

The JAX-WS programming standard introduces support for annotating Java classes with metadata that is used to define a service endpoint application as a Web service and how a client can access the Web service. JAX-WS supports the use of annotations based on the Metadata Facility for the Java Programming Language (Java Specification Request (JSR) 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.0 and later (JSR 224) specification which includes JAXB annotations. Using annotations from the JSR 181 standard, you can simply annotate the service implementation class or the service interface and now the application is enabled as a Web service. Using annotations within the Java source simplifies development and deployment of Web services by defining some of the additional information that is typically obtained from deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL into the source artifacts.

Use annotations to configure bindings, handler chains, set names of portType, service and other WSDL parameters. Annotations are used in mapping Java to WSDL and schema, and at runtime to control how the JAX-WS runtime processes and responds to Web service invocations.

For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre- Java Platform, Enterprise Edition (Java EE) 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default behavior is to not scan pre-Java EE 5 modules for annotations during application installation or server startup. You can preserve backward compatibility with the feature pack in either of two ways:

- You can set the `UseWSFEP61ScanPolicy` property in the META-INF/MANIFEST.MF of a WAR file or EJB module to `true`. For example:

  ```
  Manifest-Version: 1.0
  UseWSFEP61ScanPolicy: true
  ```

  When this property is set to `true` in the module's META-INF/MANIFEST.MF file, the module is scanned for JAX-WS annotations regardless of the Java EE version of the module. The default value is `false` and when the default value is in effect, JAX-WS annotations are only supported in modules whose version is Java EE 5 or later.
- You can set the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` custom Java virtual machine (JVM) property using the administrative console. See the JVM custom properties documentation for the correct navigation path to use. To request annotation scanning in all modules regardless of their Java EE version, set the custom property `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` to `true`. You must change the setting on each server that requires a change in the default behavior.

If the property is set within a module's META-INF/MANIFEST.MF file, this setting takes precedence over the server's custom JVM property. When using either property, you must establish the desired annotation scanning behavior before the application is installed. You cannot dynamically change the scanning behavior once an application is installed. If changes to the behavior are required after your application is installed, you must first uninstall the application, specify the desired scanning behavior using the appropriate property and then install the application again. When federating nodes that have the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` set to `true` in the configuration of the servers contained within the node, this property does not affect the deployment manager. You must set the property to `true` on the deployment manager before the node is federated to preserve the behavior as it was on the node before federation.

Note: If this JVM property is being used on the z/OS platform, it must be set in both the servant and control regions of the server.

Note: When federating nodes that have the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` set to `true` in the configuration of the servers contained within the node, this does not affect the deployment manager. You must set the property to `true` on the deployment manager before the node is federated if you wish to preserve the behavior as it was on the node before federation.

Annotations supported by JAX-WS are listed in the table below. The target for annotations is applicable for these Java objects:

- types such as a Java class, enum or interface
- methods
- fields representing local instance variables within a Java class
- parameters within a Java method

Web Services Metadata Annotations (JSR 181)

Annotation class	Annotation	Properties
`javax.jws.` `WebService`	The **@WebService** annotation marks a Java class as implementing a Web service or marks a service endpoint interface (SEI) as implementing a Web service interface. **Important:** • A Java class that implements a Web service must specify either the @WebService or @WebServiceProvider annotation. Both annotations cannot be present. This annotation is applicable on a client or server SEI or a server endpoint implementation class. • If the annotation references an SEI through the `endpointInterface` attribute, the SEI must also be annotated with the @WebService annotation. • See “Rules for methods on classes annotated with @WebService” on page 467 for additional information.	• Annotation target: Type • Properties: **- name** The name of the `wsdl:portType`. The default value is the unqualified name of the Java class or interface. (String) **- targetNamespace** Specifies the XML namespace of the WSDL and XML elements generated from the Web service. The default value is the namespace mapped from the package name containing the Web service. (String) **- serviceName** Specifies the service name of the Web service: `wsdl:service`. The default value is the simple name of the Java class + *Service*. (String) **- endpointInterface** Specifies the qualified name of the service endpoint interface that defines the services’ abstract Web service contract. If specified, the service endpoint interface is used to determine the abstract WSDL contract. (String) **- portName** The `wsdl:portName`. The default value is WebService.name +*Port* . (String) **- wsdlLocation** Specifies the Web address of the WSDL document defining the Web service. The Web address is either relative or absolute. (String)

Annotation class	Annotation	Properties
`javax.jws.` `WebMethod`	The **@WebMethod** annotation denotes a method that is a Web service operation. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class. **Important:** • The @WebMethod annotation is only supported on classes that are annotated with the @WebService annotation.	• Annotation target: Method • Properties: **- operationName** Specifies the name of the `wsdl:operation` matching this method. The default value is the name of Java method. (String) **- action** Defines the action for this operation. For SOAP bindings, this value determines the value of the SOAPAction header. The default value is the name of Java method. (String) **- exclude** Specifies whether to exclude a method from the Web service. The default value is `false`. (Boolean)
`javax.jws.` `Oneway`	The **@Oneway** annotation denotes a method as a Web service one-way operation that only has an input message and no output message. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • There are no properties on the Oneway annotation.

Annotation class	Annotation	Properties
`javax.jws.` `WebParam`	The **@WebParam** annotation customizes the mapping of an individual parameter to a Web service message part and XML element. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Parameter • Properties: **- name** The name of the parameter. If the operation is remote procedure call (RPC) style and the `partName` attribute is not specified, then this is the name of the `wsdl:part` attribute representing the parameter. If the operation is document style or the parameter maps to a header, then `-name` is the local name of the XML element representing the parameter. This attribute is required if the operation is document style, the parameter style is `BARE`, and the mode is `OUT` or `INOUT`. (String) **- partName** Defines the name of `wsdl:part` attribute representing this parameter. This is only used if the operation is RPC style, or the operation is document style and the parameter style is `BARE`. (String) **- targetNamespace** Specifies the XML namespace of the XML element for the parameter. Applies only for document bindings when the attribute maps to an XML element. The default value is the `targetNamespace` for the Web service. (String) **- mode** The value represents the direction the parameter flows for this method. Valid values are `IN`, `INOUT`, and `OUT`. (String) **- header** Specifies whether the parameter is in a message header rather than a message body. The default value is `false`. (Boolean)

Annotation class	Annotation	Properties
`javax.jws.` `WebResult`	The **@WebResult** annotation customizes the mapping of a return value to a WSDL part or XML element. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- name** Specifies the name of the return value as it is listed in the WSDL file and found in messages on the wire. For RPC bindings, this is the name of the `wsdl:part` attribute representing the return value. For document bindings, the `-name` parameter is the local name of the XML element representing the return value. The default value is `return` for RPC and DOCUMENT/WRAPPED bindings. The default value is the method name + `Response` for DOCUMENT/BARE bindings. (String) **- targetNamespace** Specifies the XML namespace for the return value. This parameter is only used if the operation is RPC style or if the operation is DOCUMENT style and the parameter style is BARE. (String) **- header** Specifies whether the result is carried in a header. The default value is `false`. (Boolean) **- partName** Specifies the part name for the result with RPC or DOCUMENT/BARE operations. The default value is `@WebResult.name`. (String)

Annotation class	Annotation	Properties
`javax.jws.HandlerChain`	The **@HandlerChain** annotation associates the Web service with an externally defined handler chain. You can only configure the server side handler by using the @HandlerChain annotation on the Service Endpoint Interface (SEI) or the server endpoint implementation class. Use one of several ways to configure a client side handler. You can configure a client side handler by using the @HandlerChain annotation on the generated service class or SEI. Additionally, you can programmatically register your own implementation of the HandlerResolver interface on the Service, or programmatically set the handler chain on the Binding object.	• Annotation target: Type • Properties: **- file** Specifies the location of the handler chain file. The file location is either an absolute java.net.URL in external form or a relative path from the class file. (String) **- name** Specifies the name of the handler chain in the configuration file. (String)
`javax.jws.SOAPBinding`	The **@SOAPBinding** annotation specifies the mapping of the Web service onto the SOAP message protocol. Apply this annotation to a type or methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class. The method level annotation is limited in what it can specify and is only used if the `style` property is `DOCUMENT`. If the method level annotation is not specified, the **@SOAPBinding** behavior from the type is used.	• Annotation target: Type or Method • Properties: **- style** Defines encoding style for messages sent to and from the Web service. The valid values are `DOCUMENT` and `RPC`. The default value is `DOCUMENT`. (String) **- use** Defines the formatting used for messages sent to and from the Web service. The default value is `LITERAL`. `ENCODED` is not supported. (String) **- parameterStyle** Determines whether the method's parameters represent the entire message body or whether parameters are elements wrapped inside a top-level element named after the operation. Valid values are `WRAPPED` or `BARE`. You can only use the `BARE` value with `DOCUMENT` style bindings. The default value is `WRAPPED`. (String)

JAX-WS Annotations (JSR 224)

Annotation class	Annotation	Properties
`javax.xml.ws.Action`	The **@Action** annotation specifies the WS-Addressing action that is associated with a Web service operation. When you use this annotation with a particular method, and generate the corresponding WSDL document, the WS-Addressing Action extension attribute is added to the input and output elements of the WSDL operation that corresponds to that method. To add this attribute to the WSDL operation, you must also specify the @Addressing annotation on the server endpoint implementation class. If you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.	• Annotation target: Method • Properties: **- fault** Specifies the array of FaultAction for the wsdl:fault of the operation. (String) **- input** Specifies the action for thewsdl:input of the operation. (String) **- output** Specifies the action for thewsdl:output of the operation. (String)
`javax.xml.ws.BindingType`	The **@BindingType** annotation specifies the binding to use when publishing an endpoint of this type. Apply this annotation to a server endpoint implementation class. **Important:** • You can use the @BindingType annotation on the JavaBeans endpoint implementation class to enable MTOM by specifying either `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING` or `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING` as the value for the annotation.	• Annotation target: Type • Properties: **- value** Indicates the binding identifier Web address. Valid values are `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`, `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING`, and `javax.xml.ws.http.HTTPBinding.HTTP2HTTP_BINDING`. The default value is `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.FaultAction`	The **@FaultAction** annotation specifies the WS-Addressing action that is added to a fault response. This annotation must be contained within an @Action annotation. When you use this annotation with a particular method, the WS-Addressing FaultAction extension attribute is added to the fault element of the WSDL operation that corresponds to that method. To add this attribute to the WSDL operation, you must also specify the @Addressing annotation on the server endpoint implementation class. If you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.	• Annotation target: Method • Properties: **- value** Specifies the action of the wsdl:fault of the operation. (String) **- output** Specifies the name of the exception class. (String) **- className** Specifies the name of the class representing the request wrapper. (String)
`javax.xml.ws.RequestWrapper`	The **@RequestWrapper** annotation supplies the JAXB generated request wrapper bean, the element name, and the namespace for serialization and deserialization with the request wrapper bean that is used at runtime. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the `className` attribute is required in this case. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- localName** Specifies the local name of the XML schema element representing the request wrapper. The default value is the `operationName` as defined in javax.jws.WebMethod annotation. (String) **- targetNamespace** Specifies the XML namespace of the request wrapper method. The default value is the target namespace of the SEI. (String) **- className** Specifies the name of the class representing the request wrapper. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.ResponseWrapper`	The **@ResponseWrapper** annotation supplies the JAXB generated response wrapper bean, the element name, and the namespace for serialization and deserialization with the response wrapper bean that is used at runtime. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the `className` attribute is required in this case. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- localName** Specifies the local name of the XML schema element representing the request wrapper. The default value is the `operationName` + *Response*. `operationName` is defined in javax.jws.WebMethod annotation. (String) **- targetNamespace** Specifies the XML namespace of the request wrapper method. The default value is the target namespace of the SEI. (String) **- className** Specifies the name of the class representing the response wrapper. (String)
`javax.xml.ws.RespectBinding`	The **@RespectBinding** annotation specifies whether the JAX-WS implementation must use the contents of the wsdl:binding for an endpoint. When this annotation is specified, a check is performed to ensure all required WSDL extensibility elements with the enabled attribute set to true are supported. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Method • Properties: **- enabled** Specifies whether the wsdl:binding must be used or not. The default value is `true`. (Boolean)
`javax.xml.ws.ServiceMode`	The **@ServiceMode** annotation specifies whether a service provider needs to have access to an entire protocol message or just the message payload. **Important:** • The @ServiceMode annotation is only supported on classes that are annotated with the @WebServiceProvider annotation.	• Annotation target: Type • Properties: **- value** Indicates whether the provider class accepts the payload of the message, `PAYLOAD` or the entire message `MESSAGE`. The default value is `PAYLOAD`. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.soap.Addressing`	The **@Addressing** annotation specifies that this service wants to enable WS-Addressing support. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Type • Properties: **- enabled** Specifies if WS-Addressing is enabled or not. The default value is `true`. (Boolean) **- required** Specifies that WS-Addressing headers must be present on incoming messages. The default value is `false`. (Boolean)
`javax.xml.ws.soap.MTOM`	The **@MTOM** annotation specifies whether binary content in the body of a SOAP message is sent using MTOM. Apply this annotation to a service endpoint implementation class.	• Annotation target: Class • Properties: **- enabled** Specifies if MTOM is enabled for the JAX-WS endpoint. The default value is `true`. (Boolean) **- threshold** Specifies the minimum size for messages that are sent using MTOM. When the message size is less than this specified integer, the message is inlined in the XML document as base64 or hexBinary data. (integer)

Annotation class	Annotation	Properties
`javax.xml.ws.WebFault`	The **@WebFault** annotation maps WSDL faults to Java exceptions. It is used to capture the name of the fault during the serialization of the JAXB type that is generated from a global element referenced by a WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults. This annotation can only be applied to a fault implementation class on the client or server.	• Annotation target: Type • Properties: **- name** Specifies the local name of the XML element that represents the corresponding fault in the WSDL file. The actual value must be specified. (String) **- targetNamespace** Specifies the namespace of the XML element that represents the corresponding fault in the WSDL file. (String) **- faultBean** Specifies the name of the fault bean class. (String)
`javax.xml.ws.WebServiceProvider`	The **@WebServiceProvider** annotation denotes that a class satisfies requirements for a JAX-WS Provider implementation class. **Important:** • A Java class that implements a Web service must specify either the @WebService or @WebServiceProvider annotation. Both annotations cannot be present. • The @WebServiceProvider annotation is only supported on the service implementation class. Any class with the @WebServiceProvider annotation must implement the javax.xml.ws.Provider interface.	• Annotation target: Type • Properties: **- targetNamespace** Specifies the XML namespace of the WSDL and XML elements generated from the Web service. The default value is the namespace mapped from the package name containing the Web service. (String) **- serviceName** Specifies the service name of the Web service: `wsdl:service`. The default value is the simple name of the Java class + *Service*. (String) **- portName** The `wsdl:portName`. The default value is the name of the class + *Port*. (String) **- wsdlLocation** The Web address of the WSDL document defining the Web service. This attribute is required. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.WebServiceRef`	The **@WebServiceRef** annotation defines a reference to a Web service invoked by the client. **Important:** • The @WebServiceRef annotation can be used to inject instances of JAX-WS services and ports. • The @WebServiceRef annotation is only supported in certain class types. Examples are JAX-WS endpoint implementation classes, JAX-WS handler classes, Enterprise JavaBeans classes, and servlet classes. This annotation is supported in the same class types as the @Resource annotation. See the Java Platform, Enterprise Edition (Java EE) 5 specification for a complete list of supported class types.	• Annotation target: Type, Field or Method • Properties: **- name** Specifies the JNDI name of the resource. The field name is the default for field annotations. The JavaBeans property name that corresponds to the method is the default for method annotations. You must specify a value for class annotations as there is no default. (String) **- type** Indicates the Java type of the resource. The field type is the default for field annotations. The type of the JavaBeans property is the default for method annotations. You must specify a value for class annotations as there is no default. (Class) **- mappedName** Specified the name to map this resource to. (String) **- value** Indicates the value of the service class and it is a type that extends `javax.xml.ws.Service`. This attribute is required when the type of the reference is a service endpoint interface. (Class) **- wsdlLocation** The Web address of the WSDL document defining the Web service. This attribute is required. (String)
`javax.xml.ws.WebServiceRefs`	The **@WebServiceRefs** annotation associates multiple @WebServiceRef annotations with a specific class. **Important:** • The @WebServiceRef annotation is only supported in certain class types. Examples are JAX-WS endpoint implementation classes, JAX-WS handler classes, Enterprise JavaBeans classes, and servlet classes. This annotation is supported in the same class types as the @Resource annotation. See the Java Platform, Enterprise Edition (Java EE) 5 specification for a complete list of supported class types.	• Annotation target: Type • Properties: **- value** Specifies an array for multiple Web service reference declarations. This attribute is required.

JAX-WS Common Annotations (JSR 250)

Annotation class	Annotation	Properties
`javax.annotation.Resource`	The **@Resource** annotation marks a WebServiceContext resource needed by the application. **Important:** Applying this annotation to a WebServiceContext type field on the server endpoint implementation class for a JavaBeans endpoint or a Provider endpoint results in the container injecting an instance of the WebServiceContext into the specified field. When this annotation is used in place of the @WebServiceRef annotation, the rules described for the @WebServiceRef annotation apply.	• Annotation target: Field or Method • Properties: **- type** Indicates the Java type of the resource. You are required to use the default, `java.lang.Object` or `javax.xml.ws.Web ServiceContext` value. If the type is the default, the resource must be injected into a field or a method. In this case, the type of the field or the type of the JavaBeans property defined by the method must be `javax.xml.ws.WebServiceContext`. (Class) If you are using this annotation to inject a Web service, see the description of the @WebServiceRef type attribute.
`javax.annotation.Resources`	The **@Resources** annotation associates multiple @Resource annotations with a specific class and serves as a container for multiple resource declarations.	• Annotation target: Field or Method • Properties: **- value** Specifies an array for multiple @Resource annotations. This attribute is required.
`javax.annotation.PostConstruct`	The **@PostConstruct** annotation marks a method that needs to run after dependency injection is performed on the class. Apply this annotation to a JAX-WS application handler, a server endpoint implementation class.	• Annotation target: Method
`javax.annotation.PreDestroy`	The **@PreDestroy** annotation marks a method that must be run when the instance is in the process of being removed by the container. Apply this annotation to a JAX-WS application handler or a server endpoint implementation class.	• Annotation target: Method

IBM proprietary annotations

Annotation class	Annotation	Properties
`com.ibm.websphere.wsaddressing.jaxws21.SubmissionAddressing`	The **@SubmissionAddressing** annotation specifies that this service wants to enable WS-Addressing support for the 2004/08 WS-Addressing specification. This annotation is part of the IBM implementation of the JAX-WS 2.1 specification. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Type • Properties: **- required** Specifies that WS-Addressing headers must be present on incoming messages. The default value is `false`. (Boolean)

Rules for methods on classes annotated with @WebService

The following rules apply for methods on classes annotated with the @WebService annotation.

- If the @WebService annotation of an implementation class references an SEI, the implementation class must not have any @WebMethod annotations.
- All public methods for an SEI are considered exposed methods regardless of whether the @WebMethod annotation is specified or not. It is incorrect to have an @WebMethod annotation on an SEI that contains the `exclude` attribute.
- For an implementation class that does not reference an SEI, if the @WebMethod annotation is specified with a value of `exclude=true`, that method is not exposed. If the @WebMethod annotation is not specified, all public methods are exposed including the inherited methods with the exception of methods inherited from java.lang.Object.

JAX-WS application packaging:

You can package a Java Application Programming Interface (API) for XML Web Services (JAX-WS) application as a Web service. A JAX-WS Web service is contained within a Web archive (WAR) file or a WAR module within an enterprise archive (EAR) file.

A JAX-WS enabled WAR file contains:

- A WEB-INF/web.xml file
- Annotated classes that implement the Web services contained in the application module
- [Optional] Web Services Description Language (WSDL) documents that describe the Web services contained in the application module

A WEB-INF/web.xml file is similar to this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
</web-app>
```

The web.xml might contain servlet or servlet-mapping elements. When customizations to the web.xml file are not needed, the WebSphere Application Server runtime defines them dynamically as the module is loaded. For more information on configuring the web.xml file, read about customizing Web URL patterns in the web.xml file for JAX-WS applications.

Annotated classes must contain, at a minimum, a Web service implementation class that includes the `@WebService` annotation. The definition and specification of the Web services-related annotations are provided by the JAX-WS and JSR-181 specifications. The Web service implementation classes can exist within the `WEB-INF/classes` or directory within a Java archive (JAR) file that is contained in the `WEB-INF/lib` directory of the WAR file.

You can optionally include WSDL documents in the JAX-WS application packaging. If the WSDL document for a particular Web service is omitted, then the WebSphere Application Server runtime constructs the WSDL definition dynamically from the annotations contained in the Web service implementation classes. You must include the `@WebService`, `@WebMethod`, `@WebParam`, `@WebResult`, and optionally the `@SOAPBinding` annotations if the WSDL document is omitted.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre-Java EE 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the

default behavior is to not scan pre-Java EE 5 modules for annotations during application installation or server startup. You can preserve compatibility with feature packs from previous releases by either setting the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a Web archive (WAR) file or EJB module or by defining the Java virtual machine custom property, com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations documentation.

JAXB

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified development of Web services. JAXB leverages the flexibility of platform-neutral XML data in Java applications to bind XML schema to Java applications without requiring extensive knowledge of XML programming.

JAXB is an XML to Java binding technology that supports transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB consists of a runtime application programming interface (API) and accompanying tools that simplify access to XML documents. JAXB also helps to build XML documents that both conform and validate to the XML schema. Java API for XML-Based Web Services (JAX-WS) leverages the JAXB API and tools as the binding technology for mappings between Java objects and XML documents. JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding. JAXB 2.1 provides enhancements such as improved compilation support and support for the @XMLSeeAlso annotation. With JAXB 2.1, you can configure the xjc schema compiler so that it does not automatically generate new classes for a particular schema. Similarly, you can configure the schemagen schema generator to not automatically generate a new schema. This enhancement is useful when you are using a common schema and you do not want a new schema generated. JAXB 2.1 also introduces the @XMLSeeAlso annotation that enables JAXB to bind additional Java classes that it might not otherwise know about when binding a Java class with this annotation. This annotation enables JAXB to know about all classes that are potentially involved in marshalling or unmarshalling as it is not always possible or practical to list all of the subclasses of a given Java class. JAX-WS 2.1 also supports the use of the @XMLSeeAlso annotation on a service endpoint interface (SEI) or on a service implementation bean to ensure all of the classes referenced by the annotation are passed to JAXB for processing.

JAXB provides the xjc schema compiler tool, the schemagen schema generator tool, and a runtime framework. You can use the xjc schema compiler tool to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. You can also start with a set of JavaBeans and use the schemagen schema generator tool to create the XML schema. Once the mapping between XML schema and Java classes exists, XML instance documents can be converted to and from Java objects through the use of the JAXB binding runtime API. Data stored in XML documents can be accessed without the need to understand the data structure. You can then use the resulting Java classes to assemble a Web services application.

JAXB annotated classes and artifacts contain all the information needed by the JAXB runtime API to process XML instance documents. The JAXB runtime API supports marshaling of JAXB objects to XML and unmarshaling the XML document back to JAXB class instances. Optionally, you can use JAXB to provide XML validation to enforce both incoming and outgoing XML documents to conform to the XML constraints defined within the XML schema.

JAXB is the default data binding technology used by the Java API for XML Web Services (JAX-WS) tooling and implementation within this product. You can develop JAXB objects for use within JAX-WS applications.

You can also use JAXB independently of JAX-WS when you want to leverage the XML data binding technology to manipulate XML within your Java applications.

The following diagram illustrates the JAXB architecture.

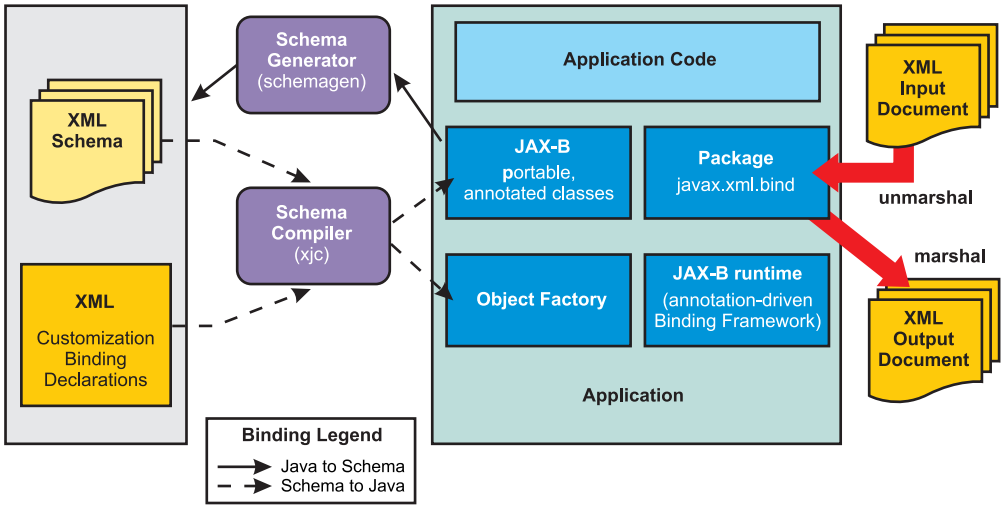


Figure 3. JAXB architecture

JAX-RPC

The *Java API for XML-based RPC (JAX-RPC)* specification enables you to develop SOAP-based interoperable and portable Web services and Web service clients. JAX-RPC 1.1 provides core APIs for developing and deploying Web services on a Java platform and is a part of the Web Services for Java Platform, Enterprise Edition (Java EE) platform. The Java EE platform enables you to develop portable Web services.

WebSphere Application Server implements JAX-RPC 1.1 standards.

The JAX-RPC standard covers the programming model and bindings for using Web Services Description Language (WSDL) for Web services in the Java language. JAX-RPC simplifies development of Web services by shielding you from the underlying complexity of SOAP communication.

On the surface, JAX-RPC looks like another instantiation of remote method invocation (RMI). Essentially, JAX-RPC enables clients to access a Web service as if the Web service was a local object mapped into the client's address space even though the Web service provider is located in another part of the world. The JAX-RPC is done by using the XML-based protocol SOAP, which typically rides on top of HTTP.

JAX-RPC defines the mappings between the WSDL port types and the Java interfaces, as well as between Java language and Extensible Markup Language (XML) schema types.

A JAX-RPC Web service can be created from a JavaBean or a enterprise bean implementation. You can specify the remote procedures by defining remote methods in a Java interface. You only need to code one or more classes that implement the methods. The remaining classes and other artifacts are generated by the Web service vendor's tools. The following is an example of a Web service interface:

```
package com.ibm.mybank.ejb;
import java.rmi.RemoteException;
import com.ibm.mybank.exception.InsufficientFundsException;
/**
 * Remote interface for Enterprise Bean: Transfer
 */
public interface Transfer_SEI extends java.rmi.Remote {
```

```
    public void transferFunds(int fromAcctId, int toAcctId, float amount)
       throws java.rmi.RemoteException;

}
```

The interface definition in JAX-RPC must follow specific rules:

- The interface must extend `java.rmi.Remote` just like RMI.
- Methods must create `java.rmi.RemoteException`.
- Method parameters cannot be remote references.
- Method parameter must be one of the parameters supported by the JAX-RPC specification. The following list are examples of method parameters that are supported. For a complete list of method parameters see the JAX-RPC specification.
  - Primitive types: `boolean`, `byte`, `double`, `float`, `short`, `int` and `long`
  - Object wrappers of primitive types: `java.lang.Boolean`, `java.lang.Byte`, `java.lang.Double`, `java.lang.Float`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Short`
  - `java.lang.String`
  - `java.lang.BigDecimal`
  - `java.lang.BigInteger`
  - `java.lang.Calendar`
  - `java.lang.Date`
- Methods can take value objects which consist of a composite of the types previously listed, in addition to aggregate value objects.

A client creates a stub and invokes methods on it. The stub acts like a proxy for the Web service. From the client code perspective, it seems like a local method invocation. However, each method invocation gets marshaled to the remote server. Marshaling includes encoding the method invocation in XML as prescribed by the SOAP protocol.

The following are key classes and interfaces needed to write Web services and Web service clients:

- **Service interface**: A factory for stubs or dynamic invocation and proxy objects used to invoke methods
- **ServiceFactory class**: A factory for Services.
- **`loadService`**

  The `loadService` method is provided in WebSphere Application Server Version 6.0 to generate the service locator which is required by a JAX-RPC implementation. If you recall, in previous versions there was no specific way to acquire a generated service locator. For managed clients you used a JNDI method to get the service locator and for non-managed clients, you were required to instantiate IBM's specific service locator `ServiceLocator service=new ServiceLocator(...);` which does not offer portability. The `loadService` parameters include:
  - **`wsdlDocumentLocation:`** A URL for the WSDL document location for the service or null.
  - **`serviceName`**: A qualified name for the service
  - **`properties`**: A set of implementation-specific properties to help locate the generated service implementation class.
- **`isUserInRole`**

  The `isUserInRole` method returns a boolean indicating whether the authenticated user for the current method invocation on the endpoint instance is included in the specified logical role.
  - **`role`**: The role parameter is a String specifying the name of the role.
- **Service**
- **Call interface**: Used for dynamic invocation
- **Stub interface**: Base interface for stubs

If you are using a stub to access the Web service provider, most of the JAX-RPC API details are hidden from you. The client creates a ServiceFactory (`java.xml.rpc.ServiceFactory`). The client instantiates a Service (`java.xml.rpc.Service`) from the ServiceFactory. The service is a factory object that creates the port. The port is the remote service endpoint interface to the Web service. In the case of DII, the Service object is used to create Call objects, which you can configure to call methods on the Web service's port.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

RMI-IIOP using JAX-RPC:

Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) can be used with JAX-RPC to support non-SOAP bindings.

Java API for XML-based Remote Procedure Call (JAX-RPC) is the Java standard API for invoking Web services through remote procedure calls. A transport is used by a programming language to communicate over the Internet. You can use protocols with the transport such as SOAP and Remote Method Invocation (RMI). You can use Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) with JAX-RPC to support non-SOAP bindings.

Using RMI-IIOP with JAX-RPC, enables WebSphere Java clients to invoke enterprise beans using a WSDL file and the JAX-RPC programming model instead of using the standard Web Services for Java Platform, Enterprise Edition (Java EE) programming model. When a enterprise JavaBeans implementation is used to invoke a Web service, multiprotocol JAX-RPC permits the Web service invocation path to be optimized for WebSphere Java clients. To learn more this optimization, read about using enterprise bean bindings to invoke an EJB from a Web services client.

Benefits of using the RMI/IIOP protocol instead of a SOAP- based protocol are:

- XML processing is not required to send and receive messages; Java serialization is used instead.
- The client JAX-RPC call can participate in a user transaction, which is not the case when SOAP is used.

Web Services-Interoperability Basic Profile

The Web Services-Interoperability (WS-I) Basic Profile is a set of non-proprietary Web services specifications that promote interoperability. WebSphere Application Server conforms to the WS-I Basic Profile Version 1.1 and WS-I Basic Security Profile Version 1.0.

The WS-I Basic Profile is governed by a consortium of industry-leading corporations, including IBM, under direction of the WS-I Organization. The profile consists of a set of principles that relate to bringing about open standards for Web services technology. All organizations that are interested in promoting interoperability among Web services are encouraged to become members of the Web Services Interoperability Organization.

Several technology components are used in the composition and implementation of Web services, including messaging, description, discovery, and security. Each of these components are supported by specifications and standards, including SOAP 1.1, Extensible Markup Language (XML) 1.0, HTTP 1.1, Web Services Description Language (WSDL) 1.1, and Universal Description, Discovery and Integration (UDDI). The WS-I Basic Profile specifies how these technology components are used together to achieve interoperability, and mandates specific use of each of the technologies when appropriate. You can read more about the WS-I Basic Profile at the WS-I Organization Web site.

As technology components are updated, these components are also used in the composition and implementation of Web Services. One example is that both SOAP 1.1 and SOAP 1.2 are now supported.

Note: Building on the support for WS-I Basic Profile Version 1.0, WS-I Basic Profile V1.1, Attachment Profile V1.0, and Basic Security Profile (BSP) V1.0, you can implement Web services with WebSphere Application Server Version 7.0 using the following current emerging standard WS-I profiles:

- *WS-I Basic Profile V1.2* builds on WS-I Basic Profile V1.0 and WS-I Basic Profile V1.1 and adds support for WS-Addressing (WS-A) and SOAP Message Transmission Optimization Mechanism (MTOM). The WS-Addressing specification enables the asynchronous message exchange pattern so that you can decouple the service request from the service response. The SOAP header of the sender's request contains the wsa:ReplyTo value that defines the endpoint reference to which the provider's response is sent. Decoupling the request from the response enables long running Web services interactions. Leveraging the asynchronous programming model support in JAX-WS Version 2.1 in combination with WS-Addressing, you can now take advantage of the ability to create Web services invocations where the client can continue to process work without waiting for a response to return. This provides for a more dynamic and efficient model to invoke Web services. Using MTOM, you can send and receive binary data optimally within a SOAP message.
- *WS-I Basic Profile V2.0* builds on top of Basic Profile V1.2 with the addition of support for SOAP 1.2.
- *WS-I Basic Security Profile V1.1* extends the WS-I Basic Security Profile V1.0 standard by profiling the latest WS-Security V1.1 specification.
- *WS-I Reliable Secure Profile 1.0* builds on WS-I Basic Profile V1.2, WS-I Basic Profile V2.0, WS-I Basic Security Profile V1.0, and WS-I Basic Security Profile V1.1 and adds support for WS-Reliable Messaging 1.1, WS-Make Connection 1.0, and WS-Secure Conversation 1.3. WS-Reliable Messaging 1.1 is a session-based protocol that provides message level reliability for Web services interactions. WS-Make Connection 1.0 was developed by the WS-Reliable Messaging workgroup to address scenarios where a Web services endpoint is behind a firewall or the endpoint has no visible endpoint reference. If a Web services endpoint loses connectivity during a reliable session, WS-Make Connection provides an efficient method to re-establish the reliable session. Additionally, WS-Secure Conversation V1.3 is a session-based security protocol that uses an efficient symmetric key based encryption algorithm for message level security. WS-I Reliable Secure Profile V1.0 provides secure reliable session-oriented Web services interactions.

Each of the technology components has requirements that you can read about in more detail at the WS-I Organization Web site. For example, support for Universal Transformation Format (UTF)-16 encoding is required by WS-I Basic Profile. UTF-16 is a kind of Unicode encoding scheme that uses 16-bit values to store Universal Character Set (UCS) characters. UTF-8 is the most common encoding that is used on the Internet; UTF-16 encoding is typically used for Java and Windows product applications; and UTF-32 is used by various Linux and UNIX systems. Unlike UTF-8, UTF-16 has issues with big-endian and little-endian, and often involves Byte Order Mark (BOM) to indicate the endian. BOM is mandatory for UTF-16 encoding and it can be used in UTF-8.

The application server only supports UTF-8 and UTF-16 encoding of SOAP messages.

The following table summarizes some of the properties of each UTF:

Bytes	Encoding form
EF BB BF	UTF-8
FF FE	UTF-16, little-endian
FE FF	UTF-16, big-endian
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian

BOM is written prior to the XML text, and it indicates to the parser how the XML is encoded. The XML declaration contains the encoding, for example: <?xml version=*xxx* encoding="utf-*xxx*"?>. BOM is used with the encoding to determine how to interpret the XML. Here is an example of a SOAP message and how BOM and UTF encoding are used:

```
POST http://www.whitemesa.net/soap12/add-test-rpc HTTP/1.1
Content-Type: application/soap+xml; charset=utf-16; action=""
SOAPAction:
Host: localhost: 8080
Content-Length: 562

0xFF0xFE<?xml version="1.0" encoding="utf-16"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2002/12/soap-envelope"
 xmlns:soapenc="http://www.w3.org/2002/12/soap-encoding
 xmlns:tns="http://whitemesa.net/wsdl/soap12-test"
 xmlns:types="http://whitemesa.net/wsdl/soap12-test/encodedTypes"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soap:Body>
  <q1:echoString xmlns:q1="http://soapinterop.org/">
   <inputString soap:encodingStyle="http://example.org/unknownEncoding"
   xsi:type="xsd:string">
    Hello SOAP 1.2
   </inputString>
  </q1:echoString>
 </soap:Body>
</soap:Envelope>
```

In the example code, 0xFF0xFE represents the byte codes, while the *<?xml>* declaration is the textual representation.

Support for styleEncoding is not supported in SOAP 1.2 so here is the same example of the SOAP message but without the encoding information:

```
OxFF0xFE<?xml version="1.0" encoding="utf-16"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2002/12/soap-envelope"
   xmlns:soapenc="http://www.w3.org/2002/12/soap-encoding
 xmlns:tns="http://whitemesa.net/wsdl/soap12-test"
 xmlns:types="http://whitemesa.net/wsdl/soap12-test/encodedTypes"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soap:Body>
  <q1:echoString xmlns:q1="http://soapinterop.org/">
   <inputString xsi:type="xsd:string">
    Hello SOAP 1.2
   </inputString>
  </q1:echoString>
 </soap:Body>
</soap:Envelope>
```

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

WS-I Attachments Profile

The *Web Services-Interoperability (WS-I) Attachments Profile* is a set of non-proprietary Web services specifications that promote interoperability. This profile compliments the WS-I Basic Profile 1.1 to add support for interoperable SOAP messages with attachments-based Web services.

WebSphere Application Server conforms to the WS-I Attachments Profile 1.0.

Attachments are typically used to send binary data, for example, data that is mapped in Java code to java.awt.Image and javax.activation.DataHandler. The raw data can be sent in the SOAP message, however, this approach is inefficient because an XML parser has to scan the data as it parses the message.

The WS-I Attachments Profile provides a solution to the limitations that are presented by Web Services Description Language (WSDL) 1.1. Because WSDL 1.1 attachments are not part of the XML schema type space, they can be message parts only. As message parts, the attachments cannot be arrays or properties of Java beans. The profile defines the wsi:swaRef XML schema type.Use the wsi:swaRef XML schema type to overcome the limitations of WSDL 1.1 attachments.

The wsi:swaRef type is an extension of the xsd:anyURI type, where its value contains the content-ID of the attachment.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Web services migration scenarios: JAX-RPC to JAX-WS and JAXB

This topic explains scenarios for migrating your Java API for XML-based RPC (JAX-RPC) Web services to Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) Web services.

Changes in the tooling can be largely hidden from the user by a development environment like the assembly tools available with WebSphere Application Server. Also, depending on the data specified in the XML, some methods that were used for the JAX-RPC service can be used with little or no change in a JAX-WS service. There are also conditions that do require changes to be made in the JAX-WS service.

Because Java EE environments emphasize compatibility, most application servers that offer support for the newer JAX-WS and JAXB specifications continue to support the previous JAX-RPC specification. A consequence of this is that existing Web services are likely to remain JAX-RPC based while new Web services are developed using the JAX-WS and JAXB programming models.

However, as time passes and applications are revised and rewritten, there might be times when the best strategy is to migrate a JAX-RPC based Web service to one that is based on the JAX-WS and JAXB programming models. This might result from a vendor choosing to provide enhancements to qualities of service that are only available in the new programming models. For example, SOAP 1.2 and SOAP Message Transmission Optimization Mechanism (MTOM) support are only available within the JAX-WS 2.x and JAXB 2.x programming models and not JAX-RPC.

The following information includes issues that you might have while migrating from JAX-RPC to JAX-WS and JAXB.

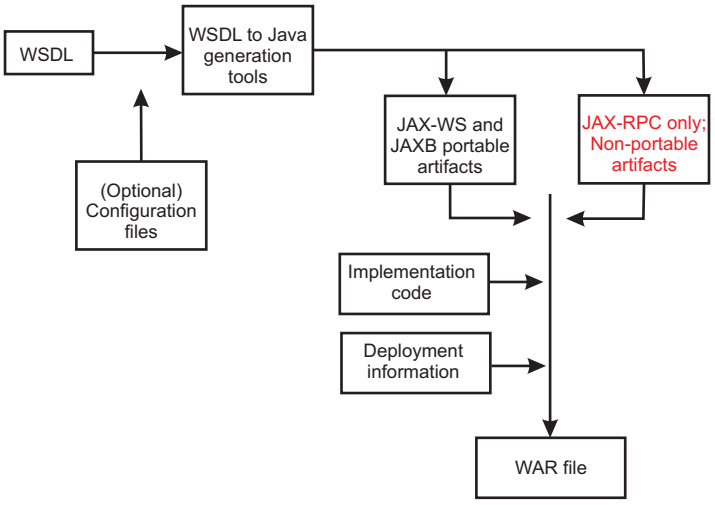
Note: When the terms *migrate*, *migrating*, and *migration* are used in this topic, unless there is some statement to indicate otherwise, the move from a JAX-RPC to a JAX-WS and JAXB environment is being described.

Comparison of JAX-RPC to JAX-WS and JAXB programming models

If you are looking for additional information that describes the changes between the JAX-RPC and JAX-WS and JAXB programming models, IBM developerWorks has published several Web services hints and tips topics.

Development models

The high level combined development models for JAX-RPC and JAX-WS and JAXB are shown the



following image:

At a high level, the models are very similar. The only difference is that the JAX-RPC model produces both portable and non-portable artifacts; the JAX-WS and JAXB model does not produce non-portable artifacts.

The image displays the following components:

- **Web Services Description Language (WSDL) file:** A document conforming to the WSDL recommendation as published by the W3C organization.
- **(Optional) Configuration files**
- **WSDL to Java generation tool:** The specifications describe the mappings required, not the name of the tools. Typically, the tool for JAX-WS and JAXB is wsimport, and for JAX-RPC the tools are wscompile and wsdeploy.
- **Portable artifacts:** These are files generated by the WSDL to Java generation tool that have well-defined mappings in the JAX-RPC or JAX-WS and JAXB, whichever is applicable, specifications. These artifacts can be used without modification between different implementations of JAX-RPC or JAX-WS and JAXB.
- **Non-portable artifacts:** These are files generated by the WSDL to Java generation tool that do not have well-defined mappings in the JAX-RPC or JAX-WS and JAXB, whichever is applicable, specifications. These artifacts generally require modification between different implementations of JAX-RPC or JAX-WS/JAXB.
- **Implementation code:** This is the code that you need to meet particular requirements for implementation.
- **Deployment information:** Additional information that is needed to run the application in a particular environment.
- **Web archive (WAR) file:** In the context of the SampleService image, a WAR file is an archive file that contains all items needed to deploy a Web service into a particular environment. This is sometimes called a *cooked WAR file*.

The Development and runtime environments

A specialized development environment simplifies Web services migration by automating many of the tasks. For development of WebSphere-based applications, including Web services, you can use the assembly tools. You can read more about the use of the assembly tools in the assembly tool information center.

The following sample code in this topic was tested in the following runtime environment:

- IBM WebSphere Application Server V6.1, which includes support for the JAX-RPC specification.
- IBM WebSphere Application Server V6.1 Feature Pack for Web Services, which includes support for the JAX-WS and JAXB specifications.

Examples

The following examples show some of the code changes that you might need to migrate your JAX-RPC Web services to JAX-WS and JAXB Web services. In particular, there is an emphasis on the changes in the implementation code that you must write, from both the client and the server side. If no new function is to be introduced, the changes required to move from JAX-RPC to JAX-WS and JAXB might be few.

Sample Web service

The first example is a sample JAX-RPC-based Web Service that was created from a WSDL file (top-down development); the same WSDL file is used to generate the JAX-WS and JAXB-based service. The WSDL file describes the Web service, SampleService, with these operations described by the following image:

As shown in the image, the five operations offered are:

- xsd:int calcShippingCost(xsd:int, xsd:int)
- xsd:string getAccountNumber(xsd:string)
- xsd:dateTime calculateShippingDate(xsd:dateTime)
- xsd:string ckAvailability(xsd:int) creates invalidDateFault
- Person findSalesRep(xsd:string)

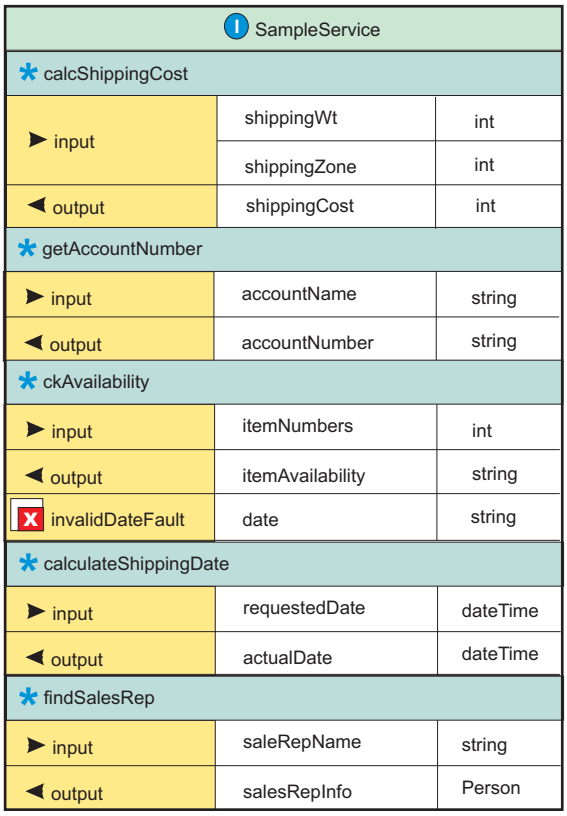
Also assume that Person is defined by the following schema fragment in the WSDL file:

```
<complexType name="Person">
 <sequence>
  <element name="name" type="xsd:string"/>
  <element name="age" type="xsd:int"/>
  <element name="location" type="impl:Address/>
 </sequence>
</complexType>
```

Address is defined by:

```
<complexType name="Address">
 <sequence>
  <element name="street" type="xsd:string"/>
  <element name="city" type="xsd:string"/>
  <element name ="state" type="xsd:string"/>
  <element name="zip" type="xsd:int"/>
 </sequence>
</complexType>
```

The following image also shows that the operation, ckAvailability(xsd:int), which produces an invalidDateFault exception.



Service operations

Review the service code created by the tooling. The following information includes examining what is created for a JAX-RPC runtime and also for a JAX-WS and JAXB runtime.

JAX-RPC

For JAX-RPC, the tooling accepts the WSDL file as input, and, amongst other files, generates the SampleService.java and SampleServiceImpl.java interfaces. The SampleService.java interface defines an interface and the generated code can be review in the following code block. The SampleServiceSoapBindingImpl.java interface provides the skeleton of an implementation, and you typically modify to add your own logic.

JAX-RPC version of SampleService.java:

```
/**
* SampleService.java
*
* This file was auto-generated from WSDL  * by the IBM Web services WSDL2Java emitter.  * cf20633.22 v82906122346
*/
package simple;
public interface SampleService extends java.rmi.Remote {
 public java.lang.Integer calcShippingCost(java.lang.Integer shippingWt,
  java.lang.Integer shippingZone) throws java.rmi.RemoteException;
 public java.lang.String getAccountNumber(java.lang.String accountName)
  throws java.rmi.RemoteException;
 public java.lang.String[] ckAvailability(int[] itemNumbers)
  throws java.rmi.RemoteException, simple.InvalidDateFault;
 public java.util.Calendar calculateShippingDate(
  java.util.Calendar requestedDate)
  throws java.rmi.RemoteException;
 public simple.Person findSalesRep(java.lang.String saleRepName)
  throws java.rmi.RemoteException; }
```

JAX-WS and JAXB

For JAX-WS and JAXB, the tooling accepts the WSDL file as input, and as in the case of JAX-RPC, generates SampleService.java and SampleServiceImpl.java interfaces. As with JAX-RPC, the SampleService.java interface also defines an interface as shown in the following code block. The SampleServiceImpl.java interface provides the skeleton of an implementation, and you typically modify to add your own logic.

Note: The code has been annotated by the tooling.
JAX-WS and JAXB version of the SampleService.java interface:

```
package simple;
 import java.util.List;
 import javax.jws.WebMethod;
 import javax.jws.WebParam;
 import javax.jws.WebResult;
 import javax.jws.WebService;
 import javax.xml.datatype.XMLGregorianCalendar;
 import javax.xml.ws.RequestWrapper;
 import javax.xml.ws.ResponseWrapper;

/**
* This class was generated by the JAX-WS SI.  * JAX-WS RI IBM 2.0_03-06/12/2007 07:44 PM(Raja)-fcs  *
Generated source version: 2.0
*
*/
@WebService(name = "SampleService", targetNamespace = "http://simple") public interface SampleService {

 /**
 *
 * @param shippingWt
 * @param shippingZone
 * @return
 * returns java.lang.Integer
 */
 @WebMethod
 @WebResult(name = "shippingCost", targetNamespace = "")
 @RequestWrapper(localName = "calcShippingCost", targetNamespace = "http://simple",
className = "simple.CalcShippingCost")
 @ResponseWrapper(localName = "calcShippingCostResponse", targetNamespace = "http://simple",
className = "simple.CalcShippingCostResponse")
 public Integer calcShippingCost(
 @WebParam(name = "shippingWt", targetNamespace = "")
 Integer shippingWt,
 @WebParam(name = "shippingZone", targetNamespace = "")
 Integer shippingZone);

 /**
 *
 *
 @param accountName
 * @return
 *     returns java.lang.String
 */
 @WebMethod
 @WebResult(name = "accountNumber", targetNamespace = "")
 @RequestWrapper(localName = "getAccountNumber", targetNamespace = "http://simple",
className = "simple.GetAccountNumber")
 @ResponseWrapper(localName = "getAccountNumberResponse", targetNamespace = "http://simple",
className = "simple.GetAccountNumberResponse")
 public String getAccountNumber(
  @WebParam(name = "accountName", targetNamespace = "")
  String accountName);
 /**
 *
 * @param requestedDate
 * @return
 *     returns javax.xml.datatype.XMLGregorianCalendar
 */
 @WebMethod
 @WebResult(name = "actualDate", targetNamespace = "")
 @RequestWrapper(localName = "calculateShippingDate", targetNamespace = "http://simple",
className = "simple.CalculateShippingDate")
 @ResponseWrapper(localName = "calculateShippingDateResponse", targetNamespace = "http://simple",
className = "simple.CalculateShippingDateResponse")
```

```
 public XMLGregorianCalendar calculateShippingDate(
  @WebParam(name = "requestedDate", targetNamespace = "")
  XMLGregorianCalendar requestedDate);
 /**
 *
 * @param itemNumbers
 * @return
 *     returns java.util.List<java.lang.String>
 * @throws InvalidDateFault_Exception
 */
 @WebMethod
 @WebResult(name = "itemAvailability", targetNamespace = "")
 @RequestWrapper(localName = "ckAvailability", targetNamespace = "http://simple", className = "simple.CkAvailability")
 @ResponseWrapper(localName = "ckAvailabilityResponse", targetNamespace = "http://simple",
className = "simple.CkAvailabilityResponse")
 public List<String> ckAvailability(
  @WebParam(name = "itemNumbers", targetNamespace = "")
  List<Integer> itemNumbers)
  throws InvalidDateFault_Exception
 ;
 /**
 *
 * @param saleRepName
 * @return
 *     returns simple.Person
 */
 @WebMethod
 @WebResult(name = "salesRepInfo", targetNamespace = "")
 @RequestWrapper(localName = "findSalesRep", targetNamespace = "http://simple", className = "simple.FindSalesRep")
 @ResponseWrapper(localName = "findSalesRepResponse", targetNamespace = "http://simple",
className = "simple.FindSalesRepResponse")
 public Person findSalesRep(
 @WebParam(name = "saleRepName", targetNamespace = "")
 String saleRepName);

}
```

Comparing the code examples

At first glance, it might seem that there is little similarity between the interfaces. If you disregard the additional information added by the annotations for JAX-WS and JAXB, the code samples are similar. For example, the calcShippingCost method in the JAX-WS and JAXB version the following lines of code exist:

```
@WebMethod
 @WebResult(name = "shippingCost", targetNamespace = "")
 @RequestWrapper(localName = "calcShippingCost", targetNamespace = "http://simple",
className = "simple.CalcShippingCost")
 @ResponseWrapper(localName = "calcShippingCostResponse", targetNamespace = "http://simple",
className = "simple.CalcShippingCostResponse")
 public Integer calcShippingCost(
  @WebParam(name = "shippingWt", targetNamespace = "")
  Integer shippingWt,
  @WebParam(name = "shippingZone", targetNamespace = "")
  Integer shippingZone);
```

But, if you discard the annotations, the following lines of code are:

```
public Integer calcShippingCost(
        Integer shippingWt,
        Integer shippingZone);
```

These lines are almost identical to what was generated for JAX-RPC. The only difference is that the JAX-RPC code can produce the java.rmi.RemoteException error as follows:

```
public java.lang.Integer calcShippingCost(java.lang.Integer shippingWt,
   java.lang.Integer shippingZone) throws java.rmi.RemoteException;
```

Following this logic, three of the methods have essentially the same signatures:

```
public Integer calcShippingCost(Integer shippingWt, Integer shippingZone)
 public String getAccountNumber(String accountName)
 public Person findSalesRep(String saleRepName)
```

This means that migrating from JAX-RPC to JAX-WS does not directly affect these methods and the original implementation code that is successfully running in the JAX-RPC based environment can probably be used without modification for these methods.

However two of the methods do have different signatures:

For JAX-RPC:

```
public java.util.Calendar calculateShippingDate(
 java.util.Calendar requestedDate)
public java.lang.String[] ckAvailability(int[] itemNumbers)
 throws java.rmi.RemoteException,
 simple.InvalidDateFault
```

JAX-WS and JAXB:

```
public XMLGregorianCalendar calculateShippingDate(
 XMLGregorianCalendar requestedDate)
public List<String> ckAvailability(List<Integer> itemNumbers)
 throws InvalidDateFault_Exception
```

Note:

You might find it easier to compare the skeleton implementation files since the annotations are not present in SampleServiceImpl.java.:

- SampleServiceSoapBindingImpl.java
- SampleServiceImpl.java

The differences in signatures are due to the following reasons:

- Differences mappings from XML names to Java names

  For the calculateShippingDate method, both the input parameter and the return parameter have changed from type java.util.Calendar to type XMLGregorianCalendar. This is because the WSDL specified these parameters to be of type, xsd:dateTime, JAX-RPC maps this data type to java.util.Calendar, while JAX-WS and JAXB maps it to XMLGregorianCalendar.
- Different mappings of arrays from XML to Java

  For the ckAvailability method, the change is due to the data mappings for XML arrays.

  JAX-RPC maps the following WSDL elements:

  ```
  <element maxOccurs="unbounded" name="itemNumbers" type="xsd:int"/>
  <element maxOccurs="unbounded" name="itemAvailability" type="xsd:string"/>
  ```

  The previous elements are mapped to the following Java source:

  ```
  int[] itemNumbers
  java.lang.String[] itemAvailability
  ```

  JAX-WS and JAXB map the following WSDL elements:

  ```
  List<Integer> itemNumbers
  List<String> ckAvailability
  ```
- Different mappings of exceptions

  For the ckAvailability method, the JAX-RPC code generated the following error:

  ```
  simple.InvalidDateFault
  ```

  The JAX-WS code generates the following error:

  ```
  InvalidDateFault_Exception
  ```

  Except for the names, the constructors for these exceptions are different. Therefore, the actual JAX-RPC code that produces the error might be displayed as:

```
throw new InvalidDateFault("this is an InvalidDateFault");
```

For JAX-WS, the code uses a command similar to the following example::

```
throw new InvalidDateFault_Exception( "this is an InvalidDateFault_Exception", new InvalidDateFault());
```

In cases that use exceptions, the code that uses it needs to change.

Migrating the code

Now that the differences between code have been explained, review the original code for the methods that need changing and how to change this code so that it works in a JAX-WS and JAXB environment.

Assume that the original (JAX-RPC) implementation code is similar to following:

```
public java.util.Calendar calculateShippingDate(
     java.util.Calendar requestedDate) throws java.rmi.RemoteException {
// Set the date to the date that was sent to us and add 7 days.
requestedDate.add(java.util.Calendar.DAY_OF_MONTH, 7);

// . . .

return requestedDate;
}
```

You can write the new code to directly use the new types as in the following examples:

```
public XMLGregorianCalendar calculateShippingDate(
XMLGregorianCalendar requestedDate) {
try {
// Create a data type factory.
DatatypeFactory df = DatatypeFactory.newInstance();
// Set the date to the date that was sent to us and add 7 days.
Duration duration = df.newDuration("P7D");
requestedDate.add(duration);

} catch (DatatypeConfigurationException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

// . . .

return requestedDate;
}
```

Migrating the code for the ckAvailability method

When migrating the code for the ckAvailability method, changes were required because of the way that arrays and exceptions are mapped from XML is different between JAX-RPC and JAX-WS. Assume that the original JAX-RPC Web service code is similar to the following example:

```
public java.lang.String[] ckAvailability(int[] itemNumbers)
{

    String[] myString = new String[itemNumbers.length];
    for (int j = 0; j < myString.length; j++) {

            . . .
             if ( ... )

    throw new simple.InvalidDateFault("InvalidDateFault");

            . . .
            if ( . . .)
               myString[j] = "available:  " + jitemNumbers[j] ;
```

```
            else
               myString[j] = "not available:  " + jitemNumbers[j];

    }
    return myString;
}
```

The previous code accepts an int[] as input and returns a String[]. For the JAX-WS and JAXB version, these are List<Integer> and List<String> elements respectively. Processing for these arrays, and discarding the Exception code, the JAX-RPC code is similar to the following:

```
public java.lang.String[] ckAvailability(int[] itemNumbers)
{

    String[] myString = new String[itemNumbers.length];
    for (int j = 0; j < jitemNumbers.length; j++) {


            . . .
            if ( . . .)
               myString[j] = "available:  " + itemNumbers.get(j);
            else
               myString[j] = "not available:  " + itemNumbers.get(j);
    }
    return myString;
}
```

The following JAX-WS and JAXB equivalent code exists using Lists instead of arrays:

```
List <String> ckAvailability(List <Integer> itemNumbers)
{

    ArrayList<String> retList = new ArrayList<String>();
    for (int count = 0; count < itemNumbers.size(); count++) {


            . . .
            if ( . . .)
               retList.add("available:  " + itemNumbers.get(j));
            else
               retList.add("not available:  " + itemNumbers.get(j));
    }
    return retList;
}
```

The differences in the mappings of exceptions from XML to Java forces the JAX-WS code to use. InvalidDateFault_Exception instead of InvalidDateFault.

This means that you must replace `throw new simple.InvalidDateFault("InvalidDateFault");` with some other code. Therefore, the following line is used for the new code:

```
throw new InvalidDateFault_Exception( "test InvalidDateFault_Exception", new InvalidDateFault());
```

The final JAX-WS and JAXB implementation of the method might be similar to the following code:

```
List <String> ckAvailability(List <Integer> itemNumbers)
{

    ArrayList<String> retList = new ArrayList<String>();
    for (int count = 0; count < itemNumbers.size(); count++) {

if ( . . . ) {
   throw new InvalidDateFault_Exception(
   "test InvalidDateFault_Exception",
   new InvalidDateFault());
}
```

```
. . .
if ( . . .)
   retList.add("available:  " + itemNumbers.get(count));
else
   retList.add("not available:  " + itemNumbers.get(count));
   }
   return retList;
}
```

There are multiple ways that you might choose to migrate the code. With practice and an effective development environment, migrating from JAX-RPC to JAX-WS can be straightforward.

Web services migration best practices

Use these Web services migration best practices when migrating Web services applications.

If you have used the Apache SOAP support to develop Web services client applications in WebSphere Application Server Versions 4, 5, or 5.1, you might need to migrate your applications or the security files for your applications. The following table summarizes the Web services specifications supported by the WebSphere products.

WebSphere Application Server Version	Web services specifications supported
4.0	Apache SOAP 2.2
5.0 and 5.0.1	Apache SOAP 2.3
5.0.2 or later	Java 2 Platform, Enterprise Edition (J2EE), also known as (JSR 109)
6.0.x and 6.1	J2EE (JSR 109)
7.0 or later	Web Services for Java Platform, Enterprise Edition (Java EE) 5 also known as JSR 109

Note: The Apache SOAP 2.2 and Apache SOAP 2.3-based implementations that were available in WebSphere Application Server Version 4.0.*x*, 5.0 and 5.0.1 have been deprecated. It is recommended that applications that are using these SOAP implementations migrate to Web Services for Java EE (JSR 109) support that is provided in current WebSphere Application Server versions.

For more information on migrating your Web services, see Migrating Apache SOAP Web services to Web Services to J2EE standards .

It is recommended that new Web services be developed using the Web services for Java EE specification. For more information, read about implementing Web services applications.

Security cannot be directly migrated from SOAP 2.3 to the Java EE standards. After you have migrated your Web services to the Java EE standards, read about securing Web services for Version 6 applications based on WS-Security.

Follow these best practices for the most optimal migration experience:

The application server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model.

Note: Existing JAX-RPC applications wanting to use JAX-WS features must be rewritten using the JAX-WS programming model.

Redeploy existing JAX-RPC Web services after migrating to a new release of the application server

When migrating to a new release of the application server, it is recommended that you redeploy your Web services applications. You should redeploy your Web services application in the new application server environment because of possible changes to the supported levels of Web services specifications and Web services deployment descriptors in each release. To redeploy your Web service, select **Deploy Web Services** in the Install New Application wizard or use the wsdeploy command. To learn more about this process, see the deploying Web services applications onto application servers documentation.

Migrating a Version 5 Java API for XML-based remote procedure call (JAX-RPC) client that uses SOAP over Java Message Service (JMS) to invoke a Web service

A JAX-RPC client that is run on WebSphere Application Server Version 5, can use SOAP over JMS to invoke a Web service that is run on a Version 5 Application Server.

A user ID and password are not required on the target WebSphere MQ queue. After the application server is migrated to Version 6.*x*, and uses the Version 6.*x* default messaging feature, client requests can fail because basic authentication is enabled. The following error message displays when this migration problem occurs:

```
SibMessage W [:] CWSIT0009W: A client request failed in the application server with
endpoint <endpoint name> in bus <bus_name> with reason: CWSIT0016E: The user
ID null failed authentication in bus <bus_name>.
```

When the application server is migrated to Version 6.*x*, and the default messaging provider (service integration technologies) is used, and administrative and application security is enabled for the server or the cell, the service integration bus queue destination inherits the security characteristics of the server or the cell by default. If the server or the cell has basic authentication enabled, the client request fails.

The following options are available to solve this problem. The solutions are listed by the level of security that they impose:

- Disable administrative and application security on the main security panel within the administrative console. To disable administrative and application security, click **Security > Global security**. Deselect the **Enable administrative security** and **Enable application security** options.
- Modify the settings for the service integration bus that hosts the queue destination so that the bus security is disabled and the bus does not inherit security characteristics from the server or the cell. This option is equivalent to the level of security that you can configure in Version 5.
- Configure the basic authentication on each client that uses the service.

Migrating Apache SOAP Web services

See Migrating Apache SOAP Web services to Web Services for J2EE standards to learn how to migrate Apache SOAP Web services. This topic explains how to migrate Web services that were developed using Apache SOAP to Web services that are developed based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

Migrating Web services assembled with early versions of the Application Server Toolkit or Assembly Toolkit

If you are migrating your Web service or Web service components from earlier versions of the Application Server Toolkit or Assembly Toolkit, refer to the following hints and tips to improve your success:

- Secure Web services are not migrated by the J2EE Migration Wizard when Web services are migrated from J2EE 1.3 to J2EE 1.4.
- The migration of secure Web services requires manual steps.

- After the J2EE migration, the secure binding and extension files must be migrated manually to J2EE 1.4 as follows:
  1. Double click on the `webservices.xml` file to open the Web Services editor.
  2. Select the **Binding Configurations** tab to edit the binding file.
  3. Add all the necessary binding configurations under the new sections **Request Consumer Binding Configuration Details** and **Response Generator Binding Configuration Details**.
  4. Select the **Extension** tab to edit the extension file.
  5. Add all the necessary extension configurations under the new sections **Request Consumer Service Configuration Details** and **Response Generator Service Configuration Details**.
  6. Save and exit the editor.

.

WebSphere Application Server roles and goals

A description of several computing roles that members of your organization might perform when working with WebSphere Application Server.

Enterprise architect

The enterprise architect provides overall leadership for all architectural and technological matters with respect to the company's IT environment.

Solution architect

The solution architect designs and coordinates a new solution, application or component with end-to-end responsibility including both hardware and software elements.

The main goal of the solution architect is to design a solution that supports the specification set by the enterprise architect.

System administrator

The system administrator is responsible for managing systems and software, and for installing operating system upgrades and middleware products in many accounts.

The system administrator installs and configures appropriate hardware and software (including middleware) to implement the design provided by the solution architect. Additionally the system administrator monitors and maintains the configured system, modifying and removing previously configured objects as and when required.

Application developer

The application developer creates business applications.

The goal of the application developer is to develop applications that provide the business services described by the solution architect.

Planning to use Web services

You can plan to develop and implement Web services based on a variety of Java programming models.

Before you begin

Read the Web services scenario: Overview to learn about the story of a fictional online garden supply retailer named Plants by WebSphere and how this retailer incorporated the Web services concept. You can also review the Samples Gallery for Web services Samples. These Samples demonstrate enterprise beans and JavaBeans components that are available as Web services.

The Samples Gallery includes Samples that demonstrate JAX-WS-based Web services. The JAX-WS Web services Samples demonstrate the simple message exchange patterns using both synchronous and asynchronous invocation of Web services in SOAP 1.1 and SOAP 1.2 environments. The Samples are composed with Web service standards such as WS-Addressing (WS-A) , WS-Reliable Messaging (WS-RM), and WS-Secure Conversation (WS-SC), which you can use to complete a broad range of interoperability tests. The samples demonstrate the use of JavaBeans artifacts and static service endpoints and proxy-based clients. Additionally, a Sample is provided that demonstrates Message Transmission Optimization Mechanism (MTOM).

About this task

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.
You must re-write existing JAX-RPC applications if you want to take advantage of the features of the JAX-WS programming model.

Web services reflect the service-oriented architecture approach to programming. This approach is based on the idea of building applications by discovering and implementing network-available services, or by invoking the available applications to accomplish a task. Web services deliver interoperability, for example, Web services applications provide a way for components created in different programming languages to work together as if they were created using the same language. Web services rely on existing transport technologies, such as HTTP, and standard data encoding techniques, such as Extensible Markup Language (XML), for invoking the implementation.

1. Identify your goals and design Web services to fit your e-business solution. Consider what you want to accomplish by using Web services. Decide how Web services fit into your current topology, applications and programming model. Determine how the Web services process requests on the server and how the clients manage and use the Web service.
2. Design your Web services for reliability, availability, manageability and security. For example, you want your Web services to process a transaction in a reasonable time at all hours of the day and provide users with optimal security, such as authentication for buyers. Planning to use Web services to work with WebSphere Application Server helps to meet these requirements.
3. Review the standards used in developing and deploying Web services onto WebSphere Application Server. Development and deployment are based on a variety of Java programming models.
4. Decide what development and implementation tools to use. You can use a variety of manual development and implementation tasks. Whether you have an existing Web service to implement or you want to develop your own from a JavaBeans implementation or from an Enterprise JavaBeans (EJB) module, you can choose different tasks respective to your resources. You can also use assembly tools to complete development and implementation tasks.
5. Install the application server. For detailed information on installing the application server, read about installing your application serving environment.
6. Review Web services Samples.

Results

You have a design plan for implementing Web services applications into your business architecture.

Developing Web services applications with JAX-WS

You can use the Java API for XML-Based Web Services (JAX-WS) programming model to develop Web services.

Before you begin

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

About this task

To develop Web services based on the JAX-WS programming model, you can use a bottom-up development approach starting from existing JavaBeans or enterprise beans or you can use a top-down development approach starting with an existing Web Services Description Language (WSDL) file. This task describes the steps when using the bottom-up development approach.

When developing JAX-WS Web services starting from existing JavaBeans or enterprise beans, you can expose the bean as a JAX-WS Web service by using annotations. Adding the @WebService or @WebServiceProvider annotation to the bean defines the bean as a JAX-WS Web service. JAX-WS Web services can optionally use a service endpoint interface. In addition to annotating the bean and the optional service endpoint interface, you must assemble all the artifacts that the Web service requires, and deploy the resulting application into the application server environment to complete the process of enabling the bean as a Web service. Although the use of a WSDL file is considered a best practice, you are not required to package a WSDL file with your JAX-WS Web services.

Considerations when using JavaBeans

JavaBeans exposed as JAX-WS Web services are supported only over an HTTP transport.

Considerations when using enterprise beans

- The enterprise bean must be a stateless session bean.
- Enterprise beans that are exposed as JAX-WS Web services must be packaged in EJB 3.0 or higher modules.
- JAX-WS applications containing enterprise beans must be deployed with the endptEnabler command.
- JAX-WS Web services using enterprise beans are supported over an HTTP or Java Message Service (JMS) transport.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre-Java EE 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default behavior is to not scan pre-Java EE 5 modules for annotations during application installation or server startup. You can preserve compatibility with feature packs from previous releases by either setting the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a Web archive (WAR) file or EJB module or by defining the Java virtual machine custom property,

com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations documentation.

1. Set up a development environment for Web services. You do not have to set up a development environment if you are using Rational Application Developer.
2. Determine the existing JavaBeans or enterprise beans that you want to expose as a JAX-WS Web service.
3. Develop a JAX-WS service endpoint implementation with annotations.
4. Generate Java artifacts for JAX-WS applications.

   Use JAX-WS tooling to generate the necessary JAX-WS and JAXB artifacts needed for JAX-WS Web services applications when starting from JavaBeans or enterprise beans components.

   If you are developing a service implementation bean that is invoked using the HTTP transport, then the WSDL file generated by the wsgen command-line tool during this step is optional. However, if you are developing a service implementation bean that is invoked using the SOAP over JMS transport, then the WSDL file generated by the wsgen tool during this step is required in subsequent steps, and therefore, not optional.
5. (optional) Enable MTOM for JAX-WS Web services. You can use SOAP Message Transmission Optimization Mechanism (MTOM) to optimize the transmission of binary attachments, such as images or files along with Web services requests.
6. (optional) Develop and configure a webservices.xml deployment descriptor for JAX-WS applications . You can optionally use the webservices.xml deployment descriptor to augment or override application metadata that is specified in annotations within your JAX-WS Web services.
7. Complete the implementation of your Web services application.
   - For JavaBeans applications, complete the JavaBeans implementation.
   - For enterprise beans applications, complete the enterprise beans implementation.
8. (Optional) Customize URL patterns in the web.xml file. When JavaBeans are exposed as JAX-WS endpoints, you can optionally customize the URL patterns within the web.xml deployment descriptor contained in the Web archive (WAR) file.
9. Assemble the artifacts for your Web service.

   Use assembly tools provided with the application server to assemble your Java-based Web services modules.

   If you have assembled an EAR file that contains enterprise beans modules that include Web services, use the endptEnabler command-line tool or an assembly tool before deployment to produce a Web services endpoint WAR file. This tool is also used to specify whether the Web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.
10. Deploy the EAR file into the application server. You can now deploy the EAR file that has been configured and enabled for JAX-WS Web services onto the application server.

Results

You have developed a JAX-WS application.

What to do next

After you deploy the EAR file, test the Web service to make sure that the service works with the application server.

Setting up a development environment for Web services

The application server provides command-line tools to develop Web services clients and implementations that are based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. You must set up your development environment before you start developing Web services.

Before you begin

Before you can set up a Web services development environment within WebSphere Application Server, you must install WebSphere Application Server. For detailed information on installing the application server, read about installing your application server environment.

About this task

Set up a Web services development environment by completing the following actions.

1. Set up the environment.
   Run the **setupCmdLine** script from the */profile_root/<application_server>/bin* directory.
2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:
   ```
   set PATH=%WAS_PATH%;%PATH%
   ```

Results

You have set up an environment so that you can develop Web services.

What to do next

Implement Web services applications. This topic is a good starting point in learning about how to develop and implement a Java EE Web service.

Developing a JAX-WS service endpoint implementation with annotations

Java API for XML-Based Web Services (JAX-WS) supports two different service endpoint implementations types, the standard Web service endpoint interface and a new Provider interface to enable services to work at the XML message level. By using annotations on the service endpoint implementation or client, you can define the service endpoint as a Web service.

Before you begin

Set up a development environment for Web services.

About this task

This task is a required step to develop a JAX-WS Web service.

When developing a JAX-WS Web service starting from existing JavaBeans or enterprise beans, you can expose the bean as a JAX-WS Web service by using annotations. Adding the @WebService or @WebServiceProvider annotation to the bean defines the bean as a JAX-WS Web service.

JAX-WS technology enables the implementation of Web services based on both the standard service endpoint interface and a new Provider interface. JAX-WS service endpoints are similar to the endpoint implementations in the Java API for XML-based RPC (JAX-RPC) specification. Unlike JAX-RPC, the requirement for a service endpoint interface (SEI) is optional for JAX-WS Web services. JAX-WS services that do not have an associated SEI are regarded as having an implicit SEI, whereas services that have an associated SEI are regarded as having an explicit SEI. The service endpoint interfaces required by JAX-WS are also more generic than the service endpoint interfaces required by JAX-RPC. With JAX-WS, the SEI is not required to extend the `java.rmi.Remote` interface as required by the JAX-RPC specification.

The JAX-WS programming model also leverages support for annotating Java classes with metadata to define a service endpoint implementation as a Web service and define how a client can access the Web service. JAX-WS supports annotations based on the Metadata Facility for the Java Programming

Language (JSR 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.0 (JSR 224) specification, which includes Java Architecture for XML Binding (JAXB) annotations. Using annotations, the service endpoint implementation can independently describe the Web service without requiring a WSDL file. Annotations can provide all of the WSDL information necessary to configure your service endpoint implementation or Web services client. You can specify annotations on the service endpoint interface used by the client and the server, or on the server-side service implementation class.

To develop a JAX-WS Web service, you must annotate your Java class with the `javax.jws.WebService` annotation for JavaBeans or enterprise beans endpoints or the `javax.jws.WebServiceProvider` annotation for a Provider endpoint. These annotations define the Java class as a Web service endpoint. For a JAX-WS service endpoint, the service endpoint interface or service endpoint implementation is a Java interface or class, respectively, that declares the business methods provided by a particular Web service. The only methods on a JAX-WS service endpoint that can be invoked by a Web services client are the business methods that are defined in the explicit or implicit service endpoint interface.

All JAX-WS service endpoints are required to have the @WebService (`javax.jws.WebService`) annotation included on the bean class. If the service implementation bean also uses an SEI, then that endpoint interface must be referenced by the `endpointInterface` attribute on that annotation. If the service implementation bean does not use an SEI, then the service is described by the implicit SEI defined in the bean.

The JAX-WS programming model introduces the new Provider API, `javax.xml.ws.Provider`, as an alternative to service endpoint interfaces. The Provider interface supports a more messaging oriented approach to Web services. With the Provider interface, you can create a Java class that implements a simple interface to produce a generic service implementation class. The Provider interface has one method, the `invoke` method, which uses generics to control both the input and output types when working with various messages or message payloads. All Provider endpoints must be annotated with the @WebServiceProvider (`javax.xml.ws.WebServiceProvider`) annotation. A service implementation cannot specify the @WebService annotation if it implements the `javax.xml.ws.Provider` interface.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre-Java EE 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default behavior is to not scan pre-Java EE 5 modules for annotations during application installation or server startup. You can preserve compatibility with feature packs from previous releases by either setting the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a Web archive (WAR) file or EJB module or by defining the Java virtual machine custom property, com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations documentation.

1. Identify your service endpoint requirements for your Web services application.

   First determine if the service implementation is a standard JAX-WS service endpoint or a Provider endpoint. If you choose to use a JAX-WS service endpoint, then determine if you want to use an explicit SEI or if the bean has an implicit SEI.

2. Annotate the service endpoints. A Java class that implements a Web service must specify either the `javax.jws.WebService` or `javax.xml.ws.WebServiceProvider` annotation. Both annotations must not be present on a Java class. The `javax.xml.ws.WebServiceProvider` annotation is only supported on classes that implement the `javax.xml.ws.Provider` interface.
   - If you have an explicit service endpoint interface with the Java class, then use the endpointInterface parameter to specify the service endpoint interface class name in the `javax.jws.WebService` annotation. You can add the @WebMethod annotation to methods of a service endpoint interface to customize the Java-to-WSDL mappings. All public methods are considered exposed methods

regardless of whether the @WebMethod annotation is specified. It is incorrect to have an @WebMethod annotation on an service endpoint interface that contains the `exclude` attribute.

- If you have an implicit service endpoint interface and the @WebMethod annotation is not specified, all public methods are exposed including the inherited methods of parent classes that are also annotated with @WebService annotations. You can use the `exclude` parameter of the @WebMethod annotation to control which methods are exposed.
- If your Web service implementation class implements the Provider interface, use the `javax.xml.ws.WebServiceProvider` annotation on the Provider endpoint.

Results

You have defined the service endpoint implementation that represents the Web services application. See the JAX-WS annotations documentation to learn more about the supported JAX-WS annotations.

Example

Sample JavaBeans service endpoint implementation and interface

The following example illustrates a simple explicit JavaBeans service endpoint implementation and the associated service endpoint interface.

```
/** This is an excerpt from the service implementation file, EchoServicePortTypeImpl.java
package com.ibm.was.wssample.echo;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;

@javax.jws.WebService(serviceName = "EchoService", endpointInterface =
"com.ibm.was.wssample.echo.EchoServicePortType", targetNamespace="http://com/ibm/was/wssample/echo/",
portName="EchoServicePort")
public class EchoServicePortTypeImpl implements EchoServicePortType {

                public EchoServicePortTypeImpl() {
                }

                public String invoke(String obj) {
                        String str;
                        ....
                        str = obj;
                        ....

                        return str;

                }

}
/**  This is a sample EchoServicePortType.java service interface */

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.ws.*;


@WebService(name = "EchoServicePortType", targetNamespace = "http://com/ibm/was/wssample/echo/",
wsdlLocation="WEB-INF/wsdl/Echo.wsdl")
public interface EchoServicePortType {
```

```
    /** ...the method process ...*/
        @WebMethod
    @WebResult(name = "response", targetNamespace = "http://com/ibm/was/wssample/echo/")
    @RequestWrapper(localName = "invoke", targetNamespace = "http://com/ibm/was/wssample/echo/",
className = "com.ibm.was.wssample.echo.Invoke")
    @ResponseWrapper(localName = "echoStringResponse", targetNamespace =
"http://com/ibm/was/wssample/echo/", className = "com.ibm.was.wssample.echo.EchoStringResponse")
    public String invoke(
        @WebParam(name = "arg0", targetNamespace = "http://com/ibm/was/wssample/echo/")
        String arg0);

}
```

Sample Provider endpoint implementation

The following example illustrates a simple Provider service endpoint interface for a Java class.

```
package jaxws.provider.source;

import javax.xml.ws.Provider;
import javax.xml.ws.WebServiceProvider;
import javax.xml.transform.Source;

@WebServiceProvider()
public class SourceProvider implements Provider<Source> {

    public Source invoke(Source data) {
        return data;
    }
}
```

In the Provider implementation example, the `javax.xml.transform.Source` type is specified in the generic `<Source>` method. The generic `<Source>` method specifies that both the input and output types are `Source` objects. An endpoint interface that implements the Provider interface must include a `@WebServiceProvider` annotation.

What to do next

Develop Java artifacts for JAX-WS applications from JavaBeans.

Generating Java artifacts for JAX-WS applications

Use Java API for XML-Based Web Services (JAX-WS) tools to generate the necessary JAX-WS and Java Architecture for XML Binding (JAXB) Java artifacts that are needed for JAX-WS Web services applications when starting from JavaBeans or enterprise beans components.

Before you begin

To develop a Java API for XML-Based Web Services (JAX-WS) Web service application, you must first develop a service endpoint interface (SEI) implementation that explicitly or implicitly describes the SEI.

About this task

When using a bottom-up approach to develop JAX-WS Web services, use the wsgen command-line tool on the existing service endpoint implementation. The wsgen tool processes a compiled service endpoint implementation class as input and generates the following portable artifacts:

- Java Architecture for XML Binding (JAXB) classes that are required to marshal and unmarshal the message contents.
- a Web Services Description Language (WSDL) file if the optional *-wsdl* argument is specified.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

You are not required to develop a WSDL file when developing JAX-WS Web services using the bottom-up approach of starting with JavaBeans. The use of annotations provides all of the WSDL information necessary to configure the service endpoint or the client. The application server supports WSDL 1.1 documents that comply with Web Services-Interoperability (WS-I) Basic Profile 1.1 specifications and are either Document/Literal style documents or RPC/Literal style documents. Additionally, WSDL documents with bindings that declare a `USE` attribute of value `LITERAL` are supported while the value, `ENCODED`, is not supported. For WSDL documents that implement a Document/Literal wrapped pattern, a root element is declared in the XML schema and is used as an operation wrapper for a message flow. Separate wrapper element definitions exist for both the request and the response.

To ensure the wsgen command does not miss inherited methods on a service endpoint implementation bean, you must either add the @WebService annotation to the desired superclass or you can override the inherited method in the implementation class with a call to the superclass method. Implementation classes only expose methods from superclasses that are annotated with the @WebService annotation.

Although a WSDL file is typically optional when developing a JAX-WS service implementation bean, it is required if your JAX-WS endpoints are exposed using the SOAP over JMS transport and you are publishing your WSDL file. If you are developing an enterprise JavaBeans service implementation bean that is invoked using the SOAP over JMS transport, and you want to publish the WSDL so that the published WSDL file contains the fully resolved JMS endpoint URL, then have wsgen tool generate the WSDL file by specifying the *-wsdl* argument. In this scenario, you must package the WSDL file with your Web service application.

1. Locate your service endpoint implementation class file.
2. Run the wsgen command to generate the portable artifacts. The wsgen tool is located in the *app_server_root*\bin\ directory.

   (Optional) Use the following options with the wsgen command:

   - Use the **-verbose** option to see a list of generated files along with additional informational messages.
   - Use the **-keep** option to keep generated Java files.
   - Use the **-wsdl** option to generate a WSDL file. If you are developing a service implementation bean that will be invoked using the HTTP transport, then the WSDL file generated by the wsgen command-line tool during this step is optional. However, if you are developing a service implementation bean that will be invoked using the SOAP over JMS transport, then the WSDL file generated by the wsgen tool during this step is required in subsequent developing JAX-WS applications steps, so it is not optional.

   Read about wsgen to learn more about this command and additional options that you can specify.

Results

You have the required Java artifacts to create a JAX-WS Web service.

Note: The wsgen command does not differentiate the XML namespace between multiple XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

Example

The following example demonstrates how to use the wsgen command to process the service endpoint implementation class to generate JAX-WS artifacts. This example EchoService service implementation class uses an explicit JavaBeans service endpoint.

1. Copy the sample `EchoServicePortTypeImpl` service implementation class file and the associated `EchoServicePortType` service interface class file into a directory. The directory must contain a directory tree structure that corresponds to the `com.ibm.was.wssample.echo` package name for the class file.

```
/* This is a sample EchoServicePortTypeImpl.java file.    */
package com.ibm.was.wssample.echo;


@javax.jws.WebService(serviceName = "EchoService", endpointInterface =
"com.ibm.was.wssample.echo.EchoServicePortType",
targetNamespace="http://com/ibm/was/wssample/echo/",
portName="EchoServicePort")

public class EchoServicePortTypeImpl implements EchoServicePortType {

 public EchoServicePortTypeImpl() {
 }

 public String invoke(String obj) {
  System.out.println(">> JAXB Provider Service:
Request received.\n");
  String str = "Failed";
  if (obj != null) {
   try {
    str = obj;
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
  return str;

 }

}
```

```
/* This is a sample EchoServicePortType.java file.    */
package com.ibm.was.wssample.echo;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;


@WebService(name = "EchoServicePortType", targetNamespace =
"http://com/ibm/was/wssample/echo/",
 wsdlLocation="WEB-INF/wsdl/Echo.wsdl")

public interface EchoServicePortType {


    /**
     *
     * @param arg0
     * @return
```

```
     *     returns java.lang.String
     */
    @WebMethod
    @WebResult(name = "response", targetNamespace =
    "http://com/ibm/was/wssample/echo/")
    @RequestWrapper(localName = "invoke", targetNamespace =
    "http://com/ibm/was/wssample/echo/",
     className = "com.ibm.was.wssample.echo.Invoke")
    @ResponseWrapper(localName = "echoStringResponse",
     targetNamespace = "http://com/ibm/was/wssample/echo/",
     className = "com.ibm.was.wssample.echo.EchoStringResponse")
    public String invoke(
        @WebParam(name = "arg0", targetNamespace =
        "http://com/ibm/was/wssample/echo/")
        String arg0);

}
```

2. Run the wsgen command from the *app_server_root*\bin\ directory. The `-cp` option specifies the location of the service implementation class file. The `-s` option specifies the directory for the generated source files. The `-d` option specifies the directory for the generated output files. When using the `-s` or `-d` options, you must first create the directory for the generated output files.

After generating the Java artifacts using the wsgen command, the following files are generated:

```
/generated_source/com/ibm/was/wssample/echo/EchoStringResponse.java
/generated_source/com/ibm/was/wssample/echo/Invoke.java
/generated_artifacts/EchoService.wsdl
/generated_artifacts/EchoService_schema1.xsd
/generated_artifacts/com/ibm/was/wssample/echo/EchoStringResponse.class
/generated_artifacts/com/ibm/was/wssample/echo/Invoke.class
```

The EchoStringResponse.java and Invoke.java files are the generated Java class files. The compiled versions of the generated Java files are EchoStringResponse.class and Invoke.class files. The EchoService.wsdl and EchoService_schema1.xsd files are generated because the -wsdl option was specified.

What to do next

Complete the implementation of your JAX-WS Web service application.

wsgen command for JAX-WS applications

The wsgen command-line tool generates the necessary artifacts required for Java API for XML Web Services (JAX-WS) applications when starting from Java code.

When using a bottoms-up approach to develop JAX-WS Web services and you are starting from a service endpoint implementation, use the wsgen tool to generate the required JAX-WS artifacts.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

The wsgen tool accepts a properly annotated service endpoint implementation using the @WebService annotation as input and generates the following artifacts:

- any additional Java Architecture for XML Binding (JAXB) classes that are required to marshal and unmarshal the message contents.
- a WSDL file if the optional -wsdl argument is specified. The wsgen tool does not automatically generate the WSDL file.

Note: The wsgen command does not differentiate the XML namespace between multiple XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

Syntax

The command line syntax is:

Parameters

The *service_implementation_class* name is the only parameter that is required. The following parameters are optional for the wsgen command:

-classpath <path>
Specifies the location of the service implementation class.

-cp <path>
This is the same as *-classpath <path>*.

-d <directory>
Specifies where to place the generated output files.

-extension
Specifies whether to enable custom extensions for functionality not specified by the JAX-WS specification. Use of the extensions can result in applications that are not portable or do not interoperate with other implementations.

-help
Displays the help menu.

-keep
Specifies whether to keep the generated source files.

-r <directory>
This parameter is only used in conjunction with the -wsdl parameter. Specifies where to place the generated WSDL file.

-s <directory>
Specifies the directory to place the generated source files.

-verbose
Specifies to output messages about what the compiler is doing.

-version
Prints the version information. If you specify this option, only the version information will be output and normal command processing will not occur.

-wsdl [:protocol]
By default, the wsgen tool does not generate a WSDL file. This optional parameter causes wsgen to generate a WSDL file and is typically only used to enable a developer to review a WSDL file before the endpoint is deployed. The `protocol` is optional and specifies the protocol used in the `wsdl:binding`. Valid values for `protocol` are `soap 1.1` and `Xsoap 1.2`. The default value is `soap 1.1`. The `Xsoap 1.2` value is not standard and is only used in conjunction with the *-extension* option.

-servicename <name>
This parameter is only used in conjunction with the -wsdl option. Specifies a `wsdl:service` name to be generated in the WSDL file. For example,

```
-servicename "{http://mynamespace/}MyService"
```

-portname
This parameter is only used in conjunction with the -wsdl option. Specifies a `wsdl:port` name to be generated in the WSDL file. For example,

```
-portname "{http://mynamespace/}MyPort"
```

Mapping between Java language, WSDL and XML for JAX-WS applications

Data for Java API for XML Web Services (JAX-WS) applications flows as extensible Markup Language (XML). JAX-WS applications use mappings to describe the data conversion between the Java language and extensible Markup Language (XML) technologies, including XML Schema, Web Services Description Language (WSDL) and SOAP that are supported by the application server.

For Web services based on the JAX-WS programming model, mappings between the Java language and XML are specified by the JAX-WS specification and the Java Architecture for XML Binding (JAXB) specification for data bindings. JAX-WS leverages the JAXB API and tools as the binding technology for mappings between Java objects and XML documents. JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents.

The JAX-WS specification describes the mapping between Web Service Definition Language (WSDL) files and the Java language. The supported mappings include WSDL-to-Java mappings and Java-to-WSDL mappings. WSDL 1.1 is required by the JAX-WS 2.0 specification. You can use annotations to customize the mapping from Java artifacts to their associated WSDL components. Refer to the JAX-WS specification for details describing the WSDL-to-Java mappings and Java-to-WSDL mappings.

Data binding mappings used by the JAX-WS programming model are described by the JAXB specification. Refer to the JAXB specification for details that describe the JAXB mappings for the Java representation of XML content, including the default and custom bindings between XML schema to Java representations.

Enabling MTOM for JAX-WS Web services

With Java API for XML-Based Web Services (JAX-WS), you can send binary attachments such as images or files along with Web services requests. JAX-WS adds support for optimized transmission of binary data as specified by the SOAP Message Transmission Optimization Mechanism (MTOM) specification.

About this task

JAX-WS supports the use of SOAP Message Transmission Optimized Mechanism (MTOM) for sending binary attachment data. By enabling MTOM, you can send and receive binary data optimally without incurring the cost of data encoding needed to embed the binary data in an XML document.

The application server supports sending attachments using MTOM only for JAX-WS applications. This product also provides the ability to provide attachments with Web services security SOAP messages by using the new MTOM and XOP standards.

JAX-WS applications can send binary data as base64 or hexBinary encoded data contained within the XML document. However, to take advantage of the optimizations provided by MTOM, enable MTOM to

send binary base64 data as attachments contained outside the XML document. MTOM optimization is not enabled by default. JAX-WS applications require separate configuration of both the client and the server artifacts to enable MTOM support. For the server, you can enable MTOM on a JavaBeans endpoint only and not on endpoints that implement the javax.xml.ws.Provider interface.

1. Develop Java artifacts for your JAX-WS application that includes an XML schema or Web Services Description Language (WSDL) file that represents your Web services application data that includes a binary attachment.
   a. If you are starting with a WSDL file, develop Java artifacts from a WSDL file by using the **wsimport** command to generate the required JAX-WS portable artifacts.
   b. If you are starting with JavaBeans components, develop Java artifacts for JAX-WS applications and optionally generate a WSDL file using the **wsgen** command. The XML schema or WSDL file includes a `xsd:base64Binary` or `xsd:hexBinary` element definition for the binary data.
   c. You can also include the `xmime:expectedContentTypes` attribute on the element to affect the mapping by JAXB.
2. Enable MTOM on a JavaBeans endpoint.

   To enable MTOM on an endpoint, use the @MTOM (javax.xml.ws.soap.MTOM) annotation on the endpoint. The @MTOM annotation has two parameters, enabled and threshold. The enabled parameter has a boolean value and indicates if MTOM is enabled for the JAX-WS endpoint. The threshold parameter has an integer value, and it specifies the minimum size for messages that are sent using MTOM. When the message size is less than this specified integer, the message is inlined in the XML document as base64 or hexBinary data.

   Additionally, you can use the @BindingType (javax.xml.ws.BindingType) annotation on a server endpoint implementation class to specify that the endpoint supports one of the MTOM binding types so that the response messages are MTOM-enabled. The javax.xml.ws.SOAPBinding class defines two different constants, `SOAP11HTTP_MTOM_BINDING` and `SOAP12HTTP_MTOM_BINDING` that you can use for the value of the @BindingType annotation.. For example:

   ```
   // for SOAP version 1.1
   @BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING)
   // for SOAP version 1.2
   @BindingType(value = SOAPBinding.SOAP12HTTP_MTOM_BINDING)
   ```

   The presence and value of an @MTOM annotation overrides the value of the @BindingType annotation. For example, if the @BindingType indicates MTOM is enabled, but an @MTOM annotation is present with an enabled value of false, then MTOM is not enabled
3. Enable MTOM on your client using either the `javax.xml.ws.soap.SOAPBinding` or the `javax.xml.ws.soap.MTOMFeature` APIs. Enabling MTOM on the client optimizes the binary messages that are sent to the server.
   a. Enable MTOM on a Dispatch client. The following example uses SOAP version 1.1:

   ```
   - First method:  Using SOAPBinding.setMTOMEnabled()

       SOAPBinding binding = (SOAPBinding)dispatch.getBinding();
       binding.setMTOMEnabled(true);

   - Second method: Using Service.addPort where you specify:

       Service svc = Service.create(serviceName);
       svc.addPort(portName,SOAPBinding.SOAP11HTTP_MTOM_BINDING,endpointUrl);

   - Third method: Using MTOMFeature:

       MTOMFeature mtom = new MTOMFeature();
       Service svc = Service.create(serviceName);
       svc.addPort(portName, SOAPBinding.SOAP11_HTTP_BINDING, endpointUrl);
       Dispatch<Source> dsp = svc.createDispatch(portName, Source.class, Service.Mode.PAYLOAD, mtom);
   ```

   b. Enable MTOM on a Dynamic Proxy client.

```
// Create a BindingProvider bp from a proxy port.
  Service svc = Service.create(serviceName);
  MtomSample proxy = svc.getPort(portName, MtomSample.class);
  BindingProvider bp = (BindingProvider) proxy;

  //Enable MTOM using the SOAPBinding
  MtomSample proxy = svc.getPort(portName, MtomSample.class);
  BindingProvider bp = (BindingProvider) proxy;
  SOAPBinding binding = (SOAPBinding) bp.getBinding();
  binding.setMTOMEnabled(true);

  //Or, enable MTOM with the MTOMFeature
  MTOMFeature mtom = new MTOMFeature();
  MtomSample proxy = svc.getPort(portName, MtomSample.class, mtom);
```

Results

You have developed a JAX-WS Web services server and client application that optimally sends and receives binary data using MTOM.

Example

The following example illustrates enabling MTOM support on both the Web services client and server endpoint when starting with an WSDL file.

1. Locate the WSDL file containing an `xsd:base64Binary` element. The following example is a portion of a WSDL file that contains an `xsd:base64Binary` element.

```
  <types>
    ........
<xs:complexType name="ImageDepot">
       <xs:sequence>
           <xs:element name="imageData"
                       type="xs:base64Binary"
                       xmime:expectedContentTypes="image/jpeg"/>

       </xs:sequence>
    </xs:complexType>
    ........
    </types>
```

2. Run the **wsimport** command from the *app_server_root*\bin\ directory against the WSDL file to generate a set of JAX-WS portable artifacts.

   Depending on the `expectedContentTypes` value contained in the WSDL file, the JAXB artifacts generated are in the Java type as described in the following table:

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
/	javax.activation.DataHandler

3. Use the JAXB artifacts in the same manner as in any other JAX-WS application. Use these beans to send binary data from both the Dispatch and the Dynamic Proxy client APIs.
4. Enable MTOM on a Dispatch client.

```
//Create the Dispatch instance.
        JAXBContext jbc = JAXBContext.newInstance("org.apache.axis2.jaxws.sample.mtom");
        Dispatch<Object> dispatch = svc.createDispatch(portName, jbc, Service.Mode.PAYLOAD);
```

```
//Enable MTOM.
        SOAPBinding binding = (SOAPBinding) dispatch.getBinding();
        binding.setMTOMEnabled(true);
```

5. Enable MTOM on a Dynamic Proxy client.

```
//Create the Dynamic Proxy instance.
        Service svc = Service.create(serviceName);
        MtomSample proxy = svc.getPort(portName, MtomSample.class);

//Enable MTOM.
        BindingProvider bp = (BindingProvider) proxy;
        SOAPBinding binding = (SOAPBinding) bp.getBinding();
        binding.setMTOMEnabled(true);
```

   Now that you have enabled the JAX-WS client for MTOM, messages sent to the server have MTOM enabled. However, for the server to respond back to the client using MTOM, you must enable MTOM on the JavaBeans endpoint.

6. Enable MTOM on JavaBeans endpoint.

```
WebService (endpointInterface="org.apache.axis2.jaxws.sample.mtom.MtomSample")
@BindingType (SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class MtomSampleService implements MtomSample {
   ....
}
```

   The `jaxax.xml.ws.SOAPBinding` class has a static member for each of the supported binding types. Include either the `SOAP11HTTP_MTOM_BINDING` or the `SOAP12HTTP_MTOM_BINDING` as the value for the `@BindingType` annotation. This value enables all server responses to have MTOM enabled.

When you enable MTOM on the server and the client, the binary data that represents the attachment is included as a Multipurpose Internet Mail Extensions (MIME) attachment to the SOAP message. Without MTOM , the same data is encoded in the format that describes the XML schema, either base64 or hex, and included in the XML document.

This example illustrates an MTOM enabled SOAP version 1.1 message with an attachment data. The `type` and `content-type` attributes both have the value, `application/xop+xml`, which indicates that the message was successfully optimized using XML-binary Optimized packaging (XOP) when MTOM was enabled. This example demonstrates how the optimized message looks on the wire with MTOM enabled.

```
... other transport headers ...
Content-Type: multipart/related; boundary=MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812;
type="application/xop+xml"; start="
<0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>"; start-info="text/xml"; charset=UTF-8

--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: application/xop+xml; charset=UTF-8; type="text/xml";
content-transfer-encoding: binary
content-id:
   <0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>

<?xml version="1.0" encoding="UTF-8"?>
         <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
            <soapenv:Header/>
            <soapenv:Body>
               <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
                  <input>
                     <imageData>
                        <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:1.urn:uuid:0FE43E4D025F0BF3DC11582467646811@apache.org"/>
                     </imageData>
                  </input>
               </sendImage>
            </soapenv:Body>
```

```
        </soapenv:Envelope>
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: text/plain
content-transfer-encoding: binary
content-id:
        <1.urn:uuid:0FE43E4D025F0BF3DC11582467646811@apache.org>

... binary data goes here ...
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812--
```

In contrast, this example demonstrates a SOAP version 1.1 message on the wire without MTOM enabled. The attachment is included in the body of the SOAP message, and the SOAP message is not optimized.

```
... other transport headers ...
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Header/>
      <soapenv:Body>
         <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
            <input>
               <imageData>R0lGADl  ... more base64 encoded data ...  KTJk8giAAA7</imageData>
            </input>
         </sendImage>
      </soapenv:Body>
   </soapenv:Envelope>
```

For additional information, refer to the Samples Gallery which includes a Sample that demonstrates the use of MTOM with JAX-WS Web services.

Developing a webservices.xml deployment descriptor for JAX-WS applications

Deployment descriptors are standard text files, formatted using XML and packaged in a Web services application. You can optionally use the webservices.xml deployment descriptor to augment or override application metadata specified in annotations within Java API for XML-Based Web Services (JAX-WS) Web services.

About this task

Similar to Java API for XML-based RPC (JAX-RPC) Web services, you can use deployment descriptors to describe JAX-WS Web services. For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

A JAX-WS Web service requires that you annotate your Java class with the javax.jws.WebService annotation or the javax.jws.WebServiceProvider annotation for Provider endpoints. You can use server-side deployment descriptors to override corresponding attributes of the annotation or to enhance information in annotations. There is a defined relationship between the deployment descriptor elements and the @WebService and @WebServiceProvider annotations. Refer to section 5.3 in the Web Services for Java Platform, Enterprise Edition (Java EE) specification, Version 1.2 for detailed information regarding the deployment descriptor elements and the mapping to the @WebService and @WebServiceProvider annotation attributes. There are also elements in the webservice.xml deployment descriptor that map to other annotations. For example, the deployment descriptor element <protocol-binding> maps to the @BindingType annotation, and the deployment descriptor element <enable-mtom> maps to the @MTOM annotation. For more information regarding the Web services deployment descriptor elements, see section 7.1 in the Web Services for Java Platform, Enterprise Edition (Java EE) specification.

Use assembly tools to generate the webservice.xml deployment descriptor.

Results

You have deployment descriptor templates that you can use to override JAX-WS annotation attributes or specify attributes that are not defined by the annotation.

Example

In the following example, the service implementation class for a JAX-WS Web service includes the @WebService annotation:

```
@WebService(wsdlLocation="http://myhost.com/location/of/the/wsdl/ExampleService.wsdl")
```

The associated webservices.xml deployment descriptor specifies a different filename for the WSDL document as follows:

```
<webservices>
<webservice-description>
<webservice-description-name>ExampleService</webservice-description-name>
<wsdl-file>META-INF/wsdl/ExampleService.wsdl</wsdl-file>
...
</webservice-description>
</webservices>
```

The value that is specified in the deployment descriptor, META-INF/wsdl/ExampleService.wsdl, overrides the annotation value.

What to do next

Configure the webservice.xml deployment descriptor. After you configure the deployment descriptors, you must assemble the Web services application for deployment.

JAX-WS annotations

Java API for XML-Based Web Services (JAX-WS) relies on the use of annotations to specify metadata associated with Web services implementations and to simplify the development of Web services. Annotations describe how a server-side service implementation is accessed as a Web service or how a client-side Java class accesses Web services.

The JAX-WS programming standard introduces support for annotating Java classes with metadata that is used to define a service endpoint application as a Web service and how a client can access the Web service. JAX-WS supports the use of annotations based on the Metadata Facility for the Java Programming Language (Java Specification Request (JSR) 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.0 and later (JSR 224) specification which includes JAXB annotations. Using annotations from the JSR 181 standard, you can simply annotate the service implementation class or the service interface and now the application is enabled as a Web service. Using annotations within the Java source simplifies development and deployment of Web services by defining some of the additional information that is typically obtained from deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL into the source artifacts.

Use annotations to configure bindings, handler chains, set names of portType, service and other WSDL parameters. Annotations are used in mapping Java to WSDL and schema, and at runtime to control how the JAX-WS runtime processes and responds to Web service invocations.

For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file.

You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre- Java Platform, Enterprise Edition (Java EE) 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default behavior is to not scan pre-Java EE 5 modules for annotations during application installation or server startup. You can preserve backward compatibility with the feature pack in either of two ways:

- You can set the `UseWSFEP61ScanPolicy` property in the META-INF/MANIFEST.MF of a WAR file or EJB module to `true`. For example:

  ```
  Manifest-Version: 1.0
  UseWSFEP61ScanPolicy: true
  ```

  When this property is set to `true` in the module's META-INF/MANIFEST.MF file, the module is scanned for JAX-WS annotations regardless of the Java EE version of the module. The default value is `false` and when the default value is in effect, JAX-WS annotations are only supported in modules whose version is Java EE 5 or later.
- You can set the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` custom Java virtual machine (JVM) property using the administrative console. See the JVM custom properties documentation for the correct navigation path to use. To request annotation scanning in all modules regardless of their Java EE version, set the custom property `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` to `true`. You must change the setting on each server that requires a change in the default behavior.

If the property is set within a module's META-INF/MANIFEST.MF file, this setting takes precedence over the server's custom JVM property. When using either property, you must establish the desired annotation scanning behavior before the application is installed. You cannot dynamically change the scanning behavior once an application is installed. If changes to the behavior are required after your application is installed, you must first uninstall the application, specify the desired scanning behavior using the appropriate property and then install the application again. When federating nodes that have the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` set to `true` in the configuration of the servers contained within the node, this property does not affect the deployment manager. You must set the property to `true` on the deployment manager before the node is federated to preserve the behavior as it was on the node before federation.

Note: If this JVM property is being used on the z/OS platform, it must be set in both the servant and control regions of the server.

Note: When federating nodes that have the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` set to `true` in the configuration of the servers contained within the node, this does not affect the deployment manager. You must set the property to `true` on the deployment manager before the node is federated if you wish to preserve the behavior as it was on the node before federation.

Annotations supported by JAX-WS are listed in the table below. The target for annotations is applicable for these Java objects:

- types such as a Java class, enum or interface
- methods
- fields representing local instance variables within a Java class
- parameters within a Java method

Web Services Metadata Annotations (JSR 181)

Annotation class	Annotation	Properties
`javax.jws.WebService`	The **@WebService** annotation marks a Java class as implementing a Web service or marks a service endpoint interface (SEI) as implementing a Web service interface. **Important:** • A Java class that implements a Web service must specify either the @WebService or @WebServiceProvider annotation. Both annotations cannot be present. This annotation is applicable on a client or server SEI or a server endpoint implementation class. • If the annotation references an SEI through the `endpointInterface` attribute, the SEI must also be annotated with the @WebService annotation. • See “Rules for methods on classes annotated with @WebService” on page 467 for additional information.	• Annotation target: Type • Properties: **- name** The name of the `wsdl:portType`. The default value is the unqualified name of the Java class or interface. (String) **- targetNamespace** Specifies the XML namespace of the WSDL and XML elements generated from the Web service. The default value is the namespace mapped from the package name containing the Web service. (String) **- serviceName** Specifies the service name of the Web service: `wsdl:service`. The default value is the simple name of the Java class + *Service*. (String) **- endpointInterface** Specifies the qualified name of the service endpoint interface that defines the services’ abstract Web service contract. If specified, the service endpoint interface is used to determine the abstract WSDL contract. (String) **- portName** The `wsdl:portName`. The default value is WebService.name +*Port* . (String) **- wsdlLocation** Specifies the Web address of the WSDL document defining the Web service. The Web address is either relative or absolute. (String)

Annotation class	Annotation	Properties
`javax.jws.` `WebMethod`	The **@WebMethod** annotation denotes a method that is a Web service operation. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class. **Important:** • The @WebMethod annotation is only supported on classes that are annotated with the @WebService annotation.	• Annotation target: Method • Properties: **- operationName** Specifies the name of the `wsdl:operation` matching this method. The default value is the name of Java method. (String) **- action** Defines the action for this operation. For SOAP bindings, this value determines the value of the SOAPAction header. The default value is the name of Java method. (String) **- exclude** Specifies whether to exclude a method from the Web service. The default value is `false`. (Boolean)
`javax.jws.` `Oneway`	The **@Oneway** annotation denotes a method as a Web service one-way operation that only has an input message and no output message. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • There are no properties on the Oneway annotation.

Annotation class	Annotation	Properties
`javax.jws.` `WebParam`	The **@WebParam** annotation customizes the mapping of an individual parameter to a Web service message part and XML element. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Parameter • Properties: **- name** The name of the parameter. If the operation is remote procedure call (RPC) style and the `partName` attribute is not specified, then this is the name of the `wsdl:part` attribute representing the parameter. If the operation is document style or the parameter maps to a header, then `-name` is the local name of the XML element representing the parameter. This attribute is required if the operation is document style, the parameter style is `BARE`, and the mode is `OUT` or `INOUT`. (String) **- partName** Defines the name of `wsdl:part` attribute representing this parameter. This is only used if the operation is RPC style, or the operation is document style and the parameter style is `BARE`. (String) **- targetNamespace** Specifies the XML namespace of the XML element for the parameter. Applies only for document bindings when the attribute maps to an XML element. The default value is the `targetNamespace` for the Web service. (String) **- mode** The value represents the direction the parameter flows for this method. Valid values are `IN`, `INOUT`, and `OUT`. (String) **- header** Specifies whether the parameter is in a message header rather than a message body. The default value is `false`. (Boolean)

Annotation class	Annotation	Properties
`javax.jws.` `WebResult`	The **@WebResult** annotation customizes the mapping of a return value to a WSDL part or XML element. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- name** Specifies the name of the return value as it is listed in the WSDL file and found in messages on the wire. For RPC bindings, this is the name of the `wsdl:part` attribute representing the return value. For document bindings, the `-name` parameter is the local name of the XML element representing the return value. The default value is `return` for RPC and DOCUMENT/WRAPPED bindings. The default value is the method name + `Response` for DOCUMENT/BARE bindings. (String) **- targetNamespace** Specifies the XML namespace for the return value. This parameter is only used if the operation is RPC style or if the operation is DOCUMENT style and the parameter style is BARE. (String) **- header** Specifies whether the result is carried in a header. The default value is `false`. (Boolean) **- partName** Specifies the part name for the result with RPC or DOCUMENT/BARE operations. The default value is `@WebResult.name`. (String)

Annotation class	Annotation	Properties
`javax.jws.HandlerChain`	The **@HandlerChain** annotation associates the Web service with an externally defined handler chain. You can only configure the server side handler by using the @HandlerChain annotation on the Service Endpoint Interface (SEI) or the server endpoint implementation class. Use one of several ways to configure a client side handler. You can configure a client side handler by using the @HandlerChain annotation on the generated service class or SEI. Additionally, you can programmatically register your own implementation of the HandlerResolver interface on the Service, or programmatically set the handler chain on the Binding object.	• Annotation target: Type • Properties: **- file** Specifies the location of the handler chain file. The file location is either an absolute java.net.URL in external form or a relative path from the class file. (String) **- name** Specifies the name of the handler chain in the configuration file. (String)
`javax.jws.SOAPBinding`	The **@SOAPBinding** annotation specifies the mapping of the Web service onto the SOAP message protocol. Apply this annotation to a type or methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class. The method level annotation is limited in what it can specify and is only used if the `style` property is `DOCUMENT`. If the method level annotation is not specified, the **@SOAPBinding** behavior from the type is used.	• Annotation target: Type or Method • Properties: **- style** Defines encoding style for messages sent to and from the Web service. The valid values are `DOCUMENT` and `RPC`. The default value is `DOCUMENT`. (String) **- use** Defines the formatting used for messages sent to and from the Web service. The default value is `LITERAL`. `ENCODED` is not supported. (String) **- parameterStyle** Determines whether the method's parameters represent the entire message body or whether parameters are elements wrapped inside a top-level element named after the operation. Valid values are `WRAPPED` or `BARE`. You can only use the `BARE` value with `DOCUMENT` style bindings. The default value is `WRAPPED`. (String)

JAX-WS Annotations (JSR 224)

Annotation class	Annotation	Properties
`javax.xml.ws.Action`	The **@Action** annotation specifies the WS-Addressing action that is associated with a Web service operation. When you use this annotation with a particular method, and generate the corresponding WSDL document, the WS-Addressing Action extension attribute is added to the input and output elements of the WSDL operation that corresponds to that method. To add this attribute to the WSDL operation, you must also specify the @Addressing annotation on the server endpoint implementation class. If you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.	• Annotation target: Method • Properties: **- fault** Specifies the array of FaultAction for the wsdl:fault of the operation. (String) **- input** Specifies the action for thewsdl:input of the operation. (String) **- output** Specifies the action for thewsdl:output of the operation. (String)
`javax.xml.ws.BindingType`	The **@BindingType** annotation specifies the binding to use when publishing an endpoint of this type. Apply this annotation to a server endpoint implementation class. **Important:** • You can use the @BindingType annotation on the JavaBeans endpoint implementation class to enable MTOM by specifying either `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING` or `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING` as the value for the annotation.	• Annotation target: Type • Properties: **- value** Indicates the binding identifier Web address. Valid values are `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`, `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING`, and `javax.xml.ws.http.HTTPBinding.HTTP2HTTP_BINDING`. The default value is `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.FaultAction`	The **@FaultAction** annotation specifies the WS-Addressing action that is added to a fault response. This annotation must be contained within an @Action annotation. When you use this annotation with a particular method, the WS-Addressing FaultAction extension attribute is added to the fault element of the WSDL operation that corresponds to that method. To add this attribute to the WSDL operation, you must also specify the @Addressing annotation on the server endpoint implementation class. If you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.	• Annotation target: Method • Properties: **- value** Specifies the action of the wsdl:fault of the operation. (String) **- output** Specifies the name of the exception class. (String) **- className** Specifies the name of the class representing the request wrapper. (String)
`javax.xml.ws.RequestWrapper`	The **@RequestWrapper** annotation supplies the JAXB generated request wrapper bean, the element name, and the namespace for serialization and deserialization with the request wrapper bean that is used at runtime. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the `className` attribute is required in this case. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- localName** Specifies the local name of the XML schema element representing the request wrapper. The default value is the `operationName` as defined in javax.jws.WebMethod annotation. (String) **- targetNamespace** Specifies the XML namespace of the request wrapper method. The default value is the target namespace of the SEI. (String) **- className** Specifies the name of the class representing the request wrapper. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.ResponseWrapper`	The **@ResponseWrapper** annotation supplies the JAXB generated response wrapper bean, the element name, and the namespace for serialization and deserialization with the response wrapper bean that is used at runtime. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the `className` attribute is required in this case. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- localName** Specifies the local name of the XML schema element representing the request wrapper. The default value is the `operationName` + *Response*. `operationName` is defined in javax.jws.WebMethod annotation. (String) **- targetNamespace** Specifies the XML namespace of the request wrapper method. The default value is the target namespace of the SEI. (String) **- className** Specifies the name of the class representing the response wrapper. (String)
`javax.xml.ws.RespectBinding`	The **@RespectBinding** annotation specifies whether the JAX-WS implementation must use the contents of the wsdl:binding for an endpoint. When this annotation is specified, a check is performed to ensure all required WSDL extensibility elements with the enabled attribute set to true are supported. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Method • Properties: **- enabled** Specifies whether the wsdl:binding must be used or not. The default value is `true`. (Boolean)
`javax.xml.ws.ServiceMode`	The **@ServiceMode** annotation specifies whether a service provider needs to have access to an entire protocol message or just the message payload. **Important:** • The @ServiceMode annotation is only supported on classes that are annotated with the @WebServiceProvider annotation.	• Annotation target: Type • Properties: **- value** Indicates whether the provider class accepts the payload of the message, `PAYLOAD` or the entire message `MESSAGE`. The default value is `PAYLOAD`. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.soap.Addressing`	The **@Addressing** annotation specifies that this service wants to enable WS-Addressing support. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Type • Properties: **- enabled** Specifies if WS-Addressing is enabled or not. The default value is `true`. (Boolean) **- required** Specifies that WS-Addressing headers must be present on incoming messages. The default value is `false`. (Boolean)
`javax.xml.ws.soap.MTOM`	The **@MTOM** annotation specifies whether binary content in the body of a SOAP message is sent using MTOM. Apply this annotation to a service endpoint implementation class.	• Annotation target: Class • Properties: **- enabled** Specifies if MTOM is enabled for the JAX-WS endpoint. The default value is `true`. (Boolean) **- threshold** Specifies the minimum size for messages that are sent using MTOM. When the message size is less than this specified integer, the message is inlined in the XML document as base64 or hexBinary data. (integer)

Annotation class	Annotation	Properties
`javax.xml.ws.WebFault`	The **@WebFault** annotation maps WSDL faults to Java exceptions. It is used to capture the name of the fault during the serialization of the JAXB type that is generated from a global element referenced by a WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults. This annotation can only be applied to a fault implementation class on the client or server.	• Annotation target: Type • Properties: **- name** Specifies the local name of the XML element that represents the corresponding fault in the WSDL file. The actual value must be specified. (String) **- targetNamespace** Specifies the namespace of the XML element that represents the corresponding fault in the WSDL file. (String) **- faultBean** Specifies the name of the fault bean class. (String)
`javax.xml.ws.WebServiceProvider`	The **@WebServiceProvider** annotation denotes that a class satisfies requirements for a JAX-WS Provider implementation class. **Important:** • A Java class that implements a Web service must specify either the @WebService or @WebServiceProvider annotation. Both annotations cannot be present. • The @WebServiceProvider annotation is only supported on the service implementation class. Any class with the @WebServiceProvider annotation must implement the javax.xml.ws.Provider interface.	• Annotation target: Type • Properties: **- targetNamespace** Specifies the XML namespace of the WSDL and XML elements generated from the Web service. The default value is the namespace mapped from the package name containing the Web service. (String) **- serviceName** Specifies the service name of the Web service: `wsdl:service`. The default value is the simple name of the Java class + *Service*. (String) **- portName** The `wsdl:portName`. The default value is the name of the class + *Port*. (String) **- wsdlLocation** The Web address of the WSDL document defining the Web service. This attribute is required. (String)

Annotation class	Annotation	Properties
`javax.xml.ws.WebServiceRef`	The **@WebServiceRef** annotation defines a reference to a Web service invoked by the client. **Important:** • The @WebServiceRef annotation can be used to inject instances of JAX-WS services and ports. • The @WebServiceRef annotation is only supported in certain class types. Examples are JAX-WS endpoint implementation classes, JAX-WS handler classes, Enterprise JavaBeans classes, and servlet classes. This annotation is supported in the same class types as the @Resource annotation. See the Java Platform, Enterprise Edition (Java EE) 5 specification for a complete list of supported class types.	• Annotation target: Type, Field or Method • Properties: **- name** Specifies the JNDI name of the resource. The field name is the default for field annotations. The JavaBeans property name that corresponds to the method is the default for method annotations. You must specify a value for class annotations as there is no default. (String) **- type** Indicates the Java type of the resource. The field type is the default for field annotations. The type of the JavaBeans property is the default for method annotations. You must specify a value for class annotations as there is no default. (Class) **- mappedName** Specified the name to map this resource to. (String) **- value** Indicates the value of the service class and it is a type that extends `javax.xml.ws.Service`. This attribute is required when the type of the reference is a service endpoint interface. (Class) **- wsdlLocation** The Web address of the WSDL document defining the Web service. This attribute is required. (String)
`javax.xml.ws.WebServiceRefs`	The **@WebServiceRefs** annotation associates multiple @WebServiceRef annotations with a specific class. **Important:** • The @WebServiceRef annotation is only supported in certain class types. Examples are JAX-WS endpoint implementation classes, JAX-WS handler classes, Enterprise JavaBeans classes, and servlet classes. This annotation is supported in the same class types as the @Resource annotation. See the Java Platform, Enterprise Edition (Java EE) 5 specification for a complete list of supported class types.	• Annotation target: Type • Properties: **- value** Specifies an array for multiple Web service reference declarations. This attribute is required.

JAX-WS Common Annotations (JSR 250)

Annotation class	Annotation	Properties
`javax.annotation.Resource`	The **@Resource** annotation marks a WebServiceContext resource needed by the application. **Important:** Applying this annotation to a WebServiceContext type field on the server endpoint implementation class for a JavaBeans endpoint or a Provider endpoint results in the container injecting an instance of the WebServiceContext into the specified field. When this annotation is used in place of the @WebServiceRef annotation, the rules described for the @WebServiceRef annotation apply.	• Annotation target: Field or Method • Properties: **- type** Indicates the Java type of the resource. You are required to use the default, `java.lang.Object` or `javax.xml.ws.Web ServiceContext` value. If the type is the default, the resource must be injected into a field or a method. In this case, the type of the field or the type of the JavaBeans property defined by the method must be `javax.xml.ws.WebServiceContext`. (Class) If you are using this annotation to inject a Web service, see the description of the @WebServiceRef type attribute.
`javax.annotation.Resources`	The **@Resources** annotation associates multiple @Resource annotations with a specific class and serves as a container for multiple resource declarations.	• Annotation target: Field or Method • Properties: **- value** Specifies an array for multiple @Resource annotations. This attribute is required.
`javax.annotation.PostConstruct`	The **@PostConstruct** annotation marks a method that needs to run after dependency injection is performed on the class. Apply this annotation to a JAX-WS application handler, a server endpoint implementation class.	• Annotation target: Method
`javax.annotation.PreDestroy`	The **@PreDestroy** annotation marks a method that must be run when the instance is in the process of being removed by the container. Apply this annotation to a JAX-WS application handler or a server endpoint implementation class.	• Annotation target: Method

IBM proprietary annotations

Annotation class	Annotation	Properties
`com.ibm.websphere.wsaddressing.jaxws21.SubmissionAddressing`	The **@SubmissionAddressing** annotation specifies that this service wants to enable WS-Addressing support for the 2004/08 WS-Addressing specification. This annotation is part of the IBM implementation of the JAX-WS 2.1 specification. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Type • Properties: **- required** Specifies that WS-Addressing headers must be present on incoming messages. The default value is `false`. (Boolean)

Rules for methods on classes annotated with @WebService

The following rules apply for methods on classes annotated with the @WebService annotation.

- If the @WebService annotation of an implementation class references an SEI, the implementation class must not have any @WebMethod annotations.
- All public methods for an SEI are considered exposed methods regardless of whether the @WebMethod annotation is specified or not. It is incorrect to have an @WebMethod annotation on an SEI that contains the `exclude` attribute.
- For an implementation class that does not reference an SEI, if the @WebMethod annotation is specified with a value of `exclude=true`, that method is not exposed. If the @WebMethod annotation is not specified, all public methods are exposed including the inherited methods with the exception of methods inherited from java.lang.Object.

Completing the JavaBeans implementation for JAX-WS applications

After you have developed the Java artifacts necessary to develop a Java API for XML-Based Web Services (JAX-WS) Web service, you must complete the JavaBeans implementation to assemble a Web archive (WAR) file. The resulting WAR file contains the JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Develop Java artifacts for JAX-WS applications and optionally generate a WSDL file using the wsgen command-line tool. You can also optionally use deployment descriptors to augment or override binding information contained in annotations for JAX-WS Web services.

About this task

For JAX-WS applications, complete the JavaBeans implementation by writing your business application.

1. Write the JavaBeans implementation. The JavaBeans implementation is not generated by JAX-WS tooling.
2. Compile all the Java classes.

Results

You have now written your JavaBeans implementation to complete your Web service application.

.

What to do next

After completing the JavaBeans implementation, assemble your Web services application.

Completing the EJB implementation for JAX-WS applications

After you have developed the Java artifacts necessary to develop a Java API for XML-Based Web Services (JAX-WS) Web service, you must complete the Enterprise JavaBeans (EJB) implementation to assemble a Java archive (JAR) file. The resulting JAR file contains the Enterprise JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Develop Java artifacts for JAX-WS applications and optionally generate a WSDL file using the **wsgen** command-line tool. You can also optionally use deployment descriptors to augment or override binding information contained in annotations for JAX-WS Web services.

About this task

For JAX-WS applications, complete the enterprise beans implementation by writing your business application.

1. Write the enterprise beans implementation. The enterprise beans implementation is not generated by JAX-WS tooling.
2. Compile all the Java classes.

Results

You have now written your enterprise beans implementation to complete your Web service application.

What to do next

After completing the enterprise beans implementation, assemble your Web services application.

Customizing URL patterns in the web.xml file for JAX-WS applications

The web.xml file contains information about the structure and external dependencies of Web components in the module and describes how the components are used at run time. For Java API for XML-Based Web Services (JAX-WS) applications, you can customize the URL pattern in the web.xml file.

Before you begin

Complete the JavaBeans implementation.

About this task

When you package a JavaBeans-based JAX-WS application as a Web service, the Web service is contained within a Web archive (WAR) file or a WAR module within an enterprise archive (EAR) file. A JAX-WS enabled WAR file contains the following items:

- A WEB-INF/web.xml file that describes configuration and deployment information for the Web components that comprise a Web application.
- Annotated classes that implement the Web services contained in the application module including the service endpoint implementation class.
- JAXB classes
- (Optional) Web Services Description Language (WSDL) documents that describe the Web services contained in the application module.
- (Optional) XML schema file
- (Optional) utility classes
- (Optional) Web service clients

The default URL pattern is defined by the @WebService.serviceName attribute that is contained in your Web service implementation class. When the WSDL file that is associated with your service implementation class contains a single port definition, you can choose to use the default URL pattern or you can customize the URL pattern within the web.xml file. When the WSDL file that is associated with your service implementation class contains multiple port definitions within the same service definition, customized URL patterns are required. If you use the default URL pattern when the service implementation class contains multiple port definitions, then multiple service implementation classes are mapped to the same URL pattern which results in an error condition. You must edit the web.xml file and customize the URL patterns for each service definition. Each port maps to a Web service implementation class and to its own custom URL pattern. By customizing the URL pattern in the web.xml file, you correct conflicting URL pattern definitions.

If your JAX-WS application is packaged in an Enterprise JavaBeans (EJB) Java archive (JAR) file within an enterprise archive (EAR) file, you can customize the URL patterns using the endptEnabler command.

1. Determine if custom URL patterns are required or desired. Custom URL patterns are only required when the WSDL file for your JAX-WS Web service contains multiple port definitions within a single service. Otherwise, you can optionally define custom URL patterns.
2. 2. To customize the URL pattern for a service implementation class, edit the web.xml file and provide a <servlet> and corresponding <servlet-mapping> entry for each JAX-WS Web service implementation class for which a custom URL pattern is desired. You must define the <url-pattern> value within the <servlet-mapping> entry.

Results

You now have a Web services-enabled Web archive (WAR) file with customized URL patterns.

Example

Single WSDL port definition within a service implementation class

The following example illustrates the default URL pattern and how to customize the URL pattern when the WSDL file associated with the service implementation class has a single port definition.

This is an excerpt from a sample Web service implementation class.

```
package com.ibm.test;
@WebService(serviceName="EchoService", portName="SOAP11EchoServicePort")
public class EchoServiceSOAP11{
```

This is an excerpt from the WSDL file associated with the EchoServiceSOAP11 Web service implementation class:

```
<wsdl:service name="EchoService">
 <wsdl:port name="SOAP11EchoServicePort" tns:binding="..." >
  ...
 </wsdl:port>
</wsdl:service>
```

As prescribed by JSR-109, the default URL pattern in this example is constructed using the `@WebService.serviceName` attribute and the default URL pattern is `/EchoService`.

To optionally customize the URL pattern for this example, edit the web.xml file and provide a `url-pattern` entry. In this example, the customized URL pattern is now `/EchoServiceSOAP11`.

The following excerpt is from a sample web.xml file that demonstrates setting up a servlet:

```
<servlet id="...">
 <servlet-name>com.ibm.test.EchoServiceSOAP11</servlet-name>
 <servlet-class>com.ibm.test.EchoServiceSOAP11</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
 <servlet-name>com.ibm.test.EchoServiceSOAP11</servlet-name>
 <url-pattern>/EchoServiceSOAP11</url-pattern> ----->  Defining the URL pattern and
pointing it to the service implementation class.
</servlet-mapping>
```

Multiple WSDL port definitions within a service implementation class

The following example illustrates the required URL pattern customizations when the WSDL file associated with the service implementation class has multiple port definitions.

The following excerpt is from a sample Web service implementation class:

```
package com.ibm.test;
@WebService(serviceName="EchoService", portName="SOAP11EchoServicePort")
public class EchoServiceSOAP11{
 ...
}
package com.ibm.test;
@WebService(serviceName="EchoService", portName="SOAP12EchoServicePort")
public class EchoServiceSOAP12{
 ...
}
```

The following excerpt is from the WSDL file associated with the EchoServiceSOAP11 Web service implementation class. Each port in the WSDL file maps to a portName in the Web service implementation class.

```
<wsdl:service name="EchoService">
 <wsdl:port name="SOAP11EchoServicePort" tns:binding="..." >
  ...
 </wsdl:port>
 <wsdl:port name="SOAP12EchoServicePort" tns:binding="..." >
  ...
 </wsdl:port>
</wsdl:service>
```

In this scenario, because there are multiple port definitions within the WSDL file, you must customize the URL pattern by editing the web.xml file. Specify custom URL patterns for each service.

The following excerpt is from a sample web.xml file that demonstrates setting up a servlet:

```
<servlet id="...">
 <servlet-name>com.ibm.test.EchoServiceSOAP11</servlet-name>
 <servlet-class>com.ibm.test.EchoServiceSOAP11</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
 <servlet-name>com.ibm.test.EchoServiceSOAP11</servlet-name>
 <url-pattern>/EchoServiceSOAP11</url-pattern>
</servlet-mapping>

<servlet id="...">
 <servlet-name>com.ibm.test.EchoServiceSOAP12</servlet-name>
 <servlet-class>com.ibm.test.EchoServiceSOAP12</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
 <servlet-name>com.ibm.test.EchoServiceSOAP12</servlet-name>
 <url-pattern>/EchoServiceSOAP12</url-pattern>
</servlet-mapping>
```

What to do next

Assemble a WAR file that is enabled for Web services from Java code.

Developing Web services applications from existing WSDL files with JAX-WS

You can develop a Web service with an existing Web Services Description Language (WSDL) file using the Java API for XML-Based Web Services (JAX-WS) programming model.

Before you begin

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

Locate the WSDL file that defines the Web service that you want to implement. You can develop a WSDL file or obtain one from an existing Web service through e-mail, downloading or a Uniform Resource Locator (URL).

About this task

To develop Web services based on the JAX-WS programming model, you can use a bottom-up development approach starting from existing JavaBeans or enterprise beans or you can use a top-down development approach starting with an existing Web Services Description Language (WSDL) file. This task describes the steps when using the top-down development approach.

Considerations when using JavaBeans

JavaBeans exposed as JAX-WS Web services are supported only over an HTTP transport.

Considerations when using enterprise beans

- The enterprise bean must be a stateless session bean.
- Enterprise beans that are exposed as JAX-WS Web services must be packaged in EJB 3.0 or higher modules.
- JAX-WS Web service applications containing enterprise beans must be deployed with the endptEnabler command.
- JAX-WS Web services using enterprise beans are supported over an HTTP or Java Message Service (JMS) transport.

1. Set up a development environment for Web services. You do not have to set up a development environment if you are using Rational Application Developer.
2. Develop Java artifacts for JAX-WS applications using the wsimport command-line tool. The wsimport tool processes a WSDL file and generates portable Java artifacts that are used to create a Web service.
3. (optional) Enable MTOM for JAX-WS Web services. You can use SOAP Message Transmission Optimization Mechanism (MTOM) to optimize the transmission of binary attachments such as images or files along with Web services requests.
4. (optional) Develop and configure a webservices.xml deployment descriptor for JAX-WS applications . You can optionally use the webservices.xml deployment descriptor to augment or override application metadata specified in annotations within your JAX-WS Web services.
5. Complete the implementation of your Web service application.
   - For JavaBeans applications, complete the JavaBeans implementation.
   - For enterprise beans applications, complete the enterprise beans implementation.
6. (Optional) Customize URL patterns in the web.xml file. When JavaBeans are exposed as JAX-WS endpoints, you can optionally customize the URL patterns within the web.xml deployment descriptor contained in the Web archive (WAR) file.
7. Assemble the artifacts for your Web service.
8. Deploy the EAR file into the application server. You can now deploy the EAR file that has been configured and enabled for Web services onto the application server.

Results

You have created a JAX-WS Web service by starting with an existing WSDL file.

What to do next

After you deploy the EAR file, test the Web service to make sure that the service works with the application server.

Generating Java artifacts for JAX-WS applications from a WSDL file

Use JAX-WS tools to generate the Java artifacts that are needed to develop JAX-WS Web services when starting with a Web Services Description Language (WSDL) file.

Before you begin

When using a top-down development approach to developing Java API for XML-Based Web Services (JAX-WS) Web services by starting with a Web Services Description Language (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

About this task

You can use the JAX-WS tool, wsimport, to process a WSDL file and generate portable Java artifacts that are used to create a Web service. The portable Java artifacts created using the wsimport tool are:

- Service endpoint interface (SEI)
- Service class
- Exception class that is mapped from the `wsdl:fault` class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Run the wsimport command to generate the portable client artifacts. The wsimport tool is located in the *app_server_root*\bin\ directory.
(Optional) Use the following options with the wsimport command:

- Use the **-verbose** option to see a list of generated files when you run the command.
- Use the **-keep** option to keep generated Java files.
- Use the **-wsdlLocation** option to specify the location of the WSDL file.

Note: A best practice for ensuring that you produce a JAX-WS Web services client enterprise archive (EAR) file that is portable to other systems is to package the WSDL document within the application module such as a Web services client Java archive (JAR) file or a Web archive (WAR) file. You can specify a relative URI for the location of your WSDL file by using the-`wsdllocation` annotation attribute. For example, if your *MyService.wsdl* file is located in the META-INF/wsdl/ directory, then run the wsimport tool and use the `-wsdllocation` option to specify the value to be used for the location of the WSDL file. This ensures that the generated artifacts contain the correct `-wsdllocation` information needed when the application is loaded into the administrative console.

```
wsimport -keep -wsdllocation=META-INF/wsdl/MyService.wsdl
```

- Use the **-b** option if you are using WSDL or schema customizations to specify external binding files that contain your customizations.

  You can customize the bindings in your WSDL file to enable asynchronous mappings or attachments. To generate asynchronous interfaces, add the client-side only customization `enableAsyncMapping` binding declaration to the `wsdl:definitions` element or in an external binding file that is defined in the WSDL file. Use the `enableMIMEContent` binding declaration in your custom client or server binding file to enable or disable the default mime:content mapping rules. For additional information on custom binding declarations, see chapter 8 the JAX-WS specification.

Read about the wsimport command to learn more about this command and additional options that you can specify.

Results

You have the required Java artifacts to create a JAX-WS Web service. To learn more about the usage, syntax, and parameters for the wsimport command, see the wsimport command for JAX-WS applications documentation.

Example

The following example illustrates how the wsimport command is used to process the sample Ping WSDL file to generate portable artifacts.

1. Copy the following `ping.wsdl` WSDL file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 * This program can be used, run, copied, modified and distributed
 * without royalty for the purpose of developing, using, marketing, or distributing.
 -->
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://com/ibm/was/wssample/sei/ping/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="PingService"
 targetNamespace="http://com/ibm/was/wssample/sei/ping/">
 <wsdl:types>
  <xsd:schema
   targetNamespace="http://com/ibm/was/wssample/sei/ping/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">


   <xsd:element name="pingStringInput">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="pingInput" type="xsd:string" />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>
 </wsdl:types>
 <wsdl:message name="pingOperationRequest">
```

```
 <wsdl:part element="tns:pingStringInput" name="parameter" />
</wsdl:message>
<wsdl:portType name="PingServicePortType">
 <wsdl:operation name="pingOperation">
  <wsdl:input message="tns:pingOperationRequest" />

 </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="PingSOAP" type="tns:PingServicePortType">
 <soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http" />
 <wsdl:operation name="pingOperation">
  <soap:operation soapAction="pingOperation" style="document" />
  <wsdl:input>
   <soap:body use="literal" />
  </wsdl:input>
 </wsdl:operation>
</wsdl:binding>
<wsdl:service name="PingService">
 <wsdl:port binding="tns:PingSOAP" name="PingServicePort">
  <soap:address
   location="http://localhost:9080/WSSampleSei/PingService" />
 </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

2. Run the wsimport command from the *app_server_root*\bin\ directory.

After generating the template files using the wsimport command, the following files are generated:

```
com\ibm\was\wssample\sei\ping\ObjectFactory.java
com\ibm\was\wssample\sei\ping\package-info.java
com\ibm\was\wssample\sei\ping\PingServicePortType.java
com\ibm\was\wssample\sei\ping\PingStringInput.java
com\ibm\was\wssample\sei\ping\PingService.java
```

The *ObjectFactory.java* file contains factory methods for each Java content interface and Java element interface generated in the associated ping package. The *package-info.java* file takes the targetNamespace value and creates the directory structure. The *PingServicePortType.java* file is the generated service endpoint interface (SEI) class that contains the `ping` method definition. The *PinStringInput.java* file contains the JAXB generated type values which are Java classes mapped from XML schema types. The *PingService.java* file is the generated service provider class file that is used by the JAX-WS client.

What to do next

Complete the implementation of your Web service application by completing the JavaBeans or enterprise beans implementation.

wsimport command for JAX-WS applications

The wsimport command-line tool processes an existing Web Services Description Language (WSDL) file and generates the required artifacts for developing Java API for XML-Based Web Services (JAX-WS) Web service applications.

The wsimport command-line tool supports the top-down approach to developing JAX-WS Web services. When you start with an existing WSDL file, use the wsimport command-line tool to generate the required JAX-WS artifacts.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

The wsimport tool reads an existing WSDL file and generates the following artifacts:

- Service Endpoint Interface (SEI) - The SEI is the annotated Java representation of the WSDL file for the Web service. This interface is used for implementing JavaBeans endpoints or creating dynamic proxy client instances.
- `javax.xml.ws.Service` extension class - This is a generated class that extends the `javax.xml.ws.Service` class. This class is used to configure and create both dynamic proxy and dispatch instances.
- required data beans, including any Java Architecture for XML Binding (JAXB) beans that are required to model the Web service data.

You can package the generated artifacts in a Web archive (WAR) file with the WSDL file and schema documents along with the endpoint implementation to be deployed.

Note: To correctly use the wsimport tool, you must adhere to the following requirements:

- You must define all your services within the main WSDL file. Services that are defined within an imported WSDL file are not processed by the wsimport tool.
- If you run the wsimport tool on a WSDL file that implements a Document or Literal style pattern, the complexTypes elements that define the input and output types must be composed of unique names to prevent naming conflicts in the parameter list for the operation..
- If you run the wsimport tool and pass a `?wsdl` Uniform Resource Identifier (URI) as a parameter for a WSDL file, ensure that you are using the actual resolved WSDL URI. The wsimport tool correctly resolves the `?wsdl` URI, but other relative URIs that are referenced might not resolve correctly.

Syntax

The command line syntax is:

Parameters

The *WSDL_URI* is the only parameter that is required. The following parameters are optional for the wsimport command:

-b <path>
Specifies the external JAX-WS or JAXB binding files. You can specify multiple JAX-WS and JAXB binding files by using the `-b` option; however, each file must be specified with its own `-b` option.

-B <jaxbOption>
Specifies to pass this option to the JAXB schema compiler.

-catalog
Specifies the catalog file to resolve external entity references. It supports the TR9401, XCatalog, and the OASIS XML Catalog formats

-d <directory>
Specifies where to place the generated output files.

-extension
Specifies whether to accept custom extensions for functionality that are not specified by the JAX-WS specification. The use of custom extensions can result in applications that are not portable or do not interoperate with other implementations.

-help
Displays the help menu.

-httpproxy:<host>:<port>
Specifies an HTTP proxy. The default port value is 8080.

-keep
Specifies whether to keep the generated source files.

-p <package_name>
Specifies a target package with this command-line option and overrides any WSDL file and schema binding customization for the package name and the default package name algorithm defined in the JAX-WS specification.

-quiet
Specifies to suppress the wsimport output.

-s <directory>
Specifies the directory to place the generated source files.

-target <version>>
Specifies to generate code that is compliant with a specific JAX-WS specification level. Specify version 2.0 to generate code that is compliant with the JAX-WS 2.0 specification. The default value is version 2.1 this indicates to generate code that is compliant with JAX-WS 2.1 specification.

-verbose
Specifies to output messages about what the compiler is doing.

-version
Prints the version information. If you specify this option, only the version information is included in the output and normal command processing does not occur.

-wsdlLocation
Specifies the `@WebServiceClient.wsdlLocation` value.

Note: The wsimport tool does not set the`@WebsService.wsdlLocation` value either by default or when the -wsdlLocation attribute is specified. The wsimport command-line tool updates the @WebServiceClient.wsdlLocation annotation only. You can manually update the @WebService.wsdlLocation annotation with a relative URL that specifies the location of the Web Services Description Language (WSDL) file. If the @WebService.wsdlLocation annotation is present on an endpoint implementation class, then the value must be a relative URL and the WSDL document that it references must be packaged with the application.

Developing and deploying JAX-WS Web services clients

You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-Based Web Services (JAX-WS) programming model.

Before you begin

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based

annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

About this task

Developing Web services clients based on the JAX-WS programming model

Web services clients that can both access and invoke JAX-WS Web services are developed based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. The application server supports Enterprise JavaBeans (EJB) clients, Java EE application clients, JavaServer Pages (JSP) files and servlets that are based on the JAX-WS programming model. Web services clients based on the JAX-RPC specification can invoke JAX-WS-based Web services if the Web Services Description Language (WSDL) file complies with the Web Services-Interoperability (WS-I) Basic Profile.

The JAX-WS client programming model supports both the Dispatch client API and the dynamic proxy client API. The Dispatch client API is a dynamic client programming model, whereas the static client programming model for JAX-WS is the dynamic proxy client. The Dispatch and dynamic proxy clients enable both synchronous and asynchronous invocation of JAX-WS Web services. The Dispatch client API, `javax.xml.ws.Dispatch`, is an XML messaging-oriented client that is intended for advanced XML developers who prefer using XML constructs. The Dispatch API can send data in either `PAYLOAD` or `MESSAGE` mode. When using the `PAYLOAD` mode, the Dispatch client is only responsible for providing the contents of the `soap:Body` and JAX-WS includes the payload in a `soap:Envelope` element. When using the `MESSAGE` mode, the Dispatch client is responsible for providing the entire SOAP envelope. The dynamic proxy client invokes a Web service based on a service endpoint interface (SEI) that is provided. The JAX-WS dynamic proxy instances leverage the dynamic proxy function in the base Java SE Runtime Environment (JRE) 6.

To develop Web services clients based on the JAX-WS programming model, you must determine the client model that best suits the needs of your Web service application. If you want to work directly with XML rather than a Java abstraction and work with either the message structure or the message payload structure, use the Dispatch API to develop a dynamic Web service client. If you want the Web services client to invoke the service based on service endpoint interfaces with a dynamic proxy, use the Dynamic Proxy API to develop a static Web service client. After the proxies are created, the client application can invoke methods on these proxies just like a standard implementation of the service endpoint interfaces. Complete this task to develop a static Web services client starting with a WSDL file.

To invoke Web services asynchronously using a static or dynamic JAX-WS client, determine if you want to implement the callback or the polling model. Read about invoking JAX-WS Web services asynchronously for more information regarding implementing asynchronous callback or polling for Web service clients. The JAX-WS programming model for the service and client uses annotations to represent the same information that was provided in JAX-RPC client binding in a vendor-neutral manner.

Managed and unmanaged JAX-WS Web services clients

The application server supports both managed and unmanaged Web Services clients when using the JAX-WS programming model:

- Managed clients

  Web services for Java EE clients are defined by Java Specification Requirements (JSR) 109 and are managed clients because they run in a Java EE container. These clients are packaged as enterprise archive (EAR) files and contain components that act as service requesters. These components are comprised of a Java EE client application, a Web component such as a servlet or JavaServer Pages (JSP), or a session Enterprise JavaBeans (EJB). Web services managed clients use JSR 109 APIs and deployment information to look up and invoke a Web service.

  For the managed clients, you can use Java Naming and Directory Interface (JNDI) look up to perform service lookup, or you can use annotations to inject instances of a JAX-WS service or

port. Read about setting up UserName token Web services security, digital signature Web services security and Lightweight Third-Party Authentication (LTPA) token Web services security. The following code is an example of a context lookup that is JSR 109 compliant:

```
InitialContext ctx = new InitialContext();
    FredsBankService service
=(FredsBankService)ctx.lookup("java:comp/env/service/FredsBankService");
    FredsBank fredsBank = service.getFredsBankPort();
    long balance = fredsBank.getBalance();
```

You can use the @WebServiceRef or @Resource annotation to declare managed clients. The usage of these annotations results in the type specified by the annotation being bound into the JNDI namespace. When the annotations are used on a field or method, they also result in injection of a JAX-WS service or port instance. You can use these annotations instead of declaring service-ref entries in the client deployment descriptor. You can still use the client deployment descriptor to declare JAX-WS managed clients, similar to JAX-RPC managed clients. You can also use the deployment descriptor to override and augment the information specified by @WebServiceRef and @Resource annotations. Use the @WebServiceRef annotation to bind and inject a JAX-WS service or port instance. You can only use the @Resource annotation to bind and inject a JAX-WS service instance. The use of either of these annotations to declare JAX-WS managed clients is only supported in certain class types. Some of these class types include JAX-WS endpoint implementation classes, JAX-WS handler classes, enterprise bean classes, and servlet classes.

The following example uses the @WebServiceRef annotation to obtain an instance of FredsBank:

```
@WebServiceRef(name="service/FredsBankPort", value=FredsBankService.class)
FredsBank fredsBank;
```

Now within the class, the fredsBank field does not have to be initialized. You can use this field directly:

```
long balance = fredsBank.getBalance();
```

You can also use the @WebServiceRef annotation to obtain instances of JAX-WS service classes; for example:

```
@WebServiceRef(name="service/FredsBankService")
FredsBankService service;
```

Now within the class, the service field does not have to be initialized. You can use this field directly:

```
 FredsBank fredsBank = service.getFredsBankPort(); long balance = fredsBank.getBalance();
```

In addition to the @WebServiceRef annotation, you can use the @Resource annotation to obtain an instance of JAX-WS service classes; for example:

```
@Resource(name="service/FredsBankService", type=FredsBankService.class)
FredsBankService service;
```

As with the @WebServiceRef annotation, you can now use the service field without instantiation; for example:

```
FredsBank fredsBank = service.getFredsBankPort(); long balance = fredsBank.getBalance();
```

You can use the @Resource or @WebServiceRef annotations on a class. In this case, JNDI must be used to lookup the JAX-WS service or port; for example:

```
@WebServiceRef(name="service/FredsBankService", type=FredsBankService")
public class J2EEClientExample {
  ...
  ...

  public static void main(String[] args) {
   ...
   ...
  InitialContext ctx = new InitialContext();
FredsBankService service =(FredsBankService)ctx.lookup("java:comp/env/service/FredsBankService");
     FredsBank fredsBank = service.getFredsBankPort();
```

```
        long balance = fredsBank.getBalance();

    }
    ...
}
```

For more information on using the @WebServiceRef and @Resource annotations, see the specifications for JSR-109, JSR-224, JSR-250, and Java Platform Enterprise Edition 5 (Java EE 5).

As mentioned previously, when using annotations or JNDI to obtain instances of JAX-WS services and ports, do not instantiate the returned objects. Doing so results in an unmanaged client instance. The following example shows an example of incorrect usage:

```
@WebServiceRef(name="service/FredsBankService")
FredsBankService service;
service = new FredsBankService(); // client becomes unmanaged.
```

For JAX-WS managed clients that are declared by the @WebServiceRef or @Resource annotation and for clients that are declared using service-ref entries in the client deployment descriptor, you can use the administrative console to supply the endpoint URL that the client uses. This specified URL overrides the endpoint URL in the WSDL document that is used by the client. To learn more about specifying this endpoint URL, see the configuring Web services client bindings documentation.

- Unmanaged clients

  Java Platform, Standard Edition (Java SE 6) clients that use the JAX-WS runtime environment to invoke Web services and do not run in any Java EE container are known as unmanaged clients. A Web services unmanaged client is a stand-alone Java client that can directly inspect a WSDL file and formulate the calls to the Web service by using JAX-WS APIs. These clients are packaged as JAR files, which do not contain any deployment information.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre-Java EE 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default behavior is to not scan pre-Java EE 5 modules for annotations during application installation or server startup. You can preserve compatibility with feature packs from previous releases by either setting the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a Web archive (WAR) file or EJB module or by defining the Java virtual machine custom property, com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations documentation.

1. Obtain the Web Services Description Language (WSDL) document for the Web service that you want to access.

   You can locate the WSDL from the services provider through e-mail, through a Uniform Resource Locator (URL), or by looking it up in a Universal Description, Discovery and Integration (UDDI) registry.

2. Develop JAX-WS client artifacts from a WSDL file.
   a. For static JAX-WS Web services applications using the dynamic proxy API, develop client artifacts from a WSDL file using the wsimport command.
   b. (optional) Develop JAX-WS client-side deployment descriptors. You can optionally use the application-client.xml, web.xml, or ejb-jar.xml client-side deployment descriptors to augment or override binding information contained in annotations for JAX-WS Web services.

3. Complete the client implementation. Write your client application code that is used to invoke the Web service.

   See Chapter 4 of the JSR 109 specification. For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Note: If an application creates a number of threads in the JSR 109 client, the metadata (including the WebSphere Application Server configuration) is not copied to the thread, and the Global Security Handler is not called.

4. (Optional) Assemble a Web services-enabled client Java archive (JAR) file into an enterprise archive (EAR) file. Complete this step if you are developing a JAX-WS Web services client that runs in the Java EE client container.
5. (Optional) Assemble a Web services-enabled client Web archive (WAR) file into an enterprise archive (EAR) file. Complete this step if you are developing a JAX-WS Web services client that runs in the Java EE web container.
6. (Optional) Deploy the Web services client application. Complete this step to deploy a JAX-WS Web services client that runs in the Java EE client container.
7. Test the Web services-enabled client application. You can test an unmanaged client JAR file or a managed client application.

Results

You have created and tested a Web services client application.

What to do next

After you develop a Web services application client, and the client is statically bound, the service endpoint used by the implementation is the one that is identified in the WSDL file that you used during the development process. During or after installation of the Web services application, you might want to change the service endpoint. For managed clients, you can change the endpoint with the administrative console or the wsadmin scripting tool. For unmanaged JAX-WS Web services clients, you can change the endpoint from within the client application.

You can additionally consider customizing your Web services by implementing extensions to your Web services client. Some examples of these extensions include sending and receiving values in SOAP headers or sending and receiving HTTP or Java Message Service (JMS) transport headers. To learn more about these extensions, read about implementing extensions to Web services clients.

Developing a JAX-WS client from a WSDL file

Java API for XML-Based Web Services (JAX-WS) tooling supports generating Java artifacts you need to develop static JAX-WS Web services clients when starting with a Web Services Description Language (WSDL) file.

Before you begin

When you use a top-down development approach to developing JAX-WS Web services by starting with a WSDL file, you must obtain the Uniform Resource Locator (URL) for the WSDL file.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

About this task

The static client programming model for JAX-WS is the called the dynamic proxy client. The dynamic proxy client invokes a Web service based on a service endpoint interface that is provided. After you create the proxy, the client application can invoke methods on the proxy just like a standard implementation of those interfaces. For JAX-WS Web service clients using the dynamic proxy programming model, use the JAX-WS tool, wsimport, to process a WSDL file and generate portable Java artifacts that are used to create a Web service client. Create the following portable Java artifacts using the wsimport tool:

- Service endpoint interface (SEI)
- Service class
- Exception class that is mapped from the wsdl:fault class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Run the wsimport command to generate the portable client artifacts. The wsimport tool is located in the *app_server_root*\bin\ directory.
(Optional) Use the following options with the wsimport command:

- Use the **-verbose** option to see a list of generated files when you run the command.
- Use the **-keep** option to keep generated Java files.
- Use the **-wsdlLocation** option to specify the location of the WSDL file.

  Note: A best practice for ensuring that you produce a JAX-WS Web services client enterprise archive (EAR) file that is portable to other systems is to package the WSDL document within the application module such as a Web services client Java archive (JAR) file or a Web archive (WAR) file. You can specify a relative URI for the location of your WSDL file by using the`-wsdllocation` annotation attribute. For example, if your *MyService.wsdl* file is located in the META-INF/wsdl/ directory, then run the wsimport tool and use the `-wsdllocation` option to specify the value to be used for the location of the WSDL file. This ensures that the generated artifacts contain the correct `-wsdllocation` information needed when the application is loaded into the administrative console.

  ```
  wsimport -keep -wsdllocation=META-INF/wsdl/MyService.wsdl
  ```

- Use the **-b** option if you are using WSDL or schema customizations to specify external binding files that contain your customizations.

  You can customize the bindings in your WSDL file to enable asynchronous mappings or attachments. To generate asynchronous interfaces, add the client-side only customization `enableAsyncMapping` binding declaration to the `wsdl:definitions` element or in an external binding file that is defined in the WSDL file. Use the `enableMIMEContent` binding declaration in your custom client or server binding file to enable or disable the default mime:content mapping rules. For additional information on custom binding declarations, see chapter 8 the JAX-WS specification.

Read about the wsimport command to learn more about this command and additional options that you can specify.

Results

You have the generated Java artifacts to create a JAX-WS client that can invoke JAX-WS Web services. To learn more about the usage, syntax, and parameters for the wsimport command, see the wsimport command for JAX-WS applications documentation.

Example

The following example illustrates how the wsimport command is used to process the sample `ping.wsdl` file to generate portable artifacts.

1. Copy the following `ping.wsdl` file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 * This program can be used, run, copied, modified and distributed
 * without royalty for the purpose of developing, using, marketing, or distributing.
 -->
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://com/ibm/was/wssample/sei/ping/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="PingService"
 targetNamespace="http://com/ibm/was/wssample/sei/ping/">
 <wsdl:types>
  <xsd:schema
   targetNamespace="http://com/ibm/was/wssample/sei/ping/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">


   <xsd:element name="pingStringInput">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="pingInput" type="xsd:string" />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>
 </wsdl:types>
 <wsdl:message name="pingOperationRequest">
  <wsdl:part element="tns:pingStringInput" name="parameter" />
 </wsdl:message>
 <wsdl:portType name="PingServicePortType">
  <wsdl:operation name="pingOperation">
   <wsdl:input message="tns:pingOperationRequest" />

  </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="PingSOAP" type="tns:PingServicePortType">
  <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="pingOperation">
   <soap:operation soapAction="pingOperation" style="document" />
   <wsdl:input>
    <soap:body use="literal" />
   </wsdl:input>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="PingService">
  <wsdl:port binding="tns:PingSOAP" name="PingServicePort">
   <soap:address
    location="http://localhost:9080/WSSampleSei/PingService" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

2. Run the wsimport command from the *app_server_root*\bin\ directory.

After generating the template files from the wsimport command, the following files are generated:

```
com\ibm\was\wssample\sei\ping\ObjectFactory.java
com\ibm\was\wssample\sei\ping\package-info.java
com\ibm\was\wssample\sei\ping\PingServicePortType.java
com\ibm\was\wssample\sei\ping\PingService.java
com\ibm\was\wssample\sei\ping\PingStringInput.java
```

The ObjectFactory.java, PingService.java and PingServicePortType.java files are the generated Java class files to use when you package the Java artifacts with your client implementation inside a Java archive (JAR) or a Web archive (WAR) file.

What to do next

Complete the client implementation.

Developing deployment descriptors for a JAX-WS client

Deployment descriptors are standard text files, formatted using XML and packaged in a Web services application. You can optionally use the Web Services for Java Platform, Enterprise Edition (Java EE) specification (JSR 109) service reference deployment descriptor to augment or override application metadata specified in annotations within Java API for XML-Based Web Services (JAX-WS) Web services client.

Before you begin

You must first generate the Web services client artifacts from a Web Services Description Language (WSDL) file using the wsimport command.

About this task

You can add `service-ref` entries within the application-client.xml, web.xml, or ejb-jar.xml Java EE deployment descriptors. A `service-ref` entry represents a reference to a Web service that is used by a Java EE component in Web, Enterprise JavaBeans (EJB) or application client containers. A `service-ref` entry has a JNDI name that is used to lookup the service. Specifying the `service-ref` entry enables the client applications to locate the service using a JNDI lookup and you can also use these service references for resource injection.

For each `service-ref` entry found in one of the deployment descriptors, the corresponding service object is bound into the JNDI namespace and the port information is included, if specified. The JAX-WS client can now perform a JNDI lookup to retrieve either a JAX-WS service or port instance.

When binding a JAX-WS service into JNDI, use the javax.xml.ws.Service subclass that is generated by the wsimport tool as the `service-interface` value. This is the class that contains the @WebServiceClient annotation. When binding a JAX-WS port into the namespace, the `service-interface` value is still the javax.xml.ws.Service subclass generated by the wsimport tool, and the `service-ref-type` value specifies the service endpoint interface (SEI) class used by the port. The SEI class is also generated by wsimport, and it is annotated with the @WebService annotation.

1. Define the `service-ref` entry in the your application-client.xml, web.xml, or ejb-jar.xml deployment descriptors.

   For example, suppose a Web archive (WAR) file contains a WEB-INF/web.xml deployment descriptor that includes the following `service-ref` entries:

```
<service-ref>
<service-ref-name>service/ExampleService</service-ref-name>
<service-interface>com.ibm.sample.ExampleService</service-interface>
</service-ref>

<service-ref>
<service-ref-name>service/ExamplePort</service-ref-name>
<service-interface>com.ibm.sample.ExampleService</service-interface>
<service-ref-type>com.ibm.sample.ExamplePort</service-ref-type>
</service-ref>

<service-ref>
<service-ref-name>service/ExamplePortInjected</service-ref-name>
<service-interface>com.ibm.sample.ExampleService</service-interface>
```

```
<service-ref-type>com.ibm.sample.ExamplePort</service-ref-type>

<injection-target>
   <injection-target-class>com.ibm.sample.J2EEClient</injection-target-class>
   <injection-target-name>injectedPort</injection-target-name>
    </injection-target>
</service-ref>
```

In this example, `com.ibm.sample.ExampleService` is a generated JAX-WS service class and this class must be a subclass of javax.xml.ws.Service. Additionally, the `ExampleService.getPort()` method returns an instance of `com.ibm.sample.ExamplePort`.

2. Use the deployment descriptors within your Web services client application to customize your application. The following code fragments are examples of how your client application can use the `service-ref` entries within the WAR module:

```
import javax.xml.ws.Service;
import com.ibm.sample.ExampleService;
import com.ibm.sample.ExamplePort;

// Create an InitialContext object for doing JNDI lookups.
InitialContext ic = new InitialContext();

// Client obtains an instance of the generic service class using JNDI.
Service genericService =
(Service) ic.lookup("java:comp/env/service/ExampleService");

// Client obtains an instance of the generated service class using JNDI.
ExampleService exampleService =
(ExampleService) ic.lookup("java:comp/env/service/ExampleService");

// Client obtains an instance of the port using JNDI.
ExamplePort ExamplePort =
(ExamplePort) ic.lookup("java:comp/env/service/ExamplePort");

// The container injects an instance of ExamplePort based on the client deployment descriptor
private ExamplePort injectedPort;
```

Results

You can now use the service references that were defined in the deployment descriptor within your client application. Additionally, you can use deployment descriptors to augment or override information specified by @WebServiceRef or @Resource annotations.

What to do next

Complete the client implementation by writing your client application code that is used to invoke the Web service.

Developing a dynamic client using JAX-WS APIs

Java API for XML-Based Web Services (JAX-WS) provides support for the dynamic invocation of service endpoint operations.

About this task

JAX-WS provides a new dynamic Dispatch client API that is more generic and offers more flexibility than the existing Java API for XML-based RPC (JAX-RPC)-based Dynamic Invocation Interface (DII). The Dispatch client interface, javax.xml.ws.Dispatch, is an XML messaging oriented client that is intended for advanced XML developers who prefer to work at the XML level using XML constructs. To write a Dispatch client, you must have expertise with the Dispatch client APIs, the supported object types, and knowledge of the message representations for the associated Web Services Description Language (WSDL) file.

The Dispatch API can send data in either PAYLOAD or MESSAGE mode. When using the PAYLOAD mode, the Dispatch client is only responsible for providing the contents of the <soap:Body> and JAX-WS includes the input payload in a <soap:Envelope> element. When using the MESSAGE mode, the Dispatch client is responsible for providing the entire SOAP envelope.

The Dispatch client API requires application clients to construct messages or payloads as XML and requires a detailed knowledge of the message or message payload. The Dispatch client can use HTTP bindings when using Source objects, Java Architecture for XML Binding (JAXB) objects, or data source objects. The Dispatch client supports the following types of objects:

- javax.xml.transform.Source: Use Source objects to enable clients to use XML APIs directly. You can use Source objects with SOAP and HTTP bindings.
- JAXB objects: Use JAXB objects so that clients can use JAXB objects that are generated from an XML schema to create and manipulate XML with JAX-WS applications. JAXB objects can only be used with SOAP and HTTP bindings.
- javax.xml.soap.SOAPMessage: Use SOAPMessage objects so that clients can work with SOAP messages. You can only use SOAPMessage objects with SOAP version 1.1 or SOAP version 1.2 bindings.
- javax.activation.DataSource: Use DataSource objects so that clients can work with Multipurpose Internet Mail Extension (MIME) messages. Use DataSource only with HTTP bindings.

The Dispatch API uses the concept of generics that are introduced in Java SE Runtime Environment (JRE) 6. For each of the invoke() methods on the Dispatch interface, generics are used to determine the return type.

1. Determine if you want your dynamic client to send data in PAYLOAD or MESSAGE mode.
2. Create a service instance and add at least one port to it. The port carries the protocol binding and service endpoint address information.
3. Create a Dispatch<T> object using either the Service.Mode.PAYLOAD method or the Service.Mode.MESSAGE method.
4. Configure the request context properties on the javax.xml.ws.BindingProvider interface. Use the request context to specify additional properties such as enabling HTTP authentication or specifying the endpoint address.
5. Compose the client request message for the dynamic client.
6. Invoke the service endpoint with the Dispatch client either synchronously or asynchronously.
7. Process the response message from the service.

Results

You have developed a dynamic JAX-WS client using the Dispatch API. Refer to Chapter 4, section 3 of the JAX-WS 2.0 specification for more information on using a Dispatch client.

Example

The following example illustrates the steps to create a Dispatch client and invoke a sample EchoService service endpoint.

```
String endpointUrl = ...;

QName serviceName = new QName("http://com/ibm/was/wssample/echo/",
 "EchoService");
QName portName = new QName("http://com/ibm/was/wssample/echo/",
 "EchoServicePort");

/** Create a service and add at least one port to it. **/
Service service = Service.create(serviceName);
service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING, endpointUrl);
```

```
/** Create a Dispatch instance from a service.**/
Dispatch<SOAPMessage> dispatch = service.createDispatch(portName,
SOAPMessage.class, Service.Mode.MESSAGE);

/** Create SOAPMessage request. **/
// compose a request message
MessageFactory mf = MessageFactory.newInstance(SOAPConstants.SOAP_1_1_PROTOCOL);

// Create a message.  This example works with the SOAPPART.
SOAPMessage request = mf.createMessage();
SOAPPart part = request.getSOAPPart();

// Obtain the SOAPEnvelope and header and body elements.
SOAPEnvelope env = part.getEnvelope();
SOAPHeader header = env.getHeader();
SOAPBody body = env.getBody();

// Construct the message payload.
SOAPElement operation = body.addChildElement("invoke", "ns1",
 "http://com/ibm/was/wssample/echo/");
SOAPElement value = operation.addChildElement("arg0");
value.addTextNode("ping");
request.saveChanges();

/** Invoke the service endpoint. **/
SOAPMessage response = dispatch.invoke(request);

/** Process the response. **/
```

Invoking JAX-WS Web services asynchronously

Java API for XML-Based Web Services (JAX-WS) provides support for invoking Web services using an asynchronous client invocation. JAX-WS provides support for both a callback and polling model when calling Web services asynchronously. Both the callback model and the polling model are available on the Dispatch client and the Dynamic Proxy client.

Before you begin

Develop a JAX-WS Dynamic Proxy or Dispatch client. When developing Dynamic Proxy clients, after you generate the portable client artifacts from a Web Services Description Language (WSDL) file using the **wsimport** command, the generated service endpoint interface (SEI) does not have asynchronous methods included in the interface. Use JAX-WS bindings to add the asynchronous callback or polling methods on the interface for the Dynamic Proxy client. To enable asynchronous mappings, you can add the `jaxws:enableAsyncMapping` binding declaration to the WSDL file. For more information on adding binding customizations to generate an asynchronous interface, see chapter 8 of the JAX-WS specification.

Note: When you run the wsimport tool and enable asynchronous invocation through the use of the JAX-WS enableAsyncMapping binding declaration, ensure that the corresponding response message your WSDL file does not contain parts. When a response message does not contain parts, the request acts as a two-way request, but the actual response that is sent back is empty. The wsimport tool does not correctly handle a void response. To avoid this scenario, you can remove the output message from the operation which makes your operation a one-way operation or you can add a `<wsdl:part>` to your message. For more information on the usage, syntax and parameters for the wsimport tool, see the wsimport command for JAX-WS applications documentation.

About this task

An asynchronous invocation of a Web service sends a request to the service endpoint and then immediately returns control to the client program without waiting for the response to return from the service. JAX-WS asynchronous Web service clients consume Web services using either the callback

approach or the polling approach. Using a polling model, a client can issue a request and receive a response object that is polled to determine if the server has responded. When the server responds, the actual response is retrieved. Using the callback model, the client provides a callback handler to accept and process the inbound response object. The `handleResponse()` method of the handler is called when the result is available. Both the polling and callback models enable the client to focus on continuing to process work without waiting for a response to return, while providing for a more dynamic and efficient model to invoke Web services. Polling invocations are valid from Enterprise JavaBeans (EJB) clients or Java Platform, Enterprise Edition (Java EE) application clients. Callback invocations are valid only from Java EE application clients.

Using the callback asynchronous invocation model

To implement an asynchronous invocation that uses the callback model, the client provides an `AsynchHandler` callback handler to accept and process the inbound response object. The client callback handler implements the `javax.xml.ws.AsynchHandler` interface, which contains the application code that is run when an asynchronous response is received from the server. The `javax.xml.ws.AsynchHandler` interface contains the `handleResponse(java.xml.ws.Response)` method that is called after the run time has received and processed the asynchronous response from the server. The response is delivered to the callback handler in the form of a `javax.xml.ws.Response` object. The response object returns the response content when the `get()` method is called. Additionally, if an error was received, then an exception is returned to the client during that call. The response method is then invoked according to the threading model used by the executor method, `java.util.concurrent.Executor` on the client's `java.xml.ws.Service` instance that was used to create the Dynamic Proxy or Dispatch client instance. The executor is used to invoke any asynchronous callbacks registered by the application. Use the `setExecutor` and `getExecutor` methods to modify and retrieve the executor configured for your service.

Using the polling asynchronous invocation model

Using the polling model, a client can issue a request and receive a response object that can subsequently be polled to determine if the server has responded. When the server responds, the actual response can then be retrieved. The response object returns the response content when the `get()` method is called. The client receives an object of type `javax.xml.ws.Response` from the `invokeAsync` method. That Response object is used to monitor the status of the request to the server, determine when the operation has completed, and to retrieve the response results.

Using an asynchronous message exchange

By default, asynchronous client invocations do not have asynchronous behavior of the message exchange pattern on the wire. The programming model is asynchronous; however, the exchange of request or response messages with the server is not asynchronous. To use an asynchronous message exchange, the `com.ibm.websphere.webservices.use.async.mep` property must be set on the client request context with a boolean value of `true`. When this property is enabled, the messages exchanged between the client and server are different from messages exchanged synchronously. With an asynchronous exchange, the request and response messages have WS-Addressing headers added that provide additional routing information for the messages. Another major difference between asynchronous and synchronous message exchange is that the response is delivered to an asynchronous listener that then delivers that response back to the client. For asynchronous exchanges, there is no timeout that is sent to notify the client to stop listening for a response. To force the client to stop waiting for a response, issue a `Response.cancel()` method on the object returned from a polling invocation or a `Future.cancel()` method on the object returned from a callback invocation. The cancel response does not affect the server when processing a request.

1. Determine if you want to implement the callback method or the polling method for the client to asynchronously invoke the Web service.
2. (Optional) Configure the client request context. Add the `com.ibm.websphere.webservices.use.async.mep`

property to the request context to enable asynchronous messaging for the Web services client. Using this property requires that the service endpoint supports WS-Addressing which is supported by default for the application server. The following example demonstrates how to set this property:

```
Map<String, Object> rc = ((BindingProvider) port).getRequestContext();
 rc.put("com.ibm.websphere.webservices.use.async.mep", Boolean.TRUE);
```

3. To implement the asynchronous callback method, perform the following steps.
   a. Find the asynchronous callback method on the SEI or `javax.xml.ws.Dispatch` interface. For an SEI, the method name ends in *Async* and has one more parameter than the synchronous method of type `javax.xml.ws.AsyncHandler`. The `invokeAsync(Object, AsyncHandler)` method is the one that is used on the Dispatch interface.
   b. (Optional) Add the `service.setExecutor` methods to the client application. Adding the executor methods gives the client control of the scheduling methods for processing the response. You can also choose to use the `java.current.Executors` class factory to obtain packaged executors or implement your own executor class. See the JAX-WS specification for more information on using executor class methods with your client.
   c. Implement the `javax.xml.ws.AsynchHandler` interface. The `javax.xml.ws.AsynchHandler` interface only has the `handleResponse(javax.xml.ws.Response)` method. The method must contain the logic for processing the response or possibly an exception. The method is called after the client run time has received and processed the asynchronous response from the server.
   d. Invoke the asynchronous callback method with the parameter data and the callback handler.
   e. The `handleResponse(Response)` method is invoked on the callback object when the response is available. The `Response.get()` method is called within this method to deliver the response.
4. To implement the polling method,
   a. Find the asynchronous polling method on the SEI or `javax.xml.ws.Dispatch` interface. For an SEI, the method name ends in *Async* and has a return type of `javax.xml.ws.Response`. The `invokeAsync(Object)` method is used on the Dispatch interface.
   b. Invoke the asynchronous polling method with the parameter data.
   c. The client receives the object type, `javax.xml.ws.Response`, that is used to monitor the status of the request to the server. The `isDone()` method indicates whether the invocation has completed. When the `isDone()` method returns a value of `true`, call the `get()` method to retrieve the response object.
5. Use the `cancel()` method for the callback or polling method if the client needs to stop waiting for a response from the service. If the `cancel()` method is invoked by the client, the endpoint continues to process the request. However, the wait and response processing for the client is stopped.

Results

You have enabled your JAX-WS Web service client to asynchronously invoke and consume Web services. See the JAX-WS 2.0 specification for additional information regarding the asynchronous client APIs.

Example

The following example illustrates a Web service interface with methods for asynchronous requests from the client.

```
@WebService

public interface CreditRatingService {
       // Synchronous operation.
       Score getCreditScore(Customer     customer);
       // Asynchronous operation with polling.
       Response<Score> getCreditScoreAsync(Customer customer);
       // Asynchronous operation with callback.
       Future<?> getQuoteAsync(Customer customer,
              AsyncHandler<Score> handler);
}
```

Using the callback method

The callback method requires a callback handler that is shown in the following example. When using the callback procedure, after a request is made, the callback handler is responsible for handling the response. The response value is a response or possibly an exception. The `Future<?>` method represents the result of an asynchronous computation and is checked to see if the computation is complete. When you want the application to find out if the request is completed, invoke the `Future.isDone()` method. Note that the `Future.get()` method does not provide a meaningful response and is not similar to the `Response.get()` method.

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerTom,
       new AsyncHandler<Score>() {
              public void handleResponse (
                     Response<Score> response)
                 {
                    score = response.get();
                    // process the request...
                 }
    }
);
```

Using the polling method

The following example illustrates an asynchronous polling client:

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerTom);

while (!response.isDone()) {
       // Do something while we wait.
}


score = response.get();
```

Configuring a Web services client to access resources using a Web proxy

You can configure a Web services client to access resources through a Web proxy server.

About this task

You can configure a Web services client to access resources by connecting to a Web proxy server either with or without requiring authentication, just like other HTTP client applications. You can configure HTTP transport properties for a Web service acting as a client to another Web service. The HTTP transport values you configure are used at runtime. Configure the HTTP transport values in one of the following ways:

- Configure the properties using the Java Virtual Machine (JVM) custom property panel in the administrative console.
- Configure the properties using the **wsadmin** command-line tool.
- Configure the properties with an assembly tool.
- Configure the properties programmatically using the application programming model

If you want to programmatically configure the properties using the Java API XML-based Remote Procedure Call (JAX-RPC) programming model or the Java API for XML Web Services (JAX-WS) programming model, review the JAX-RPC or JAX-WS specifications.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

For Java API XML-based Remote Procedure Call (JAX-RPC) Web services, the HTTP transport values take the following precedence order with the programmatic method being the most significant:

1. values specified programmatically on the Call object
2. values defined in the deployment descriptors in each `portQNameBinding` attribute using an assembly tool
3. values defined as JVM system properties

For Java API for XML Web Services (JAX-WS) Web services, the HTTP transport values you specify in your policy set definitions take precedence over the values defined programmatically. Subsequently, the HTTP transport values you define programmatically take precedence over the values defined as JVM system properties. For JAX-WS applications, deployment descriptors are not supported. Use annotations to specify deployment information.

1. Configure the HTTP or HTTPS `proxyHost` and `proxyPort` transport properties for the Web services in one of the following ways:
   - using the Java Virtual Machine (JVM) custom property panel in the administrative console
   - using the **wsadmin** command-line tool
   - using assembly tools
   - programmatically using the application programming model

   To access the Web proxy over HTTP:
   - **http.proxyHost**
   - **http.proxyPort**

   To access the Web proxy over HTTPS:
   - **https.proxyHost**
   - **https.proxyPort**
2. If HTTP proxy authentication is required for your Web services client, then additionally configure the HTTP or HTTPS `proxyUser` and `proxyPassword` transport properties using one of the methods specified in the previous step.

   To access the Web proxy over HTTP:
   - **http.proxyUser**
   - **http.proxyPassword**

   To access the Web proxy over HTTPS:
   - **https.proxyUser**
   - **https.proxyPassword**
3. If you are specifying the HTTP or HTTPS properties programmatically, set the properties in the Stub or Call instance to configure the HTTP proxy authentication.
   a. You can set the HTTP or HTTPS properties programmatically using the following Web services constants:

```
com.ibm.wsspi.webservices.Constants.HTTP_PROXYHOST_PROPERTY
com.ibm.wsspi.webservices.Constants.HTTP_PROXYPORT_PROPERTY
com.ibm.wsspi.webservices.Constants.HTTP_PROXYUSER_PROPERTY
com.ibm.wsspi.webservices.Constants.HTTP_PROXYPASSWORD_PROPERTY

com.ibm.wsspi.webservices.Constants.HTTPS_PROXYHOST_PROPERTY
com.ibm.wsspi.webservices.Constants.HTTPS_PROXYPORT_PROPERTY
com.ibm.wsspi.webservices.Constants.HTTPS_PROXYUSER_PROPERTY
com.ibm.wsspi.webservices.Constants.HTTPS_PROXYPASSWORD_PROPERTY
```

Results

You have configured your Web services client to use a Web proxy server to access resources.

You can optionally set the `http.nonProxyHosts` property to specify the host names of machines to which requests will not be sent through the proxy server. Any requests invoked by the client application that are sent to a host whose name is contained in this property will not pass through the proxy server. This property applies for both HTTP and HTTPS connections. To learn more about the `http.nonProxyHosts` property and other HTTP properties that you can configure, read about HTTP transport custom properties for Web services applications.

Example

Configuring the HTTP proxy programmatically

The following code allows you to configure the HTTP proxy programmatically:

```
import com.ibm.wsspi.webservices.Constants
Properties prop = new Properties();
InitialContext ctx = new InitialContext(prop);
Service service = (Service)ctx.lookup("java:comp/env/service/StockQuoteService");
QName portQname = new QName("http://httpchannel.test.wsfvt.ws.ibm.com", "StockQuoteHttp");
StockQuote sq = (StockQuote)service.getPort(portQname, StockQuote.class);
((javax.xml.rpc.Stub) sq)._setProperty(Constants.HTTP_PROXYHOST_PROPERTY, "proxyHost1.ibm.com");
((javax.xml.rpc.Stub) sq)._setProperty(Constants.HTTP_PROXYPORT_PROPERTY, "80");
```

Assembling a Web services-enabled client JAR file into an EAR file

Now that you have generated your application artifacts, you need to assemble these artifacts to create an enterprise archive (EAR) file that is used in the Web services application.

Before you begin

For Java API for XML-Based Web Services (JAX-WS) Web service applications, you need the portable artifacts that are generated by the wsimport command-line tool when starting from a WSDL file to complete this task. The wsimport tool processes a WSDL file as input and generates the following portable artifacts:

- Service Endpoint Interface (SEI)
- Service class
- Exception classes that are mapped from the `wsdl:fault` class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types
- An assembled client module that contains the implementation, all of the classes generated by the wsimport command-line tool and the `ejb-jar.xml` deployment descriptor or the `application-client.xml` deployment descriptor. This module can be:
  - An application client module that contains the `META-INF/application-client.xml` file.
  - An Enterprise JavaBeans (EJB) module that contains the `META-INF/ejb-jar.xml` file.

For Java API for XML-based RPC (JAX-RPC) Web service applications, you need the following artifacts that are generated from the WSDL2Java command-line tool to complete this task:

- An assembled client module that contains the implementation, all of the classes generated by the WSDL2Java command-line tool and the `ejb-jar.xml` deployment descriptor or the `application-client.xml` deployment descriptor. This module can be:
  - An application client module that contains the `META-INF/application-client.xml` file.
  - An Enterprise JavaBeans (EJB) module that contains the `META-INF/ejb-jar.xml` file.
- The WSDL file that you used to develop the client.
- The templates for the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` deployment descriptor, if used.

- A generated Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptor.

About this task

You can use assembly tools included with WebSphere Application Server to assemble Web services-enabled client applications.

Assemble the client code and artifacts that enable the application client to access a Web service with steps provided:

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Import the client implementation and the artifacts generated by the command-line tooling into the assembly tool.
4. Migrate JAR files created with the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
5. Assemble the JAR file into an enterprise archive (EAR) file using typical assembly techniques if the client runs in a container.

Results

You have assembled the artifacts required to enable the client application for Web services into an EAR file.

Example

This example of the assembly process uses the `AddressBookClient.jar` JAR file the `AddressBookClient.ear` EAR file:

```
META-INF/MANIFEST.MF
META-INF/application-client.xml
META-INF/wsdl/AddressBook.wsdl
META-INF/AddressBook_mapping.xml

com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookClient.jar` file into the `AddressBookClient.ear` file, the `AddressBookClient.ear` file contains the following files:

```
META-INF/MANIFEST.MF
AddressBookClient.jar
META-INF/application.xml
```

What to do next

For Java API for XML-Based Web Services (JAX-WS) applications, you are ready to deploy the Web services client application.

For Java API for XML-based RPC (JAX-RPC) applications, you need to configure the client deployment descriptor bindings so that the client can communicate with a Web service that is deployed on a server.

Assembling a Web services-enabled client WAR file into an EAR file

Now that you have generated your application artifacts, you need to assemble these artifacts to create an enterprise archive (EAR) file that is used in the Web services application.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

About this task

Assemble the client code and artifacts that enable the application client to access a Web service with steps provided:

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate WAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to an assembly tool. Read about importing Web archive (WAR) files using an assembly tool in the Rational Application Developer documentation.

Results

You have assembled the artifacts required to enable the client application for Web services into an EAR file.

Example

This example of the assembly process uses the `AddressBookWeb.war` WAR file and the `AddressBook.ear` EAR file:

```
WEB-INF/MANIFEST.MF
WEB-INF/web.xml
WEB-INF/wsdl/AddressBook.wsdl
WEB-INF/AddressBook_mapping.xml
WEB-INF/ibm-webservicesclient-ext.xmi (optional)
WEB-INF/ibm-webservicesclient-bnd.xmi
com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookWeb.war` file into the `AddressBook.ear` file, the `AddressBook.ear` file contains the following files:

```
META-INF/MANIFEST.MF
AddressBookWeb.war
META-INF/application.xml
```

What to do next

For Java API for XML-Based Web Services (JAX-WS) applications, you are ready to deploy the Web services client application.

For Java API for XML-based RPC (JAX-RPC) applications, you need to configure the client deployment descriptor bindings so that the client can communicate with a Web service that is deployed on a server.

Deploying a Web services client application

After you have created an enterprise archive (EAR) file for the Web services client application, you can deploy the Web services client application into the Application Server.

Before you begin

To deploy a Java-based Web services client, you need an enterprise application, also known as an enterprise archive (EAR) file that is configured and enabled for Web services.

A Java API for XML-Based Web Services (JAX-WS) application is packaged as a Web archive (WAR) file or a WAR module within an Enterprise Archive (EAR) file. A JAX-WS application does not require additional bindings and deployment descriptors for deployment whereas a Java API for XML-based RPC (JAX-RPC) Web services application requires you to add additional bindings and deployment descriptors for application deployment. JAX-WS is much more dynamic, and does not require any of the static data generated by the deployment step required for deploying JAX-RPC applications. For JAX-RPC Web services clients, you must configure the client deployment descriptors.

About this task

You can use either the administrative console or the **wsadmin** scripting tool to deploy an EAR file. If you are installing an application containing Web services by using the **wsadmin** command, specify the **-deployws** option for JAX-RPC applications.

Use the **wsdeploy** command only with JAX-RPC applications. The **wsdeploy** command is not applicable for JAX-WS applications.

If you are installing an application containing Web services by using the administrative console, select **Deploy WebServices** in the Install New Application wizard. Read about installing a new application for more information on using the administrative console.

The following actions deploy the EAR file with the **wsadmin** command:

1. Start *install_root*/bin/wsadmin from a command prompt.
2. Deploy the EAR file.
   - For JAX-WS Web service applications, enter the **$AdminApp install *EARfile* "-usedefaultbindings"** command at the **wsadmin** prompt.
   - For JAX-RPC Web service applications, enter the **$AdminApp install *EARfile* "-usedefaultbindings -deployws"** command at the **wsadmin** prompt.

Results

You have a deployed a Web service client in the application server runtime environment.

What to do next

Test the Web services client. You can now test a Web services-enabled managed client EAR file or an unmanaged client JAR file.

Implementing extensions to JAX-WS Web services clients

WebSphere Application Server provides extensions to Web services clients using the Java API for XML-based Web Services (JAX-WS) programming model.

About this task

You can customize Web services by using the following extensions to the JAX-WS client programming model.

- Set the **JAXWS_OUTBOUND_SOAP_HEADERS** and **JAXWS_INBOUND_SOAP_HEADERS** properties on the request context of the Dispatch or Proxy object to enable a JAX-WS Web services client to send or retrieve implicit SOAP headers.

  An implicit SOAP header is a SOAP header that is not explicitly defined in the WSDL file. An implicit SOAP header file fits one of the following descriptions:

  – A message part that is declared as a SOAP header in the binding in the WSDL file, but the message definition is not referenced by a portType within a WSDL file.

  – An element that is not contained in the WSDL file.

  Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

  See Sending implicit SOAP headers with JAX-WS or Receiving implicit SOAP headers with JAX-WS to learn how to modify your client code to send or retrieve transport headers.

- Set the **REQUEST_TRANSPORT_PROPERTIES** and **RESPONSE_TRANSPORT_PROPERTIES** properties to enable a Web services client to send or retrieve transport headers.

  Set the properties on the BindingProvider instance.

  By modifying your client code to send or retrieve transport headers, you can send or receive specific information within the transport headers of outgoing requests or incoming responses from the server. For requests or responses that use the HTTP transport, the information is sent or retrieved in an HTTP header. Similarly, for a request or response that uses the Java Message Service (JMS) transport, the information is sent or retrieved in a JMS message property.

  See “Sending transport headers with JAX-WS” on page 554 or “Retrieving transport headers with JAX-WS” on page 555 to learn how to modify your client code to send or retrieve transport headers. See “Transport header properties best practices” on page 551 to learn how to enable a Web services client to send or retrieve transport headers.

Example: Using JAX-WS properties to send and receive SOAP headers

WebSphere Application Server provides extensions to the Java API for XML-Based Web Services (JAX-WS) and Web Services for Java Platform, Enterprise Edition (Java EE) client programming models, including the com.ibm.wsspi.websvcs.Constants.JAXWS_OUTBOUND_SOAP_HEADERS and com.ibm.wsspi.websvcs.Constants.JAXWS_INBOUND_SOAP_HEADERS properties. This example demonstrates how these two properties are used.

The following programming example illustrates how to send two request SOAP headers and receive one response SOAP header within a JAX-WS Web services request and response:

```
1  //Create the hashmaps for the outbound soap headers
2  Map<QName, List<String>> outboundHeaders=new HashMap<QName, List<String>>();
3
4  //Add "AtmUuid1" and "AtmUuid2" to the outbound map
5  List<String> list1 = new ArrayList<String>();
6  list1.add("<AtmUuid1 xmlns=\"com.rotbank.security\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid1>");
7  List<String> list2 = new ArrayList<String>();
8  list2.add("<AtmUuid2 xmlns=\"com.rotbank.security\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid2>"
9  outboundHeaders.put(new QName("com.rotbank.security", "AtmUuid1"), list1);
10 outboundHeaders.put(new QName("com.rotbank.security", "AtmUuid2"), list2);
11 // Set the outbound map on the request context
12 dispatch.getRequestContext().put(Constants.JAXWS_OUTBOUND_HEADERS, outboundHeaders);
13 // Invoke the remote operation
14 dispatch.invoke(parm1);
15 // Get the inbound header map from the response context
16 Map<QName,List<String>> inboundMap = dispatch.getResponseContext().get(Constants.JAXWS_OUTBOUND_HEADERS);
17 List<String> serverUuidList = inboundMap.get(new QName("com.rotbank.security","ServerUuid"));
```

```
18 String text = serverUiidList.get(0);
19 //Note: text now equals a XML object that contains a SOAP header:
21//"<y:ServerUuid xmlns:y=\"com.rotbank.security\"><:uuid>ROTB-0A03519322FSA01
22  </y:uuid></y:ServerUuid.");
```

On line 2, create the outbound SOAP header map.

On lines 5-10, the AtmUuid1 and AtmUuid2 headers elements are added to the outbound map.

On line 12, the outbound map is set on the request context, which causes the AtmUuid1 and AtmUuid2 headers to be added to the request message when the operation is invoked.

On line 14, invoke the remote operation.

On line 15, obtain the outbound header map.

On lines 17-18, the ServerUuid header is retrieved from the response Map. The Map accesses the SOAP header from the response message.

Sending implicit SOAP headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) Web services client to send values in implicit SOAP headers. By modifying your client code to send implicit SOAP headers, you can send specific information within an outgoing Web service request.

Before you begin

To complete this task, you need a Web services client that you can enable to send implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

Using JAX-WS, there is no restriction on types of headers that you can manipulate.

About this task

The client application sets properties on the Dispatch or Proxy object to send and receive implicit SOAP headers.

1. Create a java.util.HashMap<QName, List<String>> object.
2. Add an entry to the HashMap object for each implicit SOAP header that the client wants to send. The HashMap entry key is the QName of the SOAP header. The HashMap entry value is a List<String> object, and each String is the XML text of the entire SOAP header element. By using List<String> object, you can add multiple SOAP header elements that each have the same QName object.
3. Set the HashMap object as a property on the request context of the Dispatch or Proxy object. The property name is com.ibm.wsspi.websvcs.Constants.JAXWS_OUTBOUND_SOAP_HEADERS. The value of the property is the HashMap.
4. Issue the remote method calls using the Dispatch or Proxy object. The headers within the HashMap object are sent in the outgoing message.

   A WebServiceException error can occur if any of the following are true:

- The HashMap object contains a key that is not a QName object or if the HashMap object contains a value that is not a List<String> object.
- The String representing an SOAP header is not a compliant XML message.
- The HashMap contains a key that represents a SOAP header that is declared protected by the owning component.

Results

You have a JAX-WS Web services client that is configured to send implicit SOAP headers.

Receiving implicit SOAP headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) Web services client to receive values from implicit SOAP headers. By modifying your client code to receive implicit SOAP headers, you can receive specific information within an incoming Web service response.

Before you begin

To complete this task, you need a Web services client that you can enable to receive implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

Using JAX-WS, there is no restriction on types of headers that you can manipulate.

About this task

The client application sets properties on the Dispatch or Proxy object to send and receive implicit SOAP headers.

1. Issue a remote method call with the Dispatch or Proxy object.
2. Using the property name, com.ibm.wsspi.websvcs.Constants.JAXWS_INBOUND_SOAP_HEADERS, retrieve the Map<QName, List<String>> from the ResponseContext of the Dispatch or Proxy object.
3. From the Map<QName, List<String>> value, retrieve a List<String> using the QName of the SOAP header. If the List<String> value is present, that value contains zero or more String objects that contain the XML text of the SOAP headers for the corresponding QName.

Results

You have a JAX-WS Web services client that can receive values from implicit SOAP headers.

Transport header properties best practices

You can set the REQUEST_TRANSPORT_PROPERTIES property and RESPONSE_TRANSPORT_PROPERTIES property on a Java API for XML-based RPC (JAX-RPC) client Stub, a Call instance, or a Java API for XML-Based Web Services (JAX-WS) BindingProvider instance to enable a Web services client to send or retrieve transport headers.

Note: Use these best practices to enable a Web services client to send or retrieve transport headers.

REQUEST_TRANSPORT_PROPERTIES best practices

Transport headers that require multiple values

Some transport headers such as the HTTP Cookie header and the Cookie2 header contain multiple embedded values. For headers that contain multiple values, the header value must be written in the following way:

- Each name=value pair embedded within the header value must be separated by a semi-colon (;).
- Each *name* and its value must be separated by an equal (=) sign.

The following is an example of how the header value must be written:

```
name1=value1;name2=value2;name3=value3
```

HashMap values

The HashMap values might be parsed before being added to the outgoing request if the outgoing request already contains a header identifier that matches one in the HashMap. For certain transport headers that contain multiple embedded values, the header values in the HashMap are parsed into individual name=value components. A semi-colon (;) separates the components, for example, name1=value1;name2=value2. Each name=value is appended to the outgoing header unless:

- **The outgoing request header contains a *name* value**.

  In this case, the name=value from the HashMap is silently ignored, preventing a client from overwriting or modifying values for the *name* value that are already set in the outgoing request header by either the server or the Web services engine.
- **The HashMap header value contains multiple *name* values**.

  When the HashMap header value contains multiple *name* values, the first occurrence of the *name* value is used and the others are silently ignored. For example, if the HashMap header value contains name1=value1;name2=value2;name1=value3, where there are two occurrences of name1, the first value, name1=value1, is used. The other value, name1=value3, is silently ignored.

RESPONSE_TRANSPORT_PROPERTIES best practices

HashMap values

Only the HashMap keys are used; the HashMap values are ignored. The values are filled in this HashMap by retrieving the transport headers, which correspond to the HashMap keys from the incoming response message. An empty HashMap causes all of the transport headers and the associated values to be retrieved from the incoming response message.

HTTP headers that are processed under special consideration

The following are HTTP headers that are given special consideration when sending and retrieving HTTP responses and requests.

The values in these headers can be set in a variety of ways. For example, some header values are sent based on settings in a deployment descriptor or binding file. In these cases, the value set through REQUEST_TRANSPORT_PROPERTIES overrides the values set any other way.

Header	Send request	Retrieve response

Transfer-encoding	• The transfer-encoding header is ignored for HTTP 1.0. • When using HTTP 1.1, the transfer-encoding header is set to `chunked` if the value is `chunked`.	There is no special processing.
Connection	• The connection header is ignored for HTTP 1.0. • When using HTTP 1.1, the following values are set: – The connection header is set to ″close″ if the value is set to ″close″. – The connection header is set to ″keep-alive″ if the value is set to ″keep-alive″. – All other value settings are ignored.	There is no special processing.
Expect	• The expect header is ignored for HTTP 1.0. • When using HTTP 1.1, the following values are set: – The connection header is set to ″100-continue″ if the value is set to ″100-continue″. – All other value settings are ignored.	There is no special processing.
Host	Ignored	There is no special processing.
Content-type	Ignored	There is no special processing.
SOAPAction	Ignored	There is no special processing.
Content-length	Ignored	There is no special processing.
Cookie The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_COOKIE	The value is sent on the header if it is structured correctly. See the information in this article for Header value format and HashMap values.	There is no special processing.
Cookie2 The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_COOKIE2	See the information in the ″Cookie″ entry.	There is no special processing.
Authorization	Ignored	There is no special processing.
Proxy-authorization	Ignored	There is no special processing.

Set-cookie The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_SET_COOKIE	There is no special processing.	If the property MAINTAIN_SESSION is set to `true`, the entire value is saved into SessionContext.CONTEXT_PROPERTY and is sent on subsequent requests in the Cookie header. See the Cookie entry in this table for more information.
Set-cookie2 The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_SET_COOKIE2	There is no special processing.	If the property MAINTAIN_SESSION is set to `true`, the entire value is saved into SessionContext.CONTEXT_PROPERTY and is sent on subsequent requests in the Cookie header. See the Cookie entry in this table for more information.

Example client code

The following is an example of how you can code a Web services client to send and retrieve HTTP transport header values:

```
HashMap sendTransportHeaders=new HashMap();
sendTransportHeaders.put("Cookie","ClientAuthenticationToken=FFEEBCC");
sendTransportHeaders.put("MyRequestHeader","MyRequestHeaderValue");
((Stub) portType)._setProperty(Constants.REQUEST_TRANSPORT_PROPERTIES, sendTransportHeaders);

HashMap receiveTransportHeaders=new HashMap();
receiveTransportHeaders.put("Set-Cookie", null);
receiveTransportHeaders.put("MyResponseHeader", null);
((Stub) portType)._setProperty(Constants.RESPONSE_TRANSPORT_PROPERTIES,
 receiveTransportHeaders);

resultString=portType.echoString("Foo");
```

Sending transport headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) Web services client to send application-defined information along with your Web services requests by using transport headers.

Before you begin

You need a JAX-WS Web services client that you can enable to send transport headers.

Sending transport headers is supported only by Web services clients, and only supported for the HTTP and JMS transports. The Web services client must call the JAX-WS APIs directly and not through any intermediary layers, such as a gateway function. Sending and retrieving transport headers on the Web services server is done through non-Web services APIs.

About this task

When using the JAX-WS programming model, the client must set a property on the BindingProvider object to send values in transport headers. After you set the property, the values are set in all the requests for subsequent remote method invocations against the BindingProvider object until the associated property is set to null or the BindingProvider object is discarded.

To send values in the transport headers on outbound requests, modify the client code.

1. Create a java.util.HashMap object that contains the transport header identifiers.
2. Add an entry to the HashMap object for each transport header that you want the client to send.
   a. Set the HashMap entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user defined, such as `MyTransportHeader`. Certain header identifiers

are processed in a unique manner, but no other checks are made as to the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class.

b. Set the HashMap entry value to a string that contains the value of the transport header.

3. Set the HashMap entry on the BindingProvider instance using the com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES property. When the REQUEST_TRANSPORT_PROPERTIES property value is set, that HashMap is used on subsequent invocations to set the header values in the outgoing requests. If the REQUEST_TRANSPORT_PROPERTIES property value is set to null, no HashMap is used on subsequent invocations to set header values in outgoing requests. To learn more about these properties, see the transport header properties documentation.
4. Issue remote method calls against the BindingProvider instance. The headers and the associated values from the HashMap are added to the outgoing request for each method invocation. If the invocation uses HTTP, then the transport headers are sent as HTTP headers within the HTTP request. If the invocation uses JMS, then the transport headers are sent as JMS message properties.

   If the property is not set correctly, you might experience API usage errors that result in a WebServiceException error. The following requirements must be met, or the process fails:

   - The property value that is set on the BindingProvider instance must be a HashMap object or null.
   - The HashMap must not be empty.
   - Each key in the HashMap must be a String object.
   - Each value in the HashMap must be a String object.

Results

You have a JAX-WS Web services client that is configured to send transport headers.

Retrieving transport headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) Web services client to retrieve values from transport headers. For a request that uses HTTP, the transport headers are retrieved from HTTP headers found in the HTTP response message. For a request that uses Java Message Service (JMS), the transport headers are retrieved from the JMS message properties found on the JMS response message.

Before you begin

You need a JAX-WS Web services client that you can enable to retrieve transport headers. B

Retrieving transport headers is supported only by Web services clients, and only supported for the HTTP and JMS transports. The Web services client must call the JAX-WS APIs directly and not through any intermediary layers, such as a gateway function. Sending and retrieving transport headers on the Web services server is done through non-Web services APIs.

About this task

When using the JAX-WS programming model, the client must set a property on the BindingProvider object to retrieve values from the transport headers. After you set the property, values are read from responses for the subsequent method invocations against that BindingProvider object until the associated property is set to null or the BindingProvider object is discarded.

To retrieve values from the transport headers on inbound responses, modify the client code.

1. Create a java.util.HashMap object that contains the names of the transport headers to be retrieved from incoming response messages.

2. Add an entry to the HashMap for each header that you want to retrieve a value from every incoming response message.
   a. Set the HashMap entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user-defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made to confirm the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class. The HashMap entry value is ignored and does not need to be set. An empty HashMap, for example, one that is non-null, but does not contain any keys, causes all the transport headers in the response to be retrieved.
3. Set the HashMap entry on the BindingProvider instance using the com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES property. When the HashMap is set, the RESPONSE_TRANSPORT_PROPERTIES property is used in subsequent invocations to retrieve the headers from the responses. If you set the property to null, no headers are retrieved from the response. To learn more about these properties, see the transport header properties documentation.
4. Issue remote method calls against the BindingProvider instance. The values from the specified transport headers are retrieved from the response message and placed in the HashMap.

   If the property is not set correctly, you might experience API usage errors that result in a WebServiceException error. The following requirements must be met, or the process fails:
   - The property value that is set on the BindingProvider instance must be either null or an instance of a HashMap.
   - All the HashMap keys must be a string data type, and the keys must not be null.

Results

You have a JAX-WS Web service that can receive transport headers from incoming response messages.

Using JAXB for XML data binding

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified Web services development. JAXB provides a schema compiler, schema generator and a runtime framework to support two-way mapping between Java objects and XML documents.

About this task

JAXB is an XML-to-Java binding technology that enables transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB technology consists of a runtime API and accompanying tools that simplify access to XML documents. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides tooling to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the XML data structure.

JAXB is the default data binding technology used by the Java API for XML Web Services (JAX-WS) tooling and implementation within this product. You can develop JAXB objects to use within your JAX-WS applications. You can also use JAXB independently of the JAX-WS programming model as a convenient way to leverage the XML data binding technology to manipulate XML within your Java applications.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

JAXB provides the xjc schema compiler tool, the schemagen schema generator tool, and a runtime framework. The xjc schema compiler tool enables you to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. You can also start with a set of JavaBeans and use the schemagen schema generator tool to create the XML schema. After using either the schema compiler or the schema generator command-line tools, you can convert your XML documents both to and from Java objects and use the resulting Java classes to assemble a Web services application.

In addition to using the tools from the command-line, you can invoke these JAXB tools from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the xjc schema compiler tool. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the schemagen schema generator tool.

JAXB annotated classes and artifacts contain all the information that the JAXB runtime API needs to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML files and unmarshaling the XML document back to JAXB class instances. The JAXB binding package, `javax.xml.bind`, defines the abstract classes and interfaces that are used directly with content classes. In addition the package defines the marshal and unmarshal APIs.

JAXB 2.1 provides enhancements such as improved compilation support and support for the @XMLSeeAlso annotation. With JAXB 2.1, you can configure the xjc schema compiler so that it does not automatically generate new classes for a particular schema. Similarly, you can configure the schemagen schema generator to not automatically generate a new schema. This enhancement is useful when you are using a common schema and you do not want a new schema generated. JAXB 2.1 also introduces the @XMLSeeAlso annotation that enables JAXB to bind additional Java classes that it might not otherwise know about when binding a Java class with this annotation. This annotation enables JAXB to know about all classes that are potentially involved in marshalling or unmarshalling as it is not always possible or practical to list all of the subclasses of a given Java class. JAX-WS 2.1 also supports the use of the @XMLSeeAlso annotation on a service endpoint interface (SEI) or on a service implementation bean to ensure all of the classes referenced by the annotation are passed to JAXB for processing.

You can optionally use JAXB binding customizations to customize generated JAXB classes by overriding or extending the default JAXB bindings when the default bindings do not meet your business application needs. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes. JAXB supports binding customizations and overrides to the default binding rules that you can make through various ways. For example, you can the overrides inline as annotations in a source schema, as declarations in an external bindings customization file that is used by the JAXB binding compiler, or as Java annotations within Java class files used by the JAXB schema generator. See the JAXB specification for information regarding binding customization options.

Using JAXB, you can manipulate data objects in the following ways:

- Generate an XML schema from a Java class. Use the schema generator schemagen command to generate an XML schema from Java classes.
- Generate Java classes from an XML schema. Use the schema compiler xjc command to create a set of JAXB-annotated Java classes from an XML schema.
- Marshal and unmarshal XML documents. After the mapping between XML schema and Java classes exists, use the JAXB binding runtime to convert XML instance documents to and from Java objects.

Results

You now have JAXB objects that your Java application can use to manipulate XML data.

Using JAXB tools to generate an XML schema file from a Java class

Use Java Architecture for XML Binding (JAXB) tooling to generate an XML schema file from Java classes.

Before you begin

Identify the Java classes or a set of Java objects to map to an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between Java classes and XML schema. XML schema documents describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop Web services using a bottom-up development approach starting from existing JavaBeans or enterprise beans, use the wsgen tool to generate the artifacts for Java API for XML-Based Web Services (JAX-WS) applications. After the Java artifacts for your application are generated, you can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, schemagen command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

You can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, schemagen command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding.

JAXB 2.1 provides improvements in compilation support to enable you to configure the schemagen schema generator so that it does not automatically generate a new schema. This is helpful if you are using a common schema such as the World Wide Web Consortium (W3C), XML Schema, Web Services Description Language (WSDL), or WS-Addressing and you do not want a new schema generated for a

particular package that is referenced. The location attribute on the @XmlSchema annotation causes the schemagen generator to refer to the URI of the existing schema instead of generating a new one.

In addition to using the schemagen tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the schemagen schema generator tool.

1. Locate the Java source files or Java class files to use in generating an XML schema file. Ensure that all classes referenced by your Java class files are contained in the classpath or are provided to the tool using the`-classpath/-cp` options.
2. Use the JAXB schema generator, schemagen command to generate an XML schema. The schema generator is located in the *app_server_root*\bin\ directory.

   The parameters, `myObj1.java` and `myObj2.java`, are the names of the Java files containing the data objects. If `myObj1.java` or `myObj2.java` refer to Java classes that are not passed into the schemagen command, you must use the `-cp` option to provide the classpath location for these Java classes.Read about the schemagen command to learn more about this command and additional options that you can specify.
3. (Optional) Use JAXB program annotations defined in the `javax.xml.bind.annotations` package to customize the JAXB XML schema mappings.
4. (Optional) Configure the `location` property on the @XmlSchema annotation to indicate to the schema compiler to use an existing schema rather than generating a new one. For example,

   ```
   @XmlSchema(namespace="foo")
   package foo;
   @XmlType
   class Foo {
   @XmlElement Bar zot;
   }
   @XmlSchema(namespace="bar",
   location="http://example.org/test.xsd")
   package bar;
   @XmlType
   class Bar {
   ...
   }
   <xs:schema targetNamespace="foo">
   <xs:import namespace="bar"
   schemaLocation="http://example.org/test.xsd"/>
   <xs:complexType name="foo">
   <xs:sequence>
   <xs:element name="zot" type="bar:Bar" xmlns:bar="bar"/>
   </xs:sequence>
   </xs:complex
   ```

   the `location="http://example.org/test.xsd"` indicates the location on the existing schema to the schemagen tool and a new schema is not generated.

Results

Now that you have generated an XML schema file from Java classes, you are ready to marshal and unmarshal the Java objects as XML instance documents.

Note: The **schemagen** command does not differentiate the XML namespace between multiple @XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

Example

The following example illustrates how JAXB tooling can generate an XML schema file from an existing Java class, Bookdata.java.

1. Copy the following `Bookdata.java` file to a temporary directory.

```
package generated;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.datatype.XMLGregorianCalendar;



@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "bookdata", propOrder = {
    "author",
    "title",
    "genre",
    "price",
    "publishDate",
    "description"
})
public class Bookdata {

    @XmlElement(required = true)
    protected String author;
    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String genre;
    protected float price;
    @XmlElement(name = "publish_date", required = true)
    protected XMLGregorianCalendar publishDate;
    @XmlElement(required = true)
    protected String description;
    @XmlAttribute
    protected String id;

     public String getAuthor() {
        return author;
    }
    public void setAuthor(String value) {
        this.author = value;
    }
    public String getTitle() {
        return title;
    }

     public void setTitle(String value) {
        this.title = value;
    }


    public String getGenre() {
        return genre;
    }

    public void setGenre(String value) {
```

```
        this.genre = value;
    }


    public float getPrice() {
        return price;
    }


    public void setPrice(float value) {
        this.price = value;
    }


    public XMLGregorianCalendar getPublishDate() {
        return publishDate;
    }


    public void setPublishDate(XMLGregorianCalendar value) {
        this.publishDate = value;
    }


    public String getDescription() {
        return description;
    }


    public void setDescription(String value) {
        this.description = value;
    }


    public String getId() {
        return id;
    }


    public void setId(String value) {
        this.id = value;
    }

}
```

2. Open a command prompt.
3. Run the **schemagen** schema generator tool from the directory where you copied the `Bookdata.java` file.
4. The XML schema file, `schema1.xsd` is generated:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="bookdata">
    <xs:sequence>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="genre" type="xs:string"/>
      <xs:element name="price" type="xs:float"/>
      <xs:element name="publish_date" type="xs:anySimpleType"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Refer to the JAXB Reference implementation documentation for additional information about the **schemagen** command.

Using JAXB tools to generate JAXB classes from an XML schema file

Use Java Architecture for XML Binding (JAXB) tooling to compile an XML schema file into fully annotated Java classes.

Before you begin

Develop or obtain an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. XML schemas describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop Web services using a top-down development approach starting with an existing Web Services Description Language (WSDL) file, use the wsimport tool to generate the artifacts for your Java API for XML-Based Web Services (JAX-WS) applications when starting with a WSDL file. After the Java artifacts for your application are generated, you can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, xjc command-line tool. The resulting annotated Java classes contain all the necessary information that the JAXB runtime requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web Services (JAX-WS) applications or other Java applications for processing XML data.

In addition to using the xjc tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the xjc schema compiler tool.

1. Use the JAXB schema compiler, xjc command to generate JAXB-annotated Java classes. The schema compiler is located in the *app_server_root*\bin\ directory. The schema compiler produces a set of packages containing Java source files and JAXB property files depending on the binding options used for compilation.
2. (Optional) Use custom binding declarations to change the default JAXB mappings. Define binding declarations either in the XML schema file or in a separate bindings file. You can pass custom binding files by using the **-b** option with the xjc command.
3. Compile the generated JAXB objects. To compile generated artifacts, add the Thin Client for JAX-WS with WebSphere Application Server to the classpath.

Results

Now that you have generated JAXB objects, you can write Java applications using the generated JAXB objects and manipulate the XML content through the generated JAXB classes.

Example

The following example illustrates how JAXB tooling can generate Java classes when starting with an existing XML schema file.

1. Copy the following `bookSchema.xsd` schema file to a temporary directory.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="CatalogData">
     <xsd:complexType >
       <xsd:sequence>
         <xsd:element name="books" type="bookdata" minOccurs="0"
```

```
 maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="bookdata">
    <xsd:sequence>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="genre" type="xsd:string"/>
      <xsd:element name="price" type="xsd:float"/>
      <xsd:element name="publish_date" type="xsd:dateTime"/>
      <xsd:element name="description" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

2. Open a command prompt.
3. Run the JAXB schema compiler, xjc command from the directory where the schema file is located. The xjc schema compiler tool is located in the *app_server_root*\bin\ directory.

   Running the xjc command generates the following JAXB Java files:

   ```
   generated\Bookdata.java
   generated\CatalogdData.java
   generated\ObjectFactory.java
   ```
4. Use the generated JAXB objects within a Java application to manipulate XML content through the generated JAXB classes.

Refer to the JAXB 2.0 Reference implementation documentation for additional information about the xjc command.

Using the JAXB runtime to marshal and unmarshal XML documents

Use the Java Architecture for XML Binding (JAXB) run time to manipulate XML instance documents.

Before you begin

Use JAXB to generate Java classes from an XML schema with the schema compiler, xjc command or to generate an XML schema from a Java class with the schema generator, schemagen command.

About this task

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. After data bindings exist, use the JAXB binding runtime API to convert XML instance documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the data structure. JAXB annotated classes and artifacts contains all the information that the JAXB runtime API needs to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML and unmarshaling the XML document back to JAXB class instances.

- Marshal JAXB objects to XML instance documents.

  Use the JAXB runtime API to marshal or convert JAXB object instances into an XML instance document.

  1. Instantiate your JAXB classes.
  2. Invoke the JAXB marshaller.

  This example demonstrates how to instantiate the generated JAXB objects within an application and use the JAXBContext class and the JAXB runtime marshaller APIs to marshal the JAXB objects into XML instances.

```
JAXBContext jc = JAXBContext.newInstance("myPackageName");
//Create marshaller
Marshaller m = jc.createMarshaller();
//Marshal object into file.
m.marshal(myJAXBObject, myOutputStream);
```

The JAXB Reference Implementation introduces additional vendor specific marshaller properties such as namespace prefix mapping, indentation, and character escaping control that are not defined by the JAXB specification. Use these properties to specify additional controls of the marshaling process. These properties operate with the JAXB Reference Implementation only and might not with other JAXB providers. Additional information regarding the vendor specific properties is located in the Java Architecture for XML Binding JAXB RI Vendor Extensions Runtime Properties specification.

- Unmarshal XML files to JAXB objects.

  Use the JAXB runtime API to unmarshal or convert an XML instance document to JAXB object instances.

  1. Obtain an existing XML instance document.
  2. Invoke the JAXB unmarshaller.

  This example demonstrates a program that reads an XML document and unmarshals or converts the XML document into JAXB object instances. Use the JAXBContext class and JAXB runtime Unmarshaller APIs to unmarshal the XML document.

```
JAXBContext jc = JAXBContext.newInstance("myPackageName");
//Create unmarshaller
Unmarshaller um = jc.createUnmarshaller();
//Unmarshal XML contents of the file myDoc.xml into your Java
  object instance.
MyJAXBObject myJAXBObject = (MyJAXBObject)
um.unmarshal(new java.io.FileInputStream( "myDoc.xml" ));
```

Results

You can now marshal JAXB Java classes, and unmarshal XML data using the JAXB binding framework. Refer to the JAXB 2.0 Reference implementation documentation for additional information about the marshal and unmarshal runtime APIs

Note: If Java 2 Security is enabled, wrap your JAXBContext.newInstance(), Unmarshaller.unmarshal() and, Marshaller.marshal() method calls within a AccessController.doPrivileged method to avoid a security exception.

.

xjc command for JAXB applications

Use the Java Architecture for XML Binding (JAXB) tools to generate Java classes from an XML schema with the xjc schema compiler tool.

JAXB is an XML-to-Java binding technology that enables transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB technology consists of a runtime API and accompanying tools that simplify access to XML documents. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides a runtime environment to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the XML data structure.

You can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, xjc command-line tool. Use the xjc schema compiler tool to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. Once the mapping between XML schema and Java classes exists, XML instance documents can be converted

to and from Java objects through the use of the JAXB binding runtime API. The resulting annotated Java classes contains all the necessary information that the JAXB runtime requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web Services (JAX-WS) applications or in your non-JAX-WS Java applications for processing XML data.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Syntax

The command line syntax is:

If a directory is specified, all schema files in the directory are compiled.

Parameters

The schema file/URL JAR file name or location of the directory is the only parameter that is required. The following parameters are optional for the xjc command:

-b <file_name or directory>
Specifies the external JAX-WS or JAXB binding files. You can specify multiple JAX-WS and JAXB binding files by using the `-b` option; however, each file must be specified with its own `-b` option. If a directory is specified, **/*.xjb is searched.

-catalog <file_name>
Specifies the catalog file to resolve external entity references. It supports the TR9401, XCatalog, and the OASIS XML Catalog formats.

-classpath <path>
Specifies the location of the class files.

-d <directory>
Specifies where to place the generated output files.

-dtd
Specifies to treat the input as XML Document Type Definition (DTD). This option is unsupported and experimental.

-extension
Specifies whether to enable custom extensions for functionality not specified by the JAX-B specification. Use of the extensions can result in applications that are not portable or do not interoperate with other implementations.

-help
Displays the help menu.

-httpproxy <[user[:password]@]<proxyhost>:<proxyport>>
Specifies an HTTP or HTTPs proxy.

-httpproxyfile <file_name>
Similar to the -httpproxy parameter, but takes the argument in a file to protect the password.

-no-header
Specifies to suppress the generation of a file header with a timestamp.

-npa
Specifies to suppress the generation of the **/package-info.java package level annotation.

-nv
Specifies to not perform a strict validation of the input schemas.

-p <package_name>
Specifies a target package.

-quiet
Specifies to suppress the output from the xjc tool.

-relaxng
Specifies to treat the input as REgular LAnguage for XML Next Generation (RELAX NG). This option is unsupported and experimental.

-readOnly
Specifies that the generated files are in read-only mode.

-relaxng-compact
Specifies to treat the input as REgular LAnguage for XML Next Generation (RELAX NG) compact syntax. This option is unsupported and experimental.

-target <version>
Specifies to generate output to conform to the specified level of the JAX-WS specification. Specify *2.0* for the tool to generate compliant code for the JAX-B 2.0 specification. The default target version is *2.1* and generates compliant code for JAX-B 2.1.

-verbose
Specifies to output messages about what the compiler is doing.

-version
Prints the version information. If you specify this option, only the version information is output and typical command processing does not occur.

-wsdl
Specifies to treat the input as a Web Services Description Language (WSDL) file and compile schemas inside the WSDL. This option is unsupported and experimental.

-xmlschema
Specifies to treat the input as a World Wide Web Consortium (W3C) XML schema. This value is the default.

schemagen command for JAXB applications

Use the schema generator tool, schemagen, to generate an XML schema using Java Architecture for XML Binding (JAXB).

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. XML schemas describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

You can generate a schema file from Java classes using the schemagen schema generator tool to create the XML schema. After the mapping between XML schema and Java classes exists, XML instance documents can be converted to and from Java objects through the use of the JAXB binding runtime API. The resulting Java classes contain all the necessary information that the JAXB run time requires to parse

the XML for marshaling and unmarshaling. You can use the JAXB classes within Java API for XML Web Services (JAX-WS) applications or in your non-JAX-WS Java applications for processing XML data.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Syntax

The command line syntax is:

Parameters

The following parameters are optional for the schemagen command:

-classpath <path>
Specifies the location of the Java source or class files.

-cp <path>
Specifies the location of the Java source or class files.

-d <path>
Specifies where to place the processor and the generated Java class files.

-episode<file_name>
Specifies to generate an episode file for separate compilation.

-help
Displays the help menu.

-version
Prints the version information. If you specify this option, only the version information is output and typical command processing does not occur.

Using handlers in JAX-WS Web services

Java API for XML Web Services (JAX-WS) provides you with a standard way of developing interoperable and portable Web services. Use JAX-WS handlers to customize Web services requests or response handling.

Before you begin

You need an enterprise archive (EAR) file for the applications that you want to configure. If you are running in a Thin Client for JAX-WS environment, then you are not required to begin with an EAR file. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file must contain the handler classes to configure. The handler classes implement the `javax.xml.ws.handler.LogicalHandler<LogicalMessageContext>` interface or the

javax.xml.ws.handler.soap.SOAPHandler<SOAPMessageContext> interface.

About this task

As in the Java API for XML-based RPC (JAX-RPC) programming model, the JAX-WS programming model provides an application handler facility that enables you to manipulate a message on either an inbound or an outbound flow. You can add handlers into the JAX-WS runtime environment to perform additional processing of request and response messages. You can use handlers for a variety of purposes such as capturing and logging information and adding security or other information to a message. Because of the support for additional protocols beyond SOAP, JAX-WS provides two different classifications for handlers. One type of handler is a logical handler that is protocol independent and can obtain the message in the flow as an extensible markup language (XML) message. The logical handlers operate on message context properties and message payload. These handlers must implement the javax.xml.ws.handler.LogicalHandler interface. A logical handler receives a LogicalMessageContext object from which the handler can get the message information. Logical handlers can exist on both SOAP and XML/HTTP-based configurations.

The second type of handler is a protocol handler. The protocol handlers operate on message context properties and protocol-specific messages. Protocol handlers are limited to SOAP-based configurations and must implement the javax.xml.ws.handler.soap.SOAPHandler interface. Protocol handlers receive the message as a javax.xml.soap.SOAPMessage to read the message data.

The JAX-WS runtime makes no distinction between server-side and client-side handler classes. The runtime does not distinguish between inbound or outbound flow when a handleMessage(MessageContext) method or handleFault(MessageContext) method for a specific handler is invoked. You must configure the handlers for the server or client, and implement sufficient logic within these methods to detect the inbound or outbound direction of the current message.

To use handlers with Web services client applications, you must add the @HandlerChain annotation to the service endpoint interface or the generated service class and provide the handler chain configuration file. The @HandlerChain annotation contains a file attribute that points to a handler chain configuration file that you create. For Web services client applications, you can also configure the handler chain programmatically using the Binding API. To modify the handlerchain class programmatically, use either the default implementation or a custom implementation of the HandlerResolver method.

To use handlers with your server application, you must set the @HandlerChain annotation on either the service endpoint interface or the endpoint implementation class, and provide the associated handler chain configuration file. Handlers for the server are only configured by setting the @HandlerChain annotation on the service endpoint implementation or the implementation class. The handler classes must be included in the server application EAR file.

For both server and client implementations of handlers using the @HandlerChain annotation, you must specify the location of the handler configuration as either a relative path from the annotated file or as an absolute URL. For example:

```
@HandlerChain(file="../../common/handlers/myhandlers.xml")
```

or

```
@HandlerChain(file="http://foo.com/myhandlers.xml")
```

For more information on the schema of the handler configuration file, see the JSR 181 specification.

For more information regarding JAX-WS handlers, see chapter 9 of the JAX-WS specification.

1. Determine if you want to implement JAX-WS handlers on the service or the client.
2. Configure the client handlers by setting the @HandlerChain annotation on the service instance or service endpoint interface, or you can modify the handler chain programmatically to control how the handler chain is built in the runtime. If you choose to modify the handler chain programmatically, then

you must determine if you will use the default handler resolver or use a custom implementation of a handler resolver that is registered on the service instance. A service instance uses a handler resolver when creating binding providers. When the binding providers are created, the handler resolver that is registered with a service is used to create a handler chain and the handler chain is subsequently used to configure the binding provider.

a. Use the default implementation of a handler resolver. The runtime now uses the @HandlerChain annotation and the default implementation of HandlerResolver class to build the handler chain. You can obtain the existing handler chain from the Binding, add or remove handlers, and then return the modified handler chain to the Binding object.
b. To use a custom implementation of a handler resolver, set the custom HandlerResolver class on the Service instance. The runtime uses your custom implementation of the HandlerResolver class to build the handler chain, and the default runtime implementation is not used. In this scenario, the @HandlerChain annotation is not read when retrieving the handler chain from the binding after the custom HandlerResolver instance is registered on the Service instance. You can obtain the existing handler chain from the Binding, add or remove handlers, and then return the modified handler chain to the Binding object.

3. Configure the server handlers by setting the @HandlerChain annotation on the service endpoint interface or implementation class. When the @HandlerChain annotation is configured on both the service endpoint interface and the implementation class, the implementation class takes priority.
4. Create the handler chain configuration XML file. You must create a handler chain configuration XML file for the @HandlerChain to reference.
5. Add the handler chain configuration XML file in the class path for the service endpoint interface when configuring the server or client handlers using the @HandlerChain annotation. You must also include the handler classes contained in the configuration XML file in your class path.
6. Write your handler implementation.

Results

You have the enabled your JAX-WS Web service or Web services client to use handlers to perform additional processing of request and response message exchange.

Example

The following example illustrates the steps necessary to configure JAX-WS handlers on a service endpoint interface using the @HandlerChain annotation.

The @HandlerChain annotation has a file attribute that points to a handler chain configuration XML file that you create. The following file illustrates a typical handler configuration file. The protocol-bindings, port-name-pattern, and service-name-pattern elements are all filters that are used to restrict which services can apply the handlers.

```
<?xml version="1.0" encoding="UTF-8"?>

<jws:handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
<!-- Note:  The '*" denotes a wildcard. -->

 <jws:handler-chain name="MyHandlerChain">
  <jws:protocol-bindings>##SOAP11_HTTP ##ANOTHER_BINDING</jws:protocol-bindings>
  <jws:port-name-pattern
              xmlns:ns1="http://handlersample.samples.ibm.com/">ns1:MySampl*</jws:port-name-pattern>
        <jws:service-name-pattern
              xmlns:ns1="http://handlersample.samples.ibm.com/">ns1:*</jws:service-name-pattern>
  <jws:handler>
   <jws:handler-class>com.ibm.samples.handlersample.SampleLogicalHandler</jws:handler-class>
  </jws:handler>
  <jws:handler>
   <jws:handler-class>com.ibm.samples.handlersample.SampleProtocolHandler2</jws:handler-class>
  </jws:handler>
```

```
  <jws:handler>
   <jws:handler-class>com.ibm.samples.handlersample.SampleLogicalHandler</jws:handler-class>
  </jws:handler>
  <jws:handler>
   <jws:handler-class>com.ibm.samples.handlersample.SampleProtocolHandler2</jws:handler-class>
  </jws:handler>
 </jws:handler-chain>

</jws:handler-chains>

</jws:handler-chains>
```

Make sure that you add the handler.xml file and the handler classes contained in the handler.xml file in your class path.

The following example demonstrates a handler implementation:

```
package com.ibm.samples.handlersample;

import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class SampleProtocolHandler implements
        javax.xml.ws.handler.soap.SOAPHandler<SOAPMessageContext> {

    public void close(MessageContext messagecontext) {
    }

    public Set<QName> getHeaders() {
        return null;
    }

    public boolean handleFault(SOAPMessageContext messagecontext) {
        return true;
    }

    public boolean handleMessage(SOAPMessageContext messagecontext) {
        Boolean outbound = (Boolean) messagecontext.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        if (outbound) {
            // Include your steps for the outbound flow.
        }
        return true;
    }

}
```

What to do next

Deploy your Web services application.

Running an unmanaged Web services JAX-WS client

WebSphere Application Server provides an unmanaged client implementation that is based on the Java API for XML-based Web Services (JAX-WS) 2.1 specification. The Thin Client for JAX-WS with WebSphere Application Server is an unmanaged and stand-alone Java client environment that enables running JAX-WS Web services client applications to invoke Web services that are hosted by WebSphere Application Server.

Before you begin

Before you set up a JAX-WS unmanaged client execution environment, obtain the Thin Client for JAX-WS Java archive (JAR) file. To obtain the Thin Client for JAX-WS, install WebSphere Application Server Version 7.0 or the Application Client for WebSphere Application Server Version 7.0. The Thin Client for JAX-WS JAR file, `com.ibm.jaxws.thinclient_7.0.0.jar`, is located in the *app_server_root*`\runtimes` directory.

Copy the Thin Client for JAX-WS, `com.ibm.jaxws.thinclient_7.0.0.jar` file, to other machines to create a lightweight client environment that enables communications with the product. Copies of the Thin Client for JAX-WS are subject to the same terms and conditions of the license agreement for the WebSphere product where you obtained the Thin Client for JAX-WS. Refer to the license agreements for correct usage and other limitations.

The Thin Client for JAX-WS works with IBM Software Development Kits (SDKs) Version 6.0. The Thin Client for JAX-WS is also supported on non-IBM SDKs V6.0 with this limitation:

- Xerces limitation on non-IBM SDKs

  You must download Xerces-J Version 2.6.2, and add the file to the classpath when setting up the Thin Client for JAX-WS environment.

About this task

Set up a Thin Client for JAX-WS environment by completing the following steps.

1. Configure the path. Enter the following command to add the Java `bin` directories to your path:

   ```
   set PATH=<your_JDK_bin_directory>;%PATH%
   ```

2. Configure the classpath.
   - Add the Thin Client for JAX-WS JAR file to the classpath definition.

     ```
     set CLASSPATH=.;<your_jax-ws_thin_client_install_directory>\com.ibm.jaxws.thinclient_7.0.0.jar;
     <your_application_jars>;%CLASSPATH%
     ```

   - If you are using a non-IBM SDK, obtain a Xerces `xml-apis.jar` file and `xercesImpl.jar` file from the Xerces Web site, and configure the classpath definition.

     ```
     set CLASSPATH=.;<your_Xerces_install_directory>\xml-apis.jar;<your_Xerces_install_directory>
     \xercesImpl.jar;%CLASSPATH%
     ```

3. Configure SSL for the client.
   a. Add the following system properties to the Java command:

      ```
      -Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props
      ```

      You can obtain the `ssl.client.props` file from the WebSphere Application Server installation and modify the file to suit your environment. You must, at a minimum, update the location of the com.ibm.ssl.keyStore and com.ibm.ssl.trustStore key files in the `ssl.client.props` file to the match location of your target environment. For example, use these SSL configuration settings when running the application with a Sun JRE:

      ```
      com.ibm.ssl.protocol=SSL
      com.ibm.ssl.trustManager=SunX509
      com.ibm.ssl.keyManager=SunX509
      com.ibm.ssl.contextProvider=SunJSSE

      com.ibm.ssl.keyStoreType=JKS
      com.ibm.ssl.keyStoreProvider=SUN
      com.ibm.ssl.keyStore=/home/user1/etc/key.jks

      com.ibm.ssl.trustStoreType=JKS
      com.ibm.ssl.trustStoreProvider=SUN
      com.ibm.ssl.trustStore=/home/user1/etc/trust.jks
      ```

      The key store file and trust store file must be created using the Java keytool utility before the application runs. The automatic key file generation is not supported with a non-IBM product JRE.

You must override the default ORB implementation of the non-IBM product JRE with the `com.ibm.ws.orb_7.0.0.jar` file, or add it to the classpath.

4. Enter the following command to run your client application:

```
%JAVA_HOME%/bin/java -Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props <your_client_application>
```

Results

You have set up an unmanaged JAX-WS client runtime environment to invoke Web services hosted on a WebSphere Application Server.

Related tasks

"Task overview: Implementing Web services applications" on page 415
Use this topic as an introduction to using Web services. WebSphere Application Server supports Web services that are developed and implemented based on a variety of Java programming models. Use Web services when operating across a variety of platforms, including Java Platform, Enterprise Edition (Java EE) and non-Java EE platforms.

"Developing and deploying JAX-WS Web services clients" on page 530
You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-Based Web Services (JAX-WS) programming model.

Example: Installing a Web Services Sample with the console
The product provides a Web Services sample application that you can install on a Version 7.x application server.

Related information

Xerces Web site

Developing Web services applications with JAX-RPC

You can use the Java API for XML-based RPC (JAX-RPC) programming model to develop Web services.

Before you begin

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

About this task

To develop Web services based on the JAX-RPC programming model, you can use a bottom-up development approach starting from existing JavaBeans or enterprise beans or you can use a top-down development approach starting with an existing Web Services Description Language (WSDL) file. This task describes the steps when using the bottom-up development approach.

When developing a JAX-RPC Web service starting from existing JavaBeans or enterprise beans, you need develop a WSDL file. You can use existing JavaBeans or enterprise beans and then enable the implementation for Web services. Enabling the bean for Web services includes developing the service endpoint interface, developing a WSDL file that is the description of the Web service, generating and configuring the deployment descriptors, assembling all artifacts required for the Web service, and deploying the application onto the application server.

Considerations when using JavaBeans

JavaBeans exposed as JAX-RPC Web services are supported only over an HTTP transport.

Considerations when using enterprise beans

- The enterprise bean must be a stateless session bean.
- Enterprise beans that are exposed as JAX-RPC Web services must be packaged in EJB 2.1 or in EJB 3.0 or higher modules.
- For JAX-RPC Web services using EJB 2.1 style endpoints, the Web service method parameters must be one of the supported JAX-RPC types. These requirements are documented in the JAX-RPC specification.
- JAX-RPC Web services using enterprise beans are supported over an HTTP or Java Message Service (JMS) transport.

Note: It is a best practice to use EJB 2.1 style enterprise beans with JAX-RPC applications.

1. Set up a development environment for Web services. You do not have to set up a development environment if you are using Rational Application Developer.
2. Determine the existing JavaBeans or enterprise beans that you want to expose as a JAX-RPC Web service.
3. Develop a service endpoint interface. The service endpoint interface defines the JavaBeans or enterprise beans methods for a particular Web service. The JavaBeans must implement methods that have the same signature as the methods on the service endpoint interface.
   - Develop a service endpoint interface for JavaBeans applications.
   - Develop a service endpoint interface for enterprise beans applications.
4. Develop the Java artifacts.
   a. Develop a WSDL file. The WSDL file is the description of a Java Platform, Enterprise Edition (Java EE) Web service. For JAX-RPC applications, a WSDL file is required.
   b. Develop JAX-RPC deployment descriptors. Use the WSDL2Java command-line tool to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans or enterprise beans implementation.
      - Develop Web services deployment descriptor templates for a JavaBeans implementation.
      - Develop Web services deployment descriptor templates for an enterprise beans implementation.
5. Complete the implementation of your Web service application.
   - For JavaBeans applications, complete the JavaBeans implementation.
   - For enterprise beans applications, complete the enterprise beans implementation.
6. Configure the `webservices.xml` deployment descriptor. For JAX-RPC Web services, configure the webservices.xml deployment descriptor so that the application server can process the incoming Web services requests.
7. Configure the `ibm-webservices-bnd.xmi` deployment descriptor. Configure the `ibm-webservices-bnd.xml` deployment descriptor so that the application server can process the incoming Web services requests.
8. Assemble the artifacts for your Web service.

   Use assembly tools provided with the application server to assemble your Java-based Web services modules.

   If you have assembled an EAR file that contains enterprise beans modules that contain Web services, use the endptEnabler command-line tool or an assembly tool before deployment to produce a Web services endpoint WAR file. This tool is also used to specify whether the Web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.
9. Deploy the EAR file into the application server. You can now deploy the EAR file that has been configured and enabled for JAX-RPC Web services onto the application server.

Results

You have developed a JAX-RPC Web service application.

What to do next

After you deploy the EAR file, test the Web service to make sure that the service works with the application server.

Developing a service endpoint interface from JavaBeans for JAX-RPC applications

You must develop a service endpoint interface if you are developing a JAX-RPC Web service from a JavaBeans implementation.

Before you begin

You need to set up a development environment for Web services and access an existing Java bean Web archive (WAR) file.

About this task

This task is a required step in developing a JAX-RPC Web service from a Java bean.

The service endpoint interface defines the methods for particular Java API for XML-based RPC (JAX-RPC) Web services. The JavaBeans implementation must implement methods with the same signature as the methods on the service endpoint interface. A number of restrictions apply on which types to use as parameters and results of service endpoint interface methods. These restrictions are documented in the JAX-RPC specification.

You can also create a service endpoint interface by using assembly tools.

Develop a service endpoint interface for a JavaBeans implementation by following the actions listed:

1. Create a Java interface that contains the methods to include in the service endpoint interface. If you start with an existing Java interface, remove any methods that do not conform to the JAX-RPC specification.
2. Compile the interface.

   Use the name of the service endpoint interface class in the **javac** command for the class to compile.

Results

You have developed a service endpoint interface that you can use to develop Web services.

Example

The following example depicts the AddressBook interface:

```
package addr;
public interface AddressBook {
    /**
     * Retrieve an entry from the AddressBook.
     *
     *@param name the name of the entry to look up.
     *@return the AddressBook entry matching name or null if none.
     *@throws java.rmi.RemoteException if communications failure.
     */
    public addr.Address getAddressFromName(java.lang.String name);
}
```

Use the AddressBook interface to create the service endpoint interface:

1. Make a copy of the AddressBook.java interface and name it `AddressBook_SEI.java`. Use this copy as a template for the service endpoint interface.

2. Compile the interface.

What to do next

Continue to gather the artifacts that are required to develop a Web service, including the Web Services Description Language (WSDL) file. You need to develop a WSDL file because it is the engine of a Web service. Without a WSDL file, you do not have a Web service.

Developing a service endpoint interface from enterprise beans for JAX-RPC applications

You can develop a service endpoint interface from an Enterprise JavaBeans (EJB) for a JAX-RPC Web service.

Before you begin

Set up a development environment for Web services.

This task is a required step in developing a Java API for XML-based RPC (JAX-RPC) Web service from an enterprise bean.

The service endpoint interface defines the Web services methods. The enterprise beans that implements the Web service must implement methods having the same signature as the methods of the service endpoint interface. A number of restrictions exist on which types to use as parameters and results of service endpoint interface methods. These restrictions are documented in the Java API for XML-based remote procedure call (JAX-RPC) specification. See the Web services specifications and API documentation to review the JAX-RPC specification along with a complete list of the supported standards and specifications.

The easiest method for creating the service endpoint interface for an EJB Web service implementation is from the EJB remote interface.

You can also create a service endpoint interface by using assembly tools..

About this task

Develop a service endpoint interface by following the steps provided in this task section.

1. Create a Java interface that contains the methods that you want to include in the service endpoint interface. If you start with an existing Java interface, remove any methods that do not conform to the JAX-RPC specification.
2. Compile the interface.

   Use the name of the service endpoint interface class in the **javac** command for the class to compile.

Results

You have a service endpoint interface that you can use to develop a Web service.

Example

This example uses the EJB remote interface, AddressBook_RI, to create a service endpoint interface for an EJB implementation that is used as a Web service. The following code example illustrates the AddressBook_RI remote interface.

```
package addr;
public interface AddressBook_RI extends javax.ejb.EJBObject {
    /**
     * Retrieve an entry from the AddressBook.
```

```
     *
     *@param name the name of the entry to look up.
     *@return the AddressBook entry matching name or null if none.
     *@throws java.rmi.RemoteException if communications failure.
     */
    public addr.Address getAddressFromName(java.lang.String name)
     throws java.rmi.RemoteException;
}
```

Use the following steps to create the service endpoint interface with the AddressBook_RI remote interface:
1. Locate a remote interface that has already been created, like the `AddressBook_RI.java` remote interface.
2. Make a copy of the `AddressBook.java` remote interface and use it as a template for the service endpoint interface.
3. Compile the `AddressBook.java` service endpoint interface.

What to do next

Continue gathering the artifacts that are required to develop a Web service, including the Web Services Description Language (WSDL) file. You need to develop a WSDL file because it is the engine of a Web service; without a WSDL file, you have no Web service.

Developing a WSDL file for JAX-RPC applications

You can develop a Web Services Description Language (WSDL) file to describe the characteristics of your Java API for XML-based RPC (JAX-RPC) Web services application including where the service resides and how to invoke the service using an XML format.

Before you begin

Depending on your development path, develop a service endpoint interface from a JavaBeans implementation or develop a service endpoint interface from an enterprise bean implementation.

About this task

You need a WSDL file to use Web services. You can develop your own WSDL file or get one from a Web services provider through e-mail, downloading, or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own.

Develop a WSDL file by following the actions listed:
1. Configure the service endpoint interface class and referenced classes into your CLASSPATH variable.
   - On Windows systems, set `CLASSPATH="%CLASSPATH%;<`*list your application Java archive (JAR) files and classes*`>"`.
   - On UNIX and Linux systems, export `CLASSPATH="$CLASSPATH:<`*list your application JAR files and classes*`>"`.
2. Run the Java2WSDL *seiInterface* command. A WSDL file named *`seiInterface`*`.wsdl` is created.

   Note: The Java2WSDL command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the Java2WSDL command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.
   - Move the WSDL file to the `META-INF/wsdl` subdirectory if you are using Enterprise JavaBeans (EJB).
   - Move the WSDL file to the `WEB-INF/wsdl` subdirectory if you are using JavaBeans.
3. Edit the generated WSDL file and inspect the part names. The WSDL parts have names like `arg_0_0`. Modify the WSDL file to use the actual names of the Java parameters.

4. (Optional) Use the **Java2WSDL** command tool to generate the correct part names of WSDL file. You can automatically generate and set the correct part names by using the **Java2WSDL** command tool. Generating and setting the part names is done by providing additional information to the **Java2WSDL** command tool in the form of a Java implementation class that implements the same methods as the service endpoint interface and is compiled with debug information turned on. Parameter names are stored in the `.class` file with the debug information. If your implementation class is compiled with debug on, you can use the **Java2WSDL -implClass *seiImpl seiInterface*** command to generate a WSDL file with the proper part names.

Results

A WSDL file that defines the Web services described by the service endpoint interface.

Example

This example uses the JAR file name `AddressBook.jar` that contains a class named `AddressBook.class` class file.

You must add the `AddressBook.jar` file to your CLASSPATH to create the WSDL file. The JAR file contains an EJB implementation class that is compiled with debugging information turned on. Run the **Java2WSDL -implClass addr.AddressBookBean addr.AddressBook** command to create the file, `AddressBook.wsdl`.

What to do next

Depending on your development path, develop Web services deployment descriptor templates for JavaBeans or develop Web services deployment descriptor templates for an enterprise beans implementation.

Java2WSDL command for JAX-RPC applications

The Java2WSDL command-line tool maps Java classes to a WSDL file for Java API for XML-based RPC (JAX-RPC) applications.

The Java2WSDL command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product.

The Java2WSDL command maps a Java class to a Web Services Description Language (WSDL) file by following the Java API for XML-based Remote Procedure Call (JAX-RPC) 1.1 specification.

The Java2WSDL command accepts a Java class as input and produces a WSDL file that represents the input class. If a file exists at the output location, it is overwritten. The WSDL file that is generated by the Java2WSDL command contains WSDL and XML schema constructs that are automatically derived from the input class. You can override these default values with command-line arguments.

The Java2WSDL command is protocol independent; when you run the Java2WSDL command, you can specify command-line options that generate both SOAP and non-SOAP protocol bindings in the WSDL file. For each binding that can be generated, the **Java2WSDL** command has a binding generator to generate the WSDL for that binding.

Command line syntax and arguments

The command line syntax is:

`Java2WSDL [`*`argument...`*`] `*`class`*

The following command-line arguments are supported:

Required arguments

- class

  Represents the fully qualified name of one of the following Java classes:
  - Stateless session Enterprise JavaBeans (EJB) remote interface that extends the javax.ejb.EJBObject class
  - Service endpoint interface that extends the java.rmi.Remote class
  - Java beans

  The **Java2WSDL** command locates the class in the CLASSPATH variable.

Important arguments

- -location *location*

  Provides the published location or the Uniform Resource Locator (URL) of the service. If this information is not provided, a warning is issued that indicates that the final published location is not determined yet. The service location is typically overridden when the Web service is deployed.

  The name after the last backslash is the name of the service port, unless the name is overridden by the -servicePortName argument. The service port address location attribute is assigned the specified value. Multiple endpoint addresses can be specified. Using the -location option is recommended only if a single binding type is required. If multiple binding types are requested, protocol binding-specific location properties are passed over the command line using the -x flag.

  The following example illustrates how to produce both SOAP over HTTP, and SOAP over Java Message Service (JMS) bindings :

  ```
  java2wsdl -bindingTypes http,jms \
    -x http.location=http://your.server.name:9080/StockQuoteService/services/StockQuote \
    -x jms.location= \
   jms:/queue?destination=jms/MyQueue&connectionFactory=jms/MyCF&targetService=StockQuote
  ```

  Use the -location option to determine to which port the -location option value applies by requiring the endpoint URLs to be specified through the binding-specific property values.
- -output *wsdl-uri*

  Indicates the path and file name of the output WSDL file. If not specified, the default *class*.wsdl file is written into the current directory.
- -input *wsdl-uri*

  Specifies the input WSDL file that is used to build an output WSDL file. Information from an existing WSDL file, is specified in this option and is used with the input Java class to generate the output.
- -bindingTypes

  Specifies the list of binding types write to the output WSDL file. Each binding generator in the Java2WSDL command supports specific binding types. The valid binding type values are `http` (SOAP over HTTP), `jms` (SOAP over JMS) and `ejb` (local or remote EJB invocation). For example, the following command can be used to generate SOAP over HTTP, EJB bindings for the my.pkg.MySEI Service Endpoint Interface and the my.pkg.MyEJBClass implementation class :

  ```
  java2wsdl -bindingTypes http,ejb -implClass my.pkg.MyEJBClass my.pkg.MySEI
  ```

  The following command is an example of using the -bindingTypes option to generate SOAP over HTTP and SOAP over JMS bindings:

  ```
  java2wsdl -bindingTypes http,jms -implClass my.pkg.MyEJBClass my.pkg.MySEI
  ```
- -style RPC | DOCUMENT

  Specifies the WSDL style to use in the generated WSDL file. For more information about styles, see Mapping between Java, WSDL and XML. This argument is used with the -use argument.

  If RPC is specified with -use ENCODED, a `style=rpc/use=encoded` WSDL file is generated. If RPC is specified with the -use LITERAL option, a `style=rpc/use=literal` WSDL file is generated. If DOCUMENT is specified with the -use LITERAL option, a `style=document/use=literal` WSDL file is generated.
- -use LITERAL | ENCODED

Specifies which style and use combinations are generated into the WSDL file when used with the -style argument. The combinations are rpc and encoded, rpc and literal, or doc and literal. This setting applies to all SOAP bindings. For more information, see the Mapping between Java language, WSDL and XML.

- -transport http | jms

  Generates SOAP bindings for either HTTP (default) or JMS. If JMS is specified, the characters `jms` are appended to the WSDL file name to prevent overwriting an existing WSDL file for another transport. The transport option can be specified only once.

  This option is deprecated. The -bindingTypes option replaces the -transport option, so that you can generate bindings that are non-SOAP specific.
- -portTypeName *name*

  Specifies the name to use for the portType element. If not specified, the binding name is the port type name.
- -bindingName *name*

  Specifies the name to use for the binding element. If not specified, the binding name is the port type name.
- -serviceElementName *name*

  Specifies the name of the service element.
- -servicePortName *name*

  Specifies the name of the service. If not specified, the service name is derived from the -location argument.
- -namespace *targetNamespace*

  Indicates the target namespace for the WSDL file being generated. See Mapping between Java code, WSDL and XML for the algorithm that is used to obtain the default namespace.
- -PkgtoNS *package namespace*

  Specifies the mapping of a Java package to a namespace. If a package does not have a namespace, the Java2WSDL command generates a namespace name. You can repeat the -PkgtoNS argument to specify mappings for multiple packages.
- -extraClasses *classes*

  Specifies other classes that are represented in the WSDL file.
- -implClass *impl-class*

  The Java2WSDL command uses method parameter names to construct the WSDL file message part names. The command automatically obtains the message names from the debug information in the class. If the class is compiled without debug information, or if the class is an interface, the method parameter names are not available. In this case, you can use the -implClass argument to provide an alternative class from which to obtain method parameter names. The impl-class does not need to implement the class if the class is an interface, but it must implement the same methods as the class.
- -verbose

  Displays verbose messages.
- -help

  Displays the help message.
- -helpX

  Displays the help message for extended options and for various options that are supported by binding generators.

Other arguments

- -wrapped *boolean*

  Specifies whether to generate the WSDL file according to wrapped rules. This option is valid if `use` is literal only. The option defaults to `true`.
- -stopClasses *parent* [, *parent*]

  The Java2WSDL command searches inherited classes and interfaces to construct the list of methods for WSDL file operations if the -all argument is specified.

The Java2WSDL command searches inherited classes and interfaces when generating extended complexTypes. The search stops when a class or an interface is found within a package that begins with `java` or `javax`. You can use the -stopClasses argument to define additional classes that cause the search to stop.

- -methods *argument*

  Specifies a list of method names from the Service Endpoint Interface that must be exposed in the output WSDL file. The list is separated by spaces or commas.

- -soapAction

  Valid arguments are:
  - DEFAULT

    Sets the soapAction field according to the deployment information.
  - NONE

    Sets the soapAction field to double quotes (`""`).
  - OPERATION

    Sets the soapAction field to the operation name.

- -outputImpl *impl-wsdl*

  Specifies if you want an interface and implementation WSDL file emitted.

- -locationImport *location-uri*

  Specifies the location of the interface WSDL file if you use the -outputImpl argument.

- -namespaceImpl *namespace*

  Specifies the target namespace for the implementation WSDL file, if you use the -outputImpl argument.

- -MIMEStyle <*style*>

  Specifies the Multipurpose Internet Mail Extensions (MIME)- type used to map to Web Services-Interoperability (WS-I) SOAP with attachments reference (wsi:swaRef) for the binding element.

  <*style*> can be one of the following:
  - WSDL11 (default): Exclusively map MIME types using WSDL 1.1 standards. If the MIME type cannot map to WSDL 1.1 standards, the command fails.
  - AXIS: Map MIME types using AXIS standards, for example image becomes axis:image.
  - swaRef: Map MIME types using WSDL 1.1 standards with two caveats:
    - DataHandler maps to the wsi:swaRef element instead of an application and octet-stream
    - If mapping is illegal through WSDL 1.1, map to the wsi:swaRef element

- -propertiesFile *argument*

  Sets existing options, such as -extraClasses, with a properties file instead of with a command line. The following example illustrates the use of this argument:

  ```
  extraClasses=com.ibm.Class1, com.sun.Class2,org.apache.Class3
  ```

- -voidReturn

  Valid arguments are:
  - ONEWAY

    Methods with void returns are one-way. This argument is the default for a JMS transport.
  - TWOWAY

    Methods with void returns are two-way. This argument is the default for an HTTP transport.

- -debug

  Displays debug messages.

- -property or -x

  You can use the -x option to pass command-line options to various binding generators. Use the -x option multiple times on the command line to specify a set of property values to pass to each binding generator method called by the **Java2WSDL** command. You can also use a single -x option to specify multiple properties by separating them with a comma, for example:

```
java2wsdl -x prop1=value1 -x prop2=value2
```

is equivalent to:

```
java2wsdl -x prop1=value1,prop2=value2
```

The -x option provides flexibility to specify each command-line option for each binding generator individually, if required. The value specified in the -x option overrides the value that is specified in the equivalent command-line option if both are specified.

Mapping between Java language, WSDL and XML for JAX-RPC applications

Data for Java API for XML-based Remote Procedure Call (JAX-RPC) applications flows as extensible Markup Language (XML). JAX-RPC applications use mappings to describe the data conversion between the Java language and extensible Markup Language (XML) technologies, including XML Schema, Web Services Description Language (WSDL) and SOAP that are supported by the application server.

For JAX-RPC applications, most of the mappings between the Java language and XML are specified by the JAX-RPC specification. Some mappings that are optional or unspecified in JAX-RPC are also supported. Review the JAX-RPC specification for a complete list of APIs. For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

Notational conventions

The following table specifies the namespace prefixes and corresponding namespace used.

Namespace prefix	Namespace
xsd	`http://www.w3.org/2001/XMLSchema`
xsi	`http://www.w3.org/2001/XMLSchema-instance`
soapenc	`http://schemas.xmlsoap.org/soap/encoding/`
wsdl	`http://schemas.xmlsoap.org/wsdl/`
wsdlsoap	`http://schemas.xmlsoap.org/wsdl/soap/`
ns	user-defined namespace
apache	`http://xml.apache.org/xml-soap`
wasws	`http://websphere.ibm.com/webservices/`

Detailed mapping information

The following sections identify the supported mappings, including:
- Java-to-WSDL mapping
- WSDL-to-Java mapping
- Mapping between WSDL and SOAP messages

Java-to-WSDL mapping

This section summarizes the Java-to-WSDL mapping rules. The Java-to-WSDL mapping rules are used by the **Java2WSDL** command for *bottom-up processing*. In bottom-up processing, an existing Java service implementation is used to create a WSDL file defining the Web service. The generated WSDL file can require additional manual editing for the following reasons:
- Not all Java classes and constructs have mappings to WSDL files. For example, Java classes that do not comply with the Java bean specification rules might not map to a WSDL construct.
- Some Java classes and constructs have multiple mappings to a WSDL file. For example, a java.lang.String class can map to either an xsd:string or soapenc:string construct. The **Java2WSDL** command chooses one of these mappings, but you must edit the WSDL file if a different mapping is required.

- Multiple ways exist to generate WSDL constructs. For example, you can generate the wsdl:part in wsdl:message with a type or element attribute. The **Java2WSDL** command makes an informed choice based on the -style and -use option settings.
- The WSDL file describes the instance data elements sent in the SOAP message. If you want to modify the names or format used in the message, the WSDL file must be edited. For example, the **Java2WSDL** command maps a Java bean property as an XML element. In some circumstances, you might want to change the WSDL file to map the Java bean property as an XML attribute.
- The WSDL file requires editing if header or attachment support is desired.
- The WSDL file requires editing if a multipart WSDL file, using the wsdl:import construct, is desired.

For simple services, the generated WSDL file is sufficient. For complicated services, the generated WSDL file is a good starting point. Read about the Java2WSDL command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

General issues

- **Package to namespace mapping**

  The JAX-RPC specification does not indicate the default mapping of Java package names to XML namespaces. The JAX-RPC specification does specify that each Java package must map to a single XML namespace. Likewise, each XML namespace must map to a single Java package. A default mapping algorithm is provided that constructs the namespace by reversing the names of the Java package and adding an `http://` prefix. For example, a package named, com.ibm.webservice, is mapped to the XML namespace `http://webservice.ibm.com`.

  You can override the default mapping between XML namespaces and Java package names by using the -NStoPkg and -PkgtoNS options of the **WSDL2Java** and **Java2WSDL** commands.
- **Identifier mapping**

  Java identifiers are mapped directly to WSDL and XML identifiers.

  Java bean property names are mapped to XML identifiers. For example, a Java bean, with getInfo and setInfo methods, maps to an XML construct with the name, `info`.

  The service endpoint interface method parameter names, if available, are mapped directly to the WSDL and XML identifiers. See the WSDL2Java command -implClass option for more details.
- **WSDL construction summary**

  The following table summarizes the mapping from a Java construct to the related WSDL and XML construct.

Java construct	WSDL and XML construct
Service endpoint interface	wsdl:portType
Method	wsdl:operation
Parameters	wsdl:input, wsdl:message, wsdl:part
Return	wsdl:output, wsdl:message, wsdl:part
Throws	wsdl:fault, wsdl:message, wsdl:part
Primitive types	xsd and soapenc simple types
Java beans	xsd:complexType
Java bean properties	Nested xsd:elements of xsd:complexType
Arrays	JAX-RPC defined xsd:complexType or xsd:element with a maxOccurs=″unbounded″ attribute
User defined exceptions	`xsd:complexType`

- **Binding** and **service** construction

  A wsdl:binding that conforms to the generated wsdl:portType is generated. A wsdl:service containing a port that references the generated wsdl:binding is generated. The names of the binding and service are controlled by the **Java2WSDL** command.

- **Style** and **use**

  Use the -style and -use options to generate different kinds of WSDL files. The four supported combinations are:
  - -style DOCUMENT -use LITERAL
  - -style RPC -use LITERAL
  - -style DOCUMENT -use LITERAL -wrapped false
  - -style RPC -use ENCODED

  The following is a brief description of each combination.
  - **DOCUMENT LITERAL**

    The **Java2WSDL** command generates a Web Services - Interoperability (WS-I) specification compliant document-literal WSDL file. The wsdl:binding is generated with embedded style=″document″ and use=″literal″ attributes. An xsd:element is generated for each service endpoint interface method to describe the request message. A similar xsd:element is generated for each service endpoint interface method to describe the response message.
  - **RPC LITERAL**

    The **Java2WSDL** command generates a WS-I compliant rpc-literal WSDL file. The wsdl:binding is generated with embedded style=″rpc″ and use=″literal″ attributes. The wsdl:message constructs are generated for the inputs and outputs of each service endpoint interface method. The parameters of the method are described by the part elements within the wsdl:message constructs.
  - **DOCUMENT LITERAL not wrapped**

    The **Java2WSDL** command generates a document-literal WSDL file according to the JAX-RPC specification. This WSDL file is not compliant with .NET. The main difference between DOCUMENT LITERAL and DOCUMENT LITERAL not wrapped is the use of wsdl:message constructs to define the request and response messages.
  - **RPC ENCODED**

    The **Java2WSDL** command generates an rpc-encoded WSDL file according to the JAX-RPC specification. This WSDL file is not compliant with the WS-I specification. The wsdl:binding is generated with embedded style=″rpc″ and use=″encoded″ attributes. Certain soapenc mappings are used to represent types and arrays.

Mapping of standard XML types from Java types

Many Java types map directly to standard XML types. For example, a java.lang.String maps to an xsd:string. These mappings are described in the JAX-RPC specification.

Generation of wsdl:types

Java types that cannot be mapped directly to standard XML types are generated in the `wsdl:types` section. A Java class that matches the Java bean pattern is mapped to an `xsd:complexType`. Review the JAX-RPC specification for a description of all the mapping rules. The following example illustrates the mapping for a sample base and derived Java classes.

Java:

```
public abstract class Base {
     public Base() {}
     public int a;                          // mapped
     private int b;                         // mapped via setter/getter
     private int c;                         // not mapped
     private int[] d;                       // mapped via indexed setter/getter

     public int getB() { return b;}         // map property b
     public void setB(int b) {this.b = b;}

     public int[] getD() { return d;}       // map indexed property d
     public void setD(int[] d) {this.d = d;}
     public int getD(int index) { return d[index];}
     public void setB(int index, int value) {this.d[index] = value;}

     public void someMethod() {...}         // not mapped
```

```
  }

  public class Derived extends Base {
     public int x;                          // mapped
     private int y;                         // not mapped
  }
```

Mapped to:

```
<xsd:complexType name="Base" abstract="true">
 <xsd:sequence>
  <xsd:element name="a" type="xsd:int"/>
  <xsd:element name="b" type="xsd:int"/>
  <xsd:element name="d" minOccurs="0" maxOccurs="unbounded" type="xsd:int"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
 <xsd:complexContent>
  <xsd:extension base="ns:Base">
   <xsd:sequence>
    <xsd:element name="x" type="xsd:int"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

- **Unsupported classes**

  If a class cannot be mapped to an XML type, the **Java2WSDL** command issues a message and an xsd:anyType reference is generated in the WSDL file. In these situations, modify the Web service implementation to use the JAX-RPC compliant classes.

WSDL-to-Java mapping

The **WSDL2Java** command generates Java classes using information described in the WSDL file.

General issues

- **Mapping of a namespace to a package**

  JAX-RPC does not specify the mapping of XML namespaces to Java package names. JAX-RPC does specify that each Java package map to a single XML namespace. Likewise, each XML namespace must map to a single Java package. A default mapping algorithm omits any protocol from the XML namespace and reverses the names. For example, an XML namespace `http://websphere.ibm.com` becomes a Java package with the name com.ibm.websphere.

  The default mapping of an XML namespace to a Java package disregards the context-root. If two namespaces are the same up to the first slash, they map to the same Java package. For example, the XML namespaces `http://websphere.ibm.com/foo` and `http://websphere.ibm.com/bar` map to the com.ibm.websphere Java package. You can override the default mapping between XML namespaces and Java package names by using the -NStoPkg and -PkgtoNS options of the **WSDL2Java** and **Java2WSDL** commands.

  Identifier mapping

  XML names are much richer than Java identifiers. They can include characters that are not permitted in Java identifiers. See Appendix 20 of the JAX-RPC specification for the rules to map an XML name to a Java identifier.

- **Java construction summary**

  The following table summarizes the Java-to-XML construction. See the JAX-RPC specification for a description of these mappings.

WSDL and XML construction	Java construction

xsd:complexType	Java bean class, Java exception class, or Java array
nested xsd:element/xsd:attribute	Java bean property
xsd:simpleType (enumeration)	JAX-RPC enumeration class
wsdl:message The method parameter signature typically is determined by the wsdl:message.	Service endpoint interface method signature
wsdl:portType	Service endpoint interface
wsdl:operation	Service endpoint interface method
wsdl:binding	Stub
wsdl:service	Service interface
wsdl:port	Port accessor method in Service interface

- **Mapping standard XML types**
  - **JAX-RPC simple XML types mapping**

    Many XML types are mapped directly to Java types. See the JAX-RPC specification for a description of these mappings.

    Mapping the XML types defined in the wsdl:types section

    The **WSDL2Java** command generates Java types for the XML schema constructs that are defined in the wsdl:types section. The XML schema language is broader than the required or optional subset defined in the JAX-RPC specification. The **WSDL2Java** command supports the required mappings and most of the optional mappings, as well as some XML schema mappings that are not included in the JAX-RPC specification. The **WSDL2Java** command ignores some constructs that it does not support. For example, the command does not support the default attribute. If an xsd:element is defined with the default attribute, the default attribute is ignored. In some cases, the command maps unsupported constructs to the Java interface, javax.xml.soap.SOAPElement.

    The standard Java bean mapping is defined in section 4.2.3 of the JAX-RPC specification. The xsd:complexType defines the type. The nested xsd:elements within the xsd:sequence or xsd:all groups are mapped to Java bean properties. For example:

    XML:

```
<xsd:complexType name="Sample">
  <xsd:sequence>
  <xsd:element name="a" type="xsd:string"/>
  <xsd:element name="b" maxOccurs="unbounded" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

    Java:

```
public class Sample {
     // ..
     public Sample() {}

     // Bean Property a
     public String getA()             {...}
     public void   setA(String value) {...}

     // Indexed Bean Property b
     public String[] getB()           {...}
     public String   getB(int index)  {...}
     public void     setB(String[] values) {...}
     public void     setB(int index, String value) {...}

  }
```

– **Mapping of the wsdl:portType construct**

  The wsdl:portType construct is mapped to the service endpoint interface. The name of the wsdl:portType construct is mapped to the name of the class of the service endpoint interface.

– **Mapping of the wsdl:operation construct**

  A wsdl:operation construct within a wsdl:portType is mapped to a method of the service endpoint interface. The name of the wsdl:operation is mapped to the name of the method. The wsdl:operation contains wsdl:input and wsdl:output elements that reference the request and response wsdl:message constructs using the message attribute. The wsdl:operation can contain a wsdl:fault element that references a wsdl:message describing the fault. These faults are mapped to Java classes that extend the exception, java.lang.Exception as discussed in section 4.3.6 of the JAX-RPC specification.

  - **Effect of document literal wrapped format**

    If the WSDL file uses the document literal wrapped format, the method parameters are mapped from the wrapper xsd:element. The document literal wrapped and literal format is automatically detected by the **WSDL2Java** command. The following criteria must be met:
    - The WSDL file must have style=″document″ in its wsdl:binding construct.
    - The input and output constructs of the operations within the wsdl:binding must contain soap:body elements that contain use=″literal″.
    - The wsdl:message referenced by the wsdl:operation input construct must have a single part.
    - The part must use the element attribute to reference an xsd:element.
    - The referenced xsd:element, or wrapper element, must have the same name as the wsdl:operation.
    - The wrapper element must not contain any xsd:attributes.

    In such cases, each parameter name is mapped from a nested xsd:element contained within wrapper element. The type of the parameter is mapped from the type of the nested xsd:element. For example:

    WSDL:

```
<xsd:element name="myMethod">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="param1" type="xsd:string"/>
   <xsd:element name="param2" type="xsd:int"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
...
<wsdl:message name="response"/>
 <part name="parameters" element="ns:myMethod"/>
</wsdl:message name="response"/>

<wsdl:message name="response"/>
...
<wsdl:operation name="myMethod">
 <input name="input" message="request"/>
 <output name="output" message="response"/>
</wsdl:operation>
```

    Java:

```
void myMethod(String param1, int param2) ...
```

  - **Parameter mapping**

    If the document and literal wrapped format is not detected, the parameter mapping follows the normal JAX-RPC mapping rules set in section 4.3.4 of the JAX-RPC specification.

    Each parameter is defined by a wsdl:message part referenced from the input and output elements.
    - A wsdl:part in the request wsdl:message is mapped to an input parameter.
    - A wsdl:part in the response wsdl:message is mapped to the return value. If multiple wsdl:parts exist in the response message, they are mapped to output parameters.

  - A Holder class is generated for each output parameter, as discussed in section 4.3.5 of the JAX-RPC specification.
- A wsdl:part that is both the request and response wsdl:message is mapped to an inout parameter.
  - A Holder class is generated for each inout parameter, as discussed in section 4.3.5 of the JAX-RPC specification.
  - The wsdl:operation parameterOrder attribute defines the order of the parameters.

XML:

```
<wsdl:message name="request">
 <part name="param1" type="xsd:string"/>
 <part name="param2" type="xsd:int"/>
</wsdl:message name="response"/>

<wsdl:message name="response"/>
...
<wsdl:operation name="myMethod" parameterOrder="param1, param2">
 <input name="input" message="request"/>
 <output name="output" message="response"/>
</wsdl:operation>


Java:

void myMethod(String param1, int param2) ...
```

– **Mapping of wsdl:binding**

The **WSDL2Java** command uses the wsdl:binding information to generate an implementation-specific client-side stub. WebSphere Application Server uses the wsdl:binding information on the server side to properly deserialize the request, invoke the Web service, and serialize the response. The information in the wsdl:binding does not affect the generation of the service endpoint interface, except when the document and literal wrapped format is used, or when MIME attachments are present.

- **MIME attachments**

  For a WSDL 1.1-compliant WSDL file, the part of an operation message, that is defined in the binding as a MIME attachment, becomes a parameter of the type of the attachment regardless of the part declared. For example:

  XML:

```
<wsdl:types>
 <schema ...>
  <complexType name="ArrayOfBinary">
   <restriction base="soapenc:Array">
     <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:binary[]"/>
   </restriction>
  </complexType>
 </schema>
</wsdl:types>

<wsdl:message name="request">
 <part name="param1" type="ns:ArrayOfBinary"/>
<wsdl:message name="response"/>

<wsdl:message name="response"/>
 ...

 <wsdl:operation name="myMethod">
  <input name="input" message="request"/>
  <output name="output" message="response"/>
 </wsdl:operation>
  ...

<binding ...
 <wsdl:operation name="myMethod">
```

```
  <input>
   <mime:multipartRelated>
    <mime:part>
     <mime:content part="param1" type="image/jpeg"/>
    </mime:part>
   </mime:multipartRelated>
  </input>
   ...
 </wsdl:operation>
```

Java:

```
void myMethod(java.awt.Image param1) ...
```

The JAX-RPC specification requires support for the following MIME types:

MIME type	Java type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
multipart/*	javax.mail.internet.MimeMultipart
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source

– **Mapping of wsdl:service**

  The wsdl:service element is mapped to a generated service interface. The generated service interface contains methods to access each of the ports in the wsdl:service element. The generated service interface is discussed in sections 4.3.9, 4.3.10, and 4.3.11 of the JAX-RPC specification.

  In addition, the wsdl:service element is mapped to the implementation-specific ServiceLocator class, which is an implementation of the generated service interface.

Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Mapping between WSDL and SOAP messages

The WSDL file defines the format of the SOAP message that are transmitted through network connections. The **WSDL2Java** command and the WebSphere Application Server runtime use the information in the WSDL file to ensure that the SOAP message is properly serialized and deserialized.

DOCUMENT versus RPC, LITERAL versus ENCODED

If a wsdl:binding element indicates that a message is sent using an RPC format, the SOAP message contains an element defining the operation. If a wsdl:binding element indicates that the message is sent using a document format, the SOAP message does not contain the operation element.

If the wsdl:part element is defined using the type attribute, the name and type of the part are used in the message. If the wsdl:part element is defined using the element attribute, the name and type of the element are used in the message. The element attribute is not supported by the JAX-RPC specification when use=″encoded″.

If a wsdl:binding element indicates that a message is encoded, the values in the message are sent with xsi:type information. If a wsdl:binding element indicates that a message is literal, the values in the message are typically not sent with xsi:type information. For example:

DOCUMENT/LITERAL
WSDL:

```
<xsd:element name="c" type="xsd:int"/>
<xsd:element name="method">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="a" type="xsd:string"/>
   <xsd:element ref="ns:c"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
...
  <wsdl:message name="request">
    <part name="parameters" element="ns:method"/>
  </wsdl:message>
 ...
 <wsdl:operation name="method">
  <input message="request"/>
 ...
```

Message:

```
<soap:body>
  <ns:method>
   <a>ABC</a>
   <c>123</a>
  <ns:method>
</soap:body>
```

RPC/ENCODED
WSDL:

```
<xsd:element name="c" type="xsd:int"/>

...
  <wsdl:message name="request">
    <part name="a" type="xsd:string"/>
    <part name="b" element="ns:c"/>
  </wsdl:message>
 ...
 <wsdl:operation name="method">
    <input message="request"/>
 ...
```

Message:

```
<soap:body>
  <ns:method>
   <a xsi:type="xsd:string">ABC</a>
   <element attribute is not permitted in rpc/encoded mode>
  </ns:method>
 </soap:body>
```

DOCUMENT/LITERAL not wrapped
WSDL:

```
<xsd:element name="c" type="xsd:int"/>

...
  <wsdl:message name="request">
    <part name="a" type="xsd:string"/>
    <part name="b" element="ns:c"/>
  </wsdl:message>
 ...
 <wsdl:operation name="method">
    <input message="request"/>

...
```

Message:

```
<soap:body>
  <a>ABC</a>
  <c>123</a>
</soap:body>
```

Developing JAX-RPC Web services deployment descriptor templates for a JavaBeans implementation

Deployment descriptors are standard text files, formatted using XML and packaged in a Web services application. Deployment descriptors are required to deploy Java API for XML-based RPC (JAX-RPC) Web services that are developed using Web Services for Java Platform, Enterprise Edition (Java EE) technology.

Before you begin

Develop a Web Services Description Language (WSDL) file.

You need a WSDL file to use Web services. You can develop your own WSDL file or get one from a Web services provider through e-mail, downloading, or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own.

About this task

Completing this task creates the deployment descriptors used to describe how to map the service implementation to a JavaBeans component for Java API for XML-based RPC (JAX-RPC) applications.

To develop the deployment descriptor templates from a WSDL file, you must obtain the Web address of the WSDL file.

If the WSDL file is a local file and you are running on the Windows platform, the Web address looks like this example: file:drive:\path\file_name.wsdl. If you are using the Linux or Unix platform, the Web address looks like this example: file:/path/file_name.wsdl. You can also specify local files using the absolute or relative file system path.

When the Web service is a JavaBeans implementation in a Web module, the webservices.xml,ibm-webservices-bnd.xmi and ibm-webservices.ext.xmi deployment descriptors and the JAX-RPC mapping file are generated in the WEB-INF subdirectory.

Run the **WSDL2Java -verbose -role develop-server -container web -genJava no *wsdlURL*** command to generate the server deployment descriptor templates and mapping file into the WEB-INF subdirectory. If the **-verbose** option is specified, a list of all the generated files is displayed when the command runs.

Note: The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Results

You have deployment descriptor templates that are required to implement or use JAX-RPC Web services.

Example

The following example uses a WSDL file named `AddressBookJ2WB.wsdl`:

Generate the template files:

```
WSDL2Java -verbose -role develop-server -container web -genJava no AddressBookJ2WB.wsdl
```

The deployment descriptor templates and mapping file are generated into the WEB-INF subdirectory:

```
Parsing XML file: AddressBookJ2WB.wsdl
Generating: WEB-INF\webservices.xml
Generating: WEB-INF\ibm-webservices-bnd.xmi
Generating: WEB-INF\ibm-webservices-ext.xmi
Generating: WEB-INF\AddressBookJ2WB_mapping.xml
```

What to do next

Now, you need to configure the webservices.xml deployment descriptor and configure the ibm-webservices-bnd.xmi deployment descriptor so that application server can process the incoming Web services. After you configure the deployment descriptors, you must assemble the Web services application for deployment.

WSDL2Java command for JAX-RPC applications

Run the WSDL2Java command-line tool against the WSDL file to create Java APIs and deployment descriptor templates.

The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform.

A Web Services Description Language (WSDL) file describes a Web service. The Java API for XML-based Remote Procedure Call (JAX-RPC) 1.1 specification defines a Java API mapping that interacts with the Web service. The Web Services for Java Platform, Enterprise Edition (Java EE) specification defines deployment descriptors that deploy a Web service in a Java EE environment. The **WSDL2Java** command is run against the WSDL file to create Java APIs and deployment descriptor templates according to these specifications.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the WSDL2Java tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. However, if you have an established WSDL or schema that relies on relative namespaces, under specific conditions you can use the allowRelativeNamespace property to disable the relative namespace restrictions in the WSDL2Java tool. For additional information, see the property description.

You can convert any relative namespaces to absolute namespaces. The following is an example of a relative namespace:

```
targetNamespace="MyRelNamespace"
```

. You can change the relative namespace in this example to an absolute namespace by adding the protocol and base URI information:

```
targetNamespace="http://www.sample.com/MyRelNamespace"
```

.

Command-line syntax

The command-line syntax is:

```
WSDL2Java [arguments] WSDL-URI
```

Required arguments

- WSDL-URI

  Specifies the location of the input WSDL file using a Universal Resource Identifier (URI). You can also use a regular file path if the WSDL file is on the local file system.

Important arguments

- -role *Java EE role*

  Specifies the Java EE development role that identifies which files to generate. Valid arguments include:
  - client

    A combination of the develop-client and deploy-client arguments.
  - deploy-client

    Generates binding files for client deployment.
  - deploy-server

    Generates binding files for server deployment.
  - develop-client (default)

    Generates files for client development.
  - develop-server

    Generates files for server development.
  - server

    A combination of the develop-server and deploy-server arguments.
- -container *Java EE-container*

  Indicates the Java EE container to use. Valid arguments include:
  - client

    Indicates client container.
  - ejb

    Indicates an Enterprise JavaBeans (EJB) container.
  - none

    Indicates no container.
  - web

    Indicates a Web container.

  For client roles (see the -role option), the default argument is `none`. For server roles, the container must be `ejb` or `web`. The same container option must be used for both development and deployment.
- -output *directory*

  Sets the root directory for emitted files.
- -inputMappingFile mapping file

  Specifies the file name of the Web Services for Java EE mapping file.
- -introspect

  Uses existing Java beans with a new Web service API.

  In some scenarios, it is good to use existing Java classes instead of generating new classes. The -introspect option directs the **WSDL2Java** command to examine existing Java classes when generating classes. The existing classes are validated against the JAX-RPC specification. For example:

  Suppose you have an existing Java bean

```
public class Bean {
 public Date x;
}
```

  The WSDL file defines *x* as xsd:dateTime. Without the -introspect option, the **WSDL2Java** command generates a Java bean that is similar to the following example:

```
public class Bean {
 private Calendar x;
 public void setx(Calendar value) (x=value;)
 public Calendar getX() { return x;)
}
```

  The **WSDL2Java** command uses the -introspect option to examine the original Java bean and to generate classes that are compatible with existing Java beans.
- -classpath *paths*

Defines an alternative class path to search for Java classes.

- -noDataBinding

  Disables the binding of XML types to Java types. Instead, each XML type is mapped to a javax.xml.soap.SOAPElement interface defined by the SOAP with Attachments API for Java (SAAJ) specification.

  The Java API for XML Web Services (JAX-WS) programming model supports SAAJ 1.2 and 1.3.

  The JAX-RPC programming model supports SAAJ 1.2.

  The Java programming models define Java mappings for a subset of XML types. Several XML types cannot be mapped to Java beans or primitives. In this situation, the **WSDL2Java** command maps the type to an SAAJ SOAPElement. A SAAJ SOAPElement is a generic representation of the element in the message. The methods on the SOAPElement can be used to examine the element and its children.

  In some scenarios, it might be more appropriate to use the generic SOAPElement mapping exclusively. To read more about the use of SOAPElement see SOAP with Attachments API for Java and Custom data binders.

  For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.
- -help

  Displays a help message and exits.
- -helpX

  Displays a help message for extended options. The options include:
- -verbose

  Displays processing information, including the names of the generated files.
- -NStoPkg *namespace=package*

  By default, package names are automatically derived from the namespace strings in the WSDL file. For example, if the namespace is of the form *http://x.y.com* or *urn:x.y.com*, the corresponding package is *com.y.x*.

  You can provide your own mapping by using the -NStoPkg argument, which you can repeat as often as necessary, once for each unique namespace mapping. For example, if a namespace in the WSDL file is called `urn:AddressFetcher2`, and you want files generated from the objects in this namespace to reside in the samples.addr package, provide the -NStoPkg ″http://urn:AddressFetcher2/″=samples.addr argument to the **WSDL2Java** command.
- -timeout *seconds*

  Specifies how long the **WSDL2Java** command waits, in seconds, for the WSDL-URI to respond before giving up. The default is 45 seconds; -1 disables the timeout.
- -genResolver

  Generates an absolute-import resolver class. The purpose of this class is to record the contents of the imported WSDL files that are used by the WSDL URI. This class is used by the run time and can also be used for future **WSDL2Java** command runs. This flexibility is desirable when the imported WSDL files are remote and possibly inaccessible. When an import resolver is used, the possibility that a remote WSDL file has different contents at run time that it did during development is eliminated. The generated class is named _AbsoluteImportResolver.java. Compile and package this class with the other Java classes that are generated by the **WSDL2Java** command.
- -useResolver *resolver-class*

  Specifies an absolute-import resolver class to use during parsing. This class must be created during a previous run of the **WSDL2Java** command that uses the -genResolver option. The class must be available in the CLASSPATH variable.
- -deployScope *argument*

  Indicates how to deploy the server implementation. Valid arguments include:
  - Application

    Uses one instance of the implementation class for all requests.
  - Request

    Creates a new instance of the implementation class for each request.

– Session

  Creates a new instance of the implementation class for each session.

Other arguments

- -user *id*

  Specifies the login user name to access the WSDL URI.
- -password *password*

  Specifies the login user password to access the WSDL URI.
- -all

  Generates Java files for all types, even those that are not referenced.
- -allowRelativeNamespace *true or false*

  Specifies whether to disable the relative namespace restrictions. If you specify `-allowRelativeNamespace=true`, the relative namespace restrictions are disabled.

  Note: Only use this property if you have an established WSDL file or schema that relies on a relative namespaces and you are seeking to interoperate with a defined set of vendors that permit the use of relative namespaces.
- -debug

  Prints debugging information.
- -genJava *argument*

  Generates Java files. Valid arguments include:
  – IfNotExists, default
  – Overwrite
  – No
- -javaSearch *argument*

  The -javaSearch option is used with the -genJava option. If the -genJava IfNotExists, use the -javaSearch option to determine how file existence is detected.
  – File (default): Looks for a file in the output directory
  – Classpath: Looks for a class in the CLASSPATH variable
  – Both: Looks for a file in the output directory or in a class in the CLASSPATH variable
- -genXML *argument*

  Generates the `.xml` and `.xmi` files. Valid arguments are:
  – IfNotExists, default
  – Overwrite
  – No
- -genImplSer *true or false*

  Indicates that each generated Java bean implements the java.io.Serializable. The default is `false`.
- -genEquals *true or false*

  Indicates that each generated Java bean have `equals` and `hashCode` methods. The default is false.
- -noWrappedOperations

  Disables wrapped operations detection. Java beans for the request and response messages are generated.
- -noWrappedArrays

  Disables wrapped array detection.
- -fileNStoPkg *file name*

  Specifies the file of the namespace to package mappings. The default is NStoPKG.properties.
- service *wsdl service name*

  Generates files for the installed WSDL service only.

- -testCase

  Generates the template for a JUnit test case for testing Web services. JUnit is a simple framework to write repeatable tests.

Developing JAX-RPC Web services deployment descriptor templates for an enterprise bean implementation

You can develop deployment descriptor templates for an Enterprise JavaBeans (EJB) implementation that is enabled for Java API for XML-based RPC (JAX-RPC) Web services.

Before you begin

You need to create a service endpoint interface and develop a Web Services Description Language (WSDL) file before you can develop the deployment descriptor templates because the service endpoint interface and the WSDL file are artifacts that are used to create the templates.

About this task

Completing this task creates deployment descriptor templates that describe how to map the service implementation to a Enterprise JavaBeans (EJB). This task is a required step in developing a Web service from an enterprise bean.

To develop the deployment descriptor templates from a WSDL file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

When the Web service implementation contains an enterprise bean in an EJB module, the `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file are generated in the `META-INF` subdirectory.

Run the **WSDL2Java -verbose -role develop-server -container ejb -genJava no *wsdlURL*** command to generate the server deployment descriptor templates and mapping file into the `META-INF` subdirectory. If the -verbose option is specified, a list of all generated files displays when the command runs.

Note: The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly provided with the z/OS version of the product. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Results

You have deployment descriptor templates that are required to implement a Web service.

Example

The following example uses the `AddressBookJ2WE.wsdl` WSDL file:

1. Generate the template files with the following command syntax:

   ```
   WSDL2Java -verbose -role develop-server -container ejb -genJava no AddressBookJ2WE.wsdl
   ```

   The deployment descriptor templates are generated into the `META-INF` subdirectory as follows:

```
Parsing XML file: AddressBookJ2WE.wsdl
Generating: META-INF\webservices.xml
Generating: META-INF\ibm-webservices-bnd.xmi
Generating: META-INF\ibm-webservices-ext.xmi
Generating: META-INF\AddressBookJ2WE_mapping.xml
```

What to do next

Continue to complete the steps that are necessary to develop a JAX-RPC Web service from an enterprise bean. The next step is to complete the EJB implementation. When you complete the EJB implementation, you assemble an enterprise bean Java archive (JAR) file that contains the enterprise bean and supporting classes created from a WSDL file.

Completing the JavaBeans implementation for JAX-RPC applications

After you have developed the Java artifacts necessary to develop a Java API for XML-based RPC (JAX-RPC) Web service, you must complete the JavaBeans implementation to assemble a Java archive (JAR) file or a Web archive (WAR) file based on your programming model. The resulting JAR file or WAR file contains the JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Develop Web services deployment descriptor templates for a JavaBeans implementation using the wsdl2java command-line tool. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans implementation.

About this task

For JAX-RPC applications, complete the JavaBeans implementation by writing your business application.

1. Edit the JavaBeans implementation template, *binding*Impl.java. The *binding* is the name of the <wsdl:binding> element in the WSDL file. The JavaBeans implementation is generated by the wsdl2java command-line tool.
   a. Complete the implementation of the methods in the template.
   b. (Optional) Make changes if necessary.
   c. (Optional) Change the class name if the binding name is not acceptable.
2. Compile all the Java classes.
3. Assemble a Web archive (WAR) file. Assemble all the Java classes into a WAR file using Web module assembly tools. Include all of the classes generated from running the wsdl2java command tool for JAX-RPC Web service applications when developing implementation templates and bindings from a WSDL file.

Results

You have now enabled the JavaBeans-based business application for JAX-RPC Web services. You have a JAR file or a WAR file containing the JavaBeans implementation and supported classes created from the WSDL file.

.

What to do next

If you are developing a JAX-RPC Web services application from JavaBeans, you need to configure the `webservices.xml` deployment descriptor and configure the `ibm-webservices-bnd.xmi` deployment descriptor so that the application server can process the incoming Web services requests.

Completing the EJB implementation for JAX-RPC applications

After you have developed the Java artifacts necessary to develop a Java API for XML-based RPC (JAX-RPC) Web service, you must complete the Enterprise JavaBeans (EJB) implementation to assemble a Java archive (JAR) file or a Web archive (WAR) file based on your programming model. The resulting JAR file or WAR file contains the Enterprise JavaBeans (EJB) implementation and the supported classes created from the tooling.

Before you begin

Develop EJB implementation templates and bindings from a WSDL file for JAX-RPC Web services using the **wsdl2java** command-line tool. The deployment descriptor templates that are generated from a Web Services Description Language (WSDL) file are required to complete the EJB implementation in the Web services development process.

About this task

For JAX-RPC applications, complete the enterprise beans implementation by writing your business application.

1. Inspect the EJB remote interface template, *portType*_RI.java. If necessary, modify the template. The value *portType* is the name of the <wsdl:portType> element in the WSDL file.
2. Edit the *binding*Impl.java EJB implementation template. Where *binding* is the name of the <wsdl:binding> element in the WSDL file.
3. Complete the implementation of the methods in the template.
4. (Optional) Make changes if necessary.
5. (Optional) Change the class name if the binding name is not acceptable.
6. Compile all the Java classes.
7. Assemble an EJB Java archive (JAR) file. Assemble all the Java classes into an enterprise bean JAR file using assembly tools. Include all of the classes generated from running the WSDL2Java command tool when developing implementation templates and bindings from a WSDL file.

Results

You have enabled an enterprise beans business application for JAX-RPC Web services. You now have an enterprise bean JAR file containing an EJB and supporting classes created from Web services artifacts.

What to do next

Now that you have gathered the required artifacts for developing a JAX-RPC Web service with an enterprise bean, you need to, configure the webservices.xml deployment descriptor.

Configuring the webservices.xml deployment descriptor for JAX-RPC Web services

You can configure the webservices.xml deployment descriptor with an assembly tool.

Before you begin

To configure the client deployment descriptor, read about the configuring the client deployment descriptor in the Rational Application Developer documentation.

Before you can configure the ibm-webservices-bnd.xml deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

About this task

For JAX-RPC Web services, this task is one of the required steps in developing a Web service. You need to configure the deployment descriptors so that the application server can process the incoming Web services requests.

If you are developing a Web service from JavaBeans, you can develop Web services JavaBeans deployment descriptor templates from a Web Services Description Language (WSDL) file. Then, you complete the JavaBeans implementation. To learn more, read about developing JavaBeans deployment descriptor templates from a WSDL file and completing the JavaBeans implementation.

If you are developing a Web service from an enterprise bean, you can develop Web services Enterprise JavaBeans (EJB) deployment descriptor templates from a WSDL file. Then, you complete the EJB implementation. To learn more, read about developing Web services EJB deployment descriptor templates from a WSDL file and completing the EJB implementation.

When the JavaBeans implementation is complete, the Web module Web archive (WAR) file is assembled. When the EJB implementation is complete, the enterprise bean Java archive (JAR) file is assembled. These archive files contain the webservices.xml deployment descriptor. The archive files must be assembled before you can configure the webservices.xml deployment descriptor.

The assembly tools provide a graphical interface for developing code artifacts, assembling the code artifacts into various archives (modules) and configuring compliant deployment descriptors for Java Platform, Enterprise Edition (Java EE).

Configure the webservices.xml deployment descriptor by following the steps provided in this task section.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer documentation.

Results

You have a webservices.xml deployment descriptor that is configured.

What to do next

For JAX-RPC Web services, you must configure the `ibm-webservices-bnd.xmi` deployment descriptor.

Configuring the webservices.xml deployment descriptor for handler classes

You can use an assembly tool to configure the `webservices.xml` deployment descriptor for user-provided handler classes.

Before you begin

You can configure deployment descriptors with assembly tools provided with the application server.

A *handler class* is a class that is written to modify a SOAP message that represents a remote procedure call (RPC) request or response. Handlers can be associated with a Web service or a Web service client.

Similar to Java API for XML-based RPC (JAX-RPC) Web services, you can use deployment descriptors to describe Java API for XML Web Services (JAX-WS) Web services. For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

To complete this task, you need an enterprise archive (EAR) file for the applications that you want to configure. For some handler use, such as logging or tracing, only the server or client application require configuration. For other handler use, including sending information in the SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file contain the handler classes to configure. These classes implement the javax.xml.rpc.handler.Handler interface. For more information on writing handler classes, see chapter 6 of the Web Services for Java EE specification. See chapter 9 in the JAX-WS specification or chapter 12 in the JAX-RPC specification for additional information on the handler framework for your programming model. The application modules must contain the `webservices.xml` deployment descriptor. See the Web services specifications and API documentation to review the JAX-RPC specification along with a complete list of the supported standards and specifications.

About this task

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer documentation.

Configuring the ibm-webservices-bnd.xmi deployment descriptor for JAX-RPC Web services

Use assembly tools to configure the ibm-webservices-bnd.xml deployment descriptor. This file stores binding information that is associated with the endpoints defined with the webservices.xml deployment descriptor file.

Before you begin

Note: The `ibm-webservices-bnd.xmi` deployment descriptor is for Java API for XML-based RPC (JAX-RPC) based Web services application. It is not used for Java API for XML-Based Web Services (JAX-WS) enabled applications.

To configure the client deployment descriptor, read about the configuring the client deployment descriptor in the Rational Application Developer documentation.

Before you can configure the `ibm-webservices-bnd.xml` deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

About this task

This task is one of the steps in developing a Web service. You need to configure the deployment descriptors so that WebSphere Application Server can process the incoming Web services requests.

Depending on if you are developing a Web service from JavaBeans or an enterprise bean:
- Develop Web services JavaBeans deployment descriptor templates from a Web Services Description Language (WSDL) file.
- Develop Web services Enterprise JavaBeans (EJB) deployment descriptor templates from a WSDL file.

Then, complete the EJB implementation or complete the JavaBeans implementation. When the EJB implementation is complete, the enterprise bean Java archive (JAR) file is assembled. When the JavaBeans implementation is complete, the Web module Web archive (WAR) file is assembled. These archive files contain the `webservices.xml` deployment descriptor. The archive files must be assembled before you can configure the `webservices.xml` deployment descriptor.

Configure the `webservices.xml` deployment descriptor by following the steps provided in this task section.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer documentation.

Results

The `ibm-webservices-bnd.xmi` deployment descriptor is configured for the Web service implementation module.

What to do next

If you are developing a Web service from JavaBeans, assemble a WAR file that is enabled for Web services from Java code.

If you are developing a Web service from an enterprise bean, assemble a JAR file that is enabled for Web services from an enterprise bean. In this task you assemble the artifacts that are required to enable the EJB module for Web services into the JAR file.

JAX-RPC Web services enabled module - deployment descriptor settings (ibm-webservices-bnd.xmi file)

This article is an introduction to the `ibm-webservices-bnd.xmi` deployment descriptor file for Java API for XML-based RPC (JAX-RPC) Web services.

The `ibm-webservices-bnd.xmi` file is a deployment descriptor for a Web services-enabled Web module or an Enterprise JavaBeans (EJB) module. This file contains information for the Web services run time that is required by WebSphere Application Server.

You can edit these properties using an assembly tool. See Configuring the `ibm-webservices-bnd.xmi` deployment descriptor for instructions.

The following user-defined assembly properties are supported:

- **wsDescNameLink**

  Attribute of the wsdescBindings element that specifies the link to the corresponding <webservice-description-name> element in the `webservices.xml` file.
- **pc-name-link**

  Attribute of the pcBindings element that specifies the link to the <port-component-name> element in the `webservices.xml` file.
- **scope**

  Attribute of the pcBindings element that specifies when new instances of implementation beans are created. Possible values are request, session, and application.

You can change scope value for a deployed Web service using the administrative console. Click **Enterprise Applications** > *application* > **Web modules** or **EJB modules** > *module* > **Web Services Implementation Scope**.

Bindings file examples

The following examples demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute with a unique string value. Review the template xmi files generated by the **WSDL2Java** command for examples of ID strings. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

```
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0" xmlns:xmi=
"http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
 <wsdescBindings wsDescNameLink="AddressBookService">
  <pcBindings pcNameLink="AddressBook" scope="Application"/>
 </wsdescBindings>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Using WSDL EJB bindings to invoke an EJB from a Web services client

WebSphere Application Server supports directly accessing an Enterprise JavaBeans (EJB) as a Web service, as an alternative to using HTTP or Java Message Service (JMS) to transport requests between the server and the client.

Before you begin

You need an EJB that you can directly access as a Web service.

About this task

You can achieve this task because of a multiprotocol technology that uses Java API for XML-based remote procedure call (JAX-RPC) and Remote Method Invocation over Internet Inter-ORB Protocol (RMI/-IIOP) together.

RMI-IIOP with JAX-RPC supports WebSphere Java clients to invoke enterprise beans with a WSDL file and the JAX-RPC programming model instead of the standard Java EE programming model. When a Web service is implemented by an enterprise bean, multiprotocol JAX-RPC permits the Web service invocation path to be optimized for WebSphere Java clients.

This method yields better performance and enables you to get support for client transactions, which are not standard for Web services.

To use EJB bindings of Web Services Description Language (WSDL) files to transport Web services requests:

1. (Optional) Create a WSDL file that contains non-SOAP protocol bindings.

   You can use the -bindingTypes option of the **Java2WSDL** command to create a WSDL file that contains non-SOAP protocol bindings. The -bindingTypes option specifies the binding types to write to the output of the WSDL document. Review the Java2WSDL article for more information on using the -bindingTypes option. The following command is an example that you can use to generate SOAP over HTTP, and EJB bindings for a service endpoint interface, my.pkg.MySEI and an EJB implementation, my.pkg.MyEJBClass:

   ```
   java2wsdl -bindingTypes http,ejb -implClass my.pkg.MyEJBClass my.pkg.MySEI
   ```

2. (Optional) Obtain an existing WSDL file to add the EJB binding to.
3. Add an EJB binding to the WSDL file.
4. Add a port address that contains an endpoint using the wsejb prefix.
5. Deploy the Web services application.
6. Configure the endpoint URL information for EJB bindings.

   The WSDL publisher uses this partial Web address string to produce the actual enterprise bean Web address for each port component that is defined in the enterprise bean JAR file. The published WSDL file can be used by clients that need to invoke the Web service.

Results

You have an EJB that can be accessed by a Web services client that uses the JAX-RPC programming model. The RMI-IIOP protocol is used instead of SOAP over HTTP

What to do next

Publish the WSDL file.

EJB endpoint URL syntax

An Enterprise JavaBean (EJB) endpoint URL is used to access a Web service with the EJB Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) transport. The URL specifies the EJB endpoint, including the EJB home class, the EJB Java Naming and Directory Interface (JNDI) name, and optional properties.

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

An EJB endpoint URL has the following format:

```
wsejb:/[classname]?<property>=<value>&amp;<property>=<value>&amp;...
```

Where:

- wsejb is the transport type
- *classname* is the name of the home interface class associated with the EJB to be invoked
- *property* and *value* pairs represent the set of required and optional properties. These properties are used to set certain values in the EJB endpoint URL. The various properties and definitions are described in the table.

JNDI-related properties

Property name	Description
jndiName	Specifies the JNDI name of the EJB. This property is required.
initialContextFactory	Specifies the name of the JNDI initial context factory. This property is optional
jndiProviderURL	Specifies the JNDI provider URL. This property is optional.

Example: Developing and deploying a JAX-RPC Web service from an existing application

You can develop a JAX-RPC Web service from an Enterprise JavaBeans (EJB) or JavaBeans implementation. The development process is based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification.

1. **Select the enterprise bean or JavaBeans implementation that you want to enable as a JAX-RPC Web service.**

   The implementation must meet the following Web Services for Java EE specification requirements:

   - It must have methods that can be mapped to a service endpoint interface. See step 2 for more information.
   - It must be a stateless session EJB implementation or a JavaBeans implementation without client-specific state, because the implementation bean might be selected to process a request from any client. If a client-specific state is required, a client identifier must be passed as a parameter of the Web services operation.

     The selected methods of an enterprise bean must not have a transaction attribute of mandatory, because no standard currently exists, for these Web services transactions.

     A JavaBeans implementation in a Web container requires the following contents:

     – A public default constructor
     – Exposed public methods
     – It must not save a client-specific state between method calls
     – It must be a public, non-final, and non-abstract class
     – It must not define a finalize method

2. **Develop a service endpoint interface**.

   Developing a Web service requires a service endpoint interface.

   If you are using an EJB implementation, develop a service endpoint interface from an EJB remote interface.

   If you are using a JavaBeans implementation, develop a service endpoint interface for a JavaBeans implementation.

3. Develop a Web Services Description Language (WSDL) file.

4. **Develop deployment descriptor templates**.

   If you are using an EJB implementation, develop Web services deployment descriptor templates from an EJB implementation.

   If you are using a JavaBeans implementation, develop Web services deployment descriptor templates for a JavaBeans implementation.

5. **Configure the deployment descriptors.**

   By setting the `ejb-link` or `servlet-link` values of the service-impl-bean elements you can link to the enterprise bean or JavaBeans implementation that implement the service.

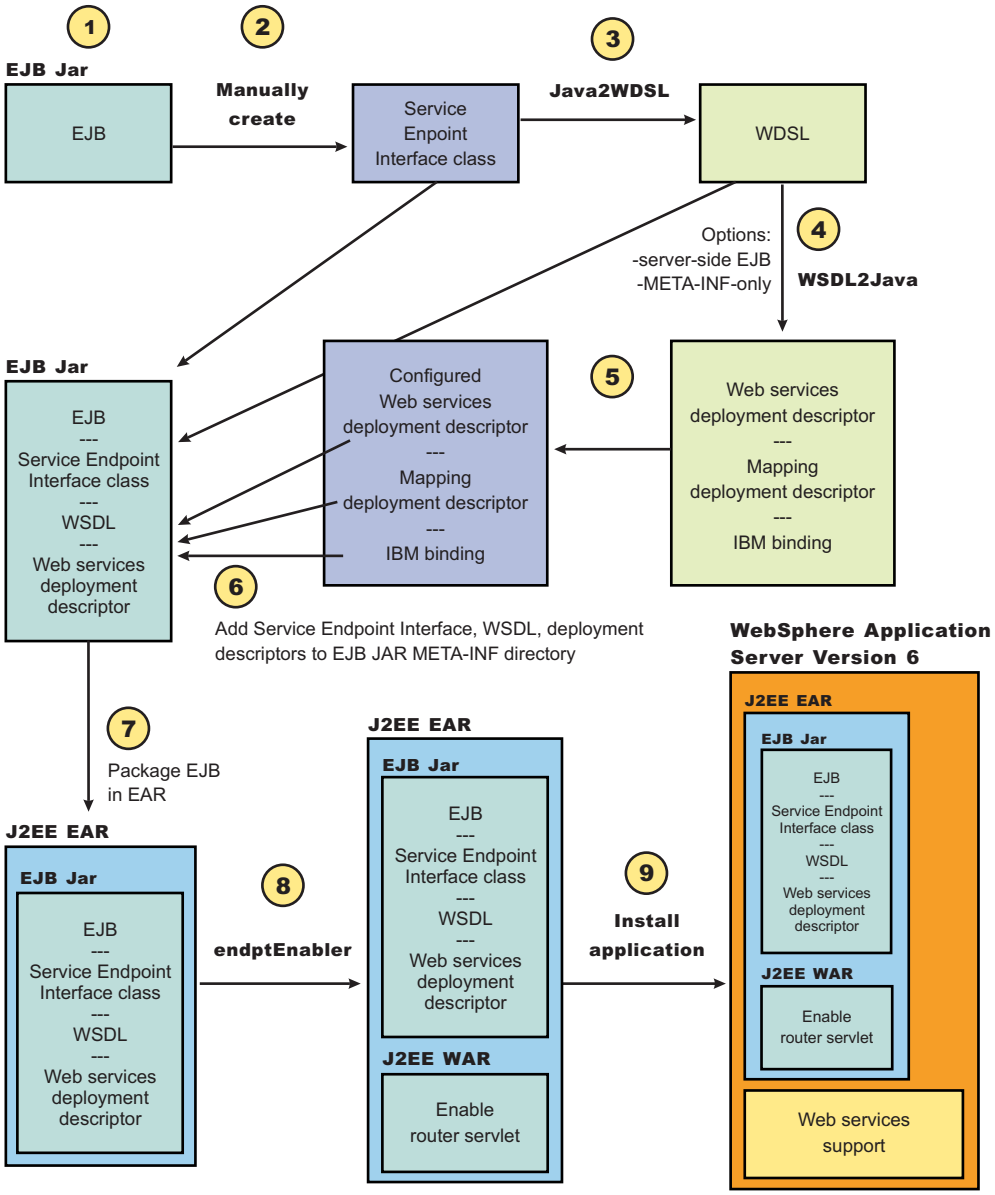
   Configure the `webservices.xml` deployment descriptor.

Configure the `ibm-webservices-bnd.xmi` deployment descriptor.

6. Assemble an enterprise archive (EAR) file from a JAR file or assemble an EAR file from a WAR file.
7. Enable the Web service-enabled EAR file.

   This step only applies if you are using an EJB implementation.
8. Deploy the Web service application.
9. Publish the WSDL file.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.



Developing Web services applications from existing WSDL files with JAX-RPC

You can develop a Web service with an existing Web Services Description Language (WSDL) file using the Java API for XML-based RPC (JAX-RPC) programming model.

Before you begin

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

Locate the WSDL file that defines the Web service that you want to implement. You can develop a WSDL or obtain one from an existing Web service through e-mail, downloading or a Uniform Resource Locator (URL).

About this task

To develop Web services based on the JAX-RPC programming model, you can use a bottom-up development approach starting from existing JavaBeans or enterprise beans or you can use a top-down development approach starting with an existing Web Services Description Language (WSDL) file. This task describes the steps when using the top-down development approach.

When developing a JAX-RPC Web service starting from an existing WSDL file, create the JavaBeans or enterprise bean and artifacts that enable the bean as Web services and assemble all artifacts that are required for the Web service, and deploy the application onto the application server.

Considerations when using JavaBeans

JavaBeans exposed as JAX-RPC Web services are supported only over an HTTP transport.

Considerations when using enterprise beans

- The enterprise bean must be a stateless session bean.
- Enterprise beans that are exposed as JAX-RPC Web services must be packaged in EJB 2.1 or in EJB 3.0 or higher modules.
- For JAX-RPC Web services using EJB 2.1 style endpoints, the Web service method parameters must be one of the supported JAX-RPC types. These requirements are documented in the JAX-RPC specification.
- JAX-RPC Web services using enterprise beans are supported over an HTTP or Java Message Service (JMS) transport.

Note: It is a best practice to use EJB 2.1 style enterprise beans with JAX-RPC applications.

1. Set up a development environment for Web services. You do not have to set up a development environment if you are using Rational Application Developer.
2. Develop the Java artifacts from a WSDL file. You need to create the deployment descriptor templates and bindings that are configured to map the service implementation to the JavaBeans or enterprise beans implementation.
   - Develop JavaBeans artifacts from a WSDL file.
   - Develop enterprise beans artifacts from a WSDL file.
3. Complete the implementation of your Web service application.
   - For JavaBeans applications, complete the JavaBeans implementation.
   - For enterprise beans applications, complete the enterprise beans implementation.
4. Configure the `webservices.xml` deployment descriptor. For JAX-RPC Web services, configure the webservices.xml deployment descriptor so that the application server can process the incoming Web services requests.

5. Configure the `ibm-webservices-bnd.xmi` deployment descriptor. Configure the `ibm-webservices-bnd.xml` deployment descriptor so that the application server can process the incoming Web services requests.
6. Assemble the artifacts for your Web service.

   Use assembly tools provided with the application server to assemble your Java-based Web services modules.

   If you have assembled an Enterprise Archive (EAR) file that contains enterprise beans modules that contain Web services, use the endptEnabler command-line tool or an assembly tool before deployment to produce a Web services endpoint WAR file. This tool is also used to specify whether the Web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.
7. Deploy the EAR file into the application server. You can now deploy the EAR file that has been configured and enabled for JAX-RPC Web services onto the application server.

Results

You have developed a JAX-RPC Web service application by starting with an existing WSDL file.

What to do next

After you deploy the EAR file, test the Web service to make sure that it works with the application server.

Developing Java artifacts for JAX-RPC applications from a WSDL file

You can develop Java artifacts from a Web Services Description (WSDL) file for JAX-RPC applications from a WSDL file by using the WSDL2Java command-line tool to create Java implementation templates and bindings.

Before you begin

To develop the JavaBeans implementation templates and bindings from a WSDL file, you must obtain the Uniform Resource Locator (URL) of the WSDL file.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

Implementation templates are generated using the -role develop-server option of the WSDL2Java command. The WSDL2Java command also generates bindings and deployment descriptors.

The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

About this task

Develop JavaBeans implementation templates and bindings from a WSDL file by issuing the proper command.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the WSDL2Java tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. You can

convert any relative namespaces to absolute namespaces. To learn more about the use of namespaces with the WSDL2Java tool, see the WSDL2Java command for JAX-RPC applications documentation.

Run the **WSDL2Java -verbose -role develop-server -container web *wsdlURL*** command. Since the -verbose option is specified, a list of all the generated files is displayed when the command runs.

Results

You have templates for the implementation and deployment descriptors required to implement a Web service, as well as bindings files. These templates are partially filled with information from the WSDL file.

Example

The following example uses the AddressBook JavaBeans implementation and the `AddressBook.wsdl` WSDL file. After generating the template files from the **WSDL2Java -verbose -role develop-server -container web AddressBook.wsdl** command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookSoapBindingImpl.java..
WSWS3282I: Info: Generating WEB-INF\webservices.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating WEB-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-ext.xmi.
```

The `AddressBookSOAPBindingImpl.java` file is the template for the implementation bean. It is named after the port in the WSDL file. Generally, this class is renamed to a more meaningful name.

What to do next

Complete the Java bean implementation.

Developing EJB implementation templates and bindings from a WSDL file for JAX-RPC Web services

You can develop Enterprise JavaBeans (EJB) implementation deployment descriptor templates and bindings from a Web Services Description Language (WSDL) file for a JAX-RPC application.

Before you begin

To develop EJB implementation templates and bindings from a WSDL file for a Java API for XML-based RPC (JAX-RPC) Web service, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If the WSDL file is a local file, the URL looks like the following example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

About this task

This task is one a required step in developing a Web service from an enterprise bean.

Implementation templates are generated using the -role develop-server option of the WSDL2Java command.

Templates are generated for an EJB implementation for the following components:
- enterprise bean
- EJB remote interface
- EJB Home

The WSDL2Java command also generates bindings and deployment descriptors.

The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the WSDL2Java tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. You can convert any relative namespaces to absolute namespaces. To learn more about the use of namespaces with the WSDL2Java tool, see the WSDL2Java command for JAX-RPC applications documentation.

Run the **WSDL2Java -verbose -role develop-server -container ejb *wsdlURL*** command. Because the verbose option is specified, a list of all the generated files is displayed when the command runs.

Results

You have templates for the implementation and deployment descriptors required to implement Web services, as well as bindings files. These templates are partially completed with information from the WSDL file.

Example

The following example uses the enterprise bean AddressBook enterprise bean and the `AddressBook.wsdl` file. After generating the template files from the **WSDL2Java -verbose -role develop-server -container EJB AddressBook.wsdl** command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookSoapBindingImpl.java.
WSWS3282I: Info: Generating addr\AddressBook_RI.java.
WSWS3282I: Info: Generating addr\AddressBookHome.java.
WSWS3282I: Info: Generating META-INF\webservices.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservices-ext.xmi.
```

What to do next

Complete the EJB implementation. When you complete the EJB implementation, an EJB Java archive (JAR) file that contains an EJB and supporting classes is created from a WSDL file.

Developing and deploying JAX-RPC Web services clients

You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-based RPC (JAX-RPC) programming model.

Before you begin

Note: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of Web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications and clients.

About this task

Developing Web services clients based on the JAX-RPC programming model

The Web services client programming model provides the guidelines for accessing Web services in a Java EE environment. You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-based remote procedure call (JAX-RPC) specification. The application server supports Enterprise JavaBeans (EJB) clients, Java EE application clients, JavaServer Pages (JSP) files and servlets that are based on the JAX-RPC programming model.

Managed and unmanaged JAX-RPC Web services clients

The application server supports both managed and unmanaged Web Services clients when using the JAX-RPC programming model:

- Managed clients

  Web services for Java EE clients are defined by Java Specification Requirements (JSR) 109 and are managed clients because they run in a Java EE container. These clients are packaged as enterprise archive (EAR) files and contain components that act as service requesters. These components are comprised of a Java EE client application, a Web component such as a servlet or JavaServer Pages (JSP), or a session Enterprise JavaBeans (EJB). Web services managed clients use JSR 109 APIs and deployment information to look up and invoke a Web service.

  For the managed clients, the service look up is through Java Naming and Directory Interface (JNDI) lookup. Read about setting up UserName token Web services security, digital signature Web services security and Lightweight Third-Party Authentication (LTPA) token Web services security. The following code is an example of a context lookup that is JSR 109 compliant:

  ```
  InitialContext ctx = new InitialContext();
      FredsBankServiceLocator locator
  =(FredsBankService)ctx.lookup("java:comp/env/service/FredsBankService");
      FredsBank fb = locator.getFredsBank(url);
      long balance = fb.getBalance();
  ```

  When you are instantiating a context lookup for a managed client, do not use new() for the service locator. Here is an example that is not JSR 109 compliant (new ServiceLocator):

  ```
  Properties prop = new Properties();
      InitialContext ctx = new InitialContext(prop);
      FredsBankServiceLocator locator = new FredsBankServiceLocator();
      FredsBank fb = locator.getFredsBank(url);
      long balance = fb.getBalance();
  ```

  Without the lookup() call, the client has no access to the deployment descriptor. For JAX-RPC Web services, the Web services security configuration is in the Web services deployment descriptor.

- Unmanaged clients

  Java Platform, Standard Edition (Java SE 6) clients that use the JAX-RPC runtime environment to invoke Web services and do not run in any Java EE container are known as unmanaged clients. A Web services unmanaged client is a stand-alone Java client that can directly inspect a

WSDL file and formulate the calls to the Web service by using the JAX-RPC APIs directly. These clients are packaged as JAR files which do not contain any deployment information.

For a Java application to act as a Web service client, a mapping between the WSDL file and the Java application must exist. For JAX-RPC Web services, the mapping is defined by the JAX-RPC specification. You can use a Java component to implement a Web service by specifying the component interface and binding information in the WSDL file and designing the application server infrastructure to accept the service request. This entire process is based on the Web Services for Java EE specification. The JAX-RPC specification defines the mapping between a WSDL file, Java code, and XML Schema types.

1. Obtain the Web Services Description Language (WSDL) document for the Web service that you want to access.

   You can locate the WSDL from the services provider through e-mail, through a Uniform Resource Locator (URL) or by looking it up in a Universal Description, Discovery and Integration (UDDI) registry.
2. Develop client bindings from a WSDL file using the WSDL2Java command-line tool. The information needed to invoke the Web service is generated, including the service endpoint interface and implementations, the generated service interface and the `ibm-webservicesclient-bnd.xmi` and `ibm-webservicesclient-ext.xmi` deployment descriptors.
3. Complete the client implementation. Write your client application code that is used to invoke the Web service.

   See Chapter 4 of the JSR 109 specification. For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

   Note: If an application creates a number of threads in the JSR 109 client, the metadata (including the WebSphere Application Server configuration) is not copied to the thread, and the Global Security Handler is not called.

   You can review the JAX-RPC-based Web services sample, GetQuote client, in the WebServicesSamples application that is available in the Samples Gallery.
4. (Optional) Assemble a Web services-enabled client Java archive (JAR) file into an enterprise archive (EAR) file. Complete this step if you are developing a managed JAX-RPC Web services client that runs in the Java EE client container.
5. (Optional) Assemble a Web services-enabled client Web archive (WAR) file into an enterprise archive (EAR) file. Complete this step if you are developing a managed JAX-RPC Web services client that runs in the Java EE client container.
6. (Optional) Configure the client deployment descriptor. Complete this step if you are developing a managed JAX-RPC client.
7. (Optional) Configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor. Complete this step if you are deploying a managed JAX-RPC client that runs in the Java EE client container, and you want to override the default client settings. See ibm-webservicesclient-bnd.xmi assembly properties for more information about the `ibm-webservicesclient-bnd.xmi` deployment descriptor.
8. (Optional) Deploy the Web services client application. Complete this step to deploy a managed JAX-RPC Web services client that runs in the Java EE client container.
9. Test the Web services-enabled client application. You can test an unmanaged client JAR file or a managed client application.

Results

You have created and tested a Web services client application.

What to do next

After you develop a Web services application client, and the client is statically bound, the service endpoint used by the implementation is the one that is identified in the WSDL file that you used during the

development process. During or after installation of the Web services application, you might want to change the service endpoint. For managed clients, you can change the endpoint with the administrative console or the wsadmin scripting tool.

You can additionally consider customizing your Web service by implementing extensions to your Web services client. Some examples of these extensions include sending and receiving values in SOAP headers, sending and receiving HTTP or JMS transport headers, or using custom bindings. To learn more about these extensions, read about implementing extensions to Web services clients.

Developing client bindings from a WSDL file for a JAX-RPC client

You can develop client bindings from a Web Services Description (WSDL) file for a JAX-RPC client.

Before you begin

To develop the client bindings from a WSDL file for JAX-RPC Web service applications, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use. You need bindings and deployment descriptors in order for a client to use a Web service.

If the WSDL file is a local file, the URL looks like the following example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

Client bindings are generated using the -role develop-client option in combination with the -container option of the **WSDL2Java** command. The -container option takes the following parameters:

- **-container client**

  Generates bindings and deployment descriptors for a client residing in the application client container.
- **-container ejb**

  Generates bindings and deployment descriptors for a client that is an enterprise bean in the Enterprise JavaBeans (EJB) module.
- **-container web**

  Generates bindings and deployment descriptors for a client residing in the Web container.

The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

About this task

Develop client bindings from a WSDL file by running the appropriate command.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the WSDL2Java tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. You can convert any relative namespaces to absolute namespaces. To learn more about the use of namespaces with the WSDL2Java tool, see the WSDL2Java command for JAX-RPC applications documentation.

Run the **WSDL2Java -verbose -role develop-client -container *type wsdlURL*** command,
where *type* is **ejb** for an enterprise EJB client, **web** for a JavaBeans client, or **client** for an application client.
You can use the following combinations in the command-line:

- -container web

- -container `ejb`
- -container `client`

Because the verbose option is specified, a list of all generated files is displayed when the command runs.

Results

You have the bindings and deployment descriptors needed by a client to use a Web service.

Example

The following example uses the AddressBook enterprise bean the `AddressBook.wsdl` WSDL file. After generating the bindings from the **WSDL2Java -verbose -role develop-client -container client AddressBook.wsdl** command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookService.java.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-ext.xmi.
```

What to do next

Complete the client implementation by writing your client application and then assembling the client artifacts.

.

Changing SOAP message encoding to support WSI-Basic Profile

Support for Universal Transformation Format (UTF)-16 encoding is required by the WS-I Basic Profile 1.0. WebSphere Application Server conforms to the WS-I Basic Profile 1.1. UTF-16 is a kind of unicode encoding scheme using 16-bit values to store Universal Character Set (UCS) characters. UTF-8 is the most common encoding that is used on the Internet and UTF-16 encoding is typically used for Java and Windows product applications. You can change the encoding in a SOAP message from UTF-8 to UTF-16.

Before you begin

To learn more about the requirements of the Web Services-Interoperability Basic Profile (WS-I), including UTF-16, see Web Services-Interoperability Basic Profile.

About this task

Support for UTF-16 encoding is required by WS-I Basic Profile. The application server only supports UTF-8 and UTF-16 encoding of SOAP messages.

You can change the character encoding in one of two ways:

- Use a property on the Stub for users to set.

  This choice applies to the client only.

  For a client, the encoding is specified in the SOAP request. The SOAP engine serializes the request and sends it to the Web service engine. The Web service engine receives the request and deserializes the message to Java objects, which are returned to you in a response.

When the Web service engine on the server receives the serialized request, a raw message in the form of an input stream, is passed to the parser, which understands Byte Order Mark (BOM). BOM is mandatory for UTF-16 encoding and it can be used in UTF-8. The message is deserialized to a Java objects and a service invocation is made. For two-way invocation, the engine needs to serialize the message using a specific encoding and send it back to the caller. The following example shows you how to use a property on the Stub to change the character set:

```
javax.xml.rpc.Stub stub=service.getPort("MyPortType");
((javax.xml.rpc.Stub)stub).setProperty(com.ibm.wsspi.webservices.Constants.MESSAGE_CHARACTER_SET_ENCODING,"UTF-16");
stub.invokeMethod();
```

In this code example, **com.ibm.wsspi.webservices.Constants.MESSAGE_CHARACTER_SET_ENCODING** = *"com.ibm.wsspi.webservices.xmlcharset";*

- Use a handler to change the character set through SOAP with Attachments API for Java (SAAJ).

  If you are using a handler, the SOAP message is transformed to a SAAJ format from other possible forms, such as an input stream. In such cases as a handleRequest method on the client side and a handleResponse method on the server side, the Web services engine transforms from a SAAJ format back to the stream with appropriate character encoding. This transformation or change is called a *roundtrip transformation*. The following is an example of how you can use a handler to specify the character encoding through SAAJ:

```
handleResponse(MessageContext mc) {
 SOAPMessageContext smc = (SOAPMessageContext) context;
 javax.xml.soap.SOAPMessage msg = smc.getMessage();
 msg.setProperty (javax.xml.soap.SOAPMessage.CHARACTER_SET_ENCODING, "UTF-16");
 }
}
```

Results

You have modified the character encoding from UTF-8 to UTF-16 in the Web service SOAP message.

Configuring the JAX-RPC Web services client deployment descriptor with an assembly tool

You can configure JAX-RPC Web services client deployment descriptor with an assembly tool.

Before you begin

You can configure deployment descriptors with assembly tools provided with WebSphere Application Server.

Also, you need an enterprise JavaBeans (EJB) Java archive (JAR) file, Web archive (WAR) file or an application client file that you can import into the assembly tool.

Assemble the client JAR file into an EAR file or assemble the client WAR file into an EAR file.

About this task

Complete this task if you are developing a managed client that runs in the Java EE client container. This task is done after you assemble the EJB or Web module.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.

3. Migrate the Web archive (WAR) or Java Archive (JAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to assembly tools. To migrate files, import your WAR or JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer documentation.

Results

You have a client deployment descriptor that is configured.

What to do next

Test the Web services client. This task explains how to test an unmanaged client Java archive (JAR) file and an unmanaged client application.

Configuring the JAX-RPC client deployment descriptor for handler classes

You can configure the JAX-RPC client deployment descriptor for user-provided handler classes.

Before you begin

You need an enterprise archive (EAR) file for the applications that you want to configure. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file should contain the handler classes to configure. These classes implement the `javax.xml.rpc.handler.Handler` interface. For more information on writing handler classes, see chapter 6 of the Web Services for Java Platform, Enterprise Edition (Java EE) specification and chapter 12 of the Java API for XML-based remote procedure call (JAX-RPC) specification. The application modules must contain the `webservices.xml`(for server) and the client deployment descriptors.

For a complete list of the supported standards and specifications, see the Web services specifications and API documentation.

About this task

Configure a handler in the client deployment descriptor by following the steps provided:

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the Web archive (WAR) or Java archive (JAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. Read about importing WAR or JAR files using an assembly tool in the Rational Application Developer documentation.
4. Configure the client deployment descriptor. Read about creating Web services handlers in the Rational Application Developer documentation.

Results

You have a client deployment descriptor that is configured.

What to do next

Test the Web services client. This task explains how to test an unmanaged client Java archive (JAR) file and an unmanaged client application.

Handler class properties with JAX-RPC

This article describes handler class properties using Java API for XML-based RPC (JAX-RPC).

You can configure the following handler class properties with assembly tools provided with WebSphere Application Server. See Configuring the `webservices.xml` deployment descriptor for Handler classes or Configuring the client deployment descriptors for Handler classes for instructions on how to configure the properties.

Description:

Standard Java Platform, Enterprise Edition (Java EE) technology descriptor field.

Display name:

Standard Java EE technology descriptor field.

Small icon:

Standard Java EE technology descriptor field.

Large icon:

Standard Java EE technology descriptor field.

Handler name:

The name of the handler. This name must be unique within the module.

Handler class:

The fully qualified name of the handler class. Initially, it is set by an assembly tool.

Initial parameters:

Property names and values available to the handler.

SOAP headers:

Qualified names (Qnames) of the SOAP headers that are processed by this handler.

See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) specification, available through Web services specifications and APIs , for more information about setting this property.

SOAP roles:

URIs containing the SOAP actor names for which the handler acts in the role.

See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) specification, available through Web services specifications and APIs , for more information about setting this property.

Example: Configuring handler classes for Web services deployment descriptors

This scenario explains how to add a client and server handler class to a sample application, `WebServicesSamples.ear`. The handler classes display messages when given a request or response to handle.

The code for the client handler class is illustrated in the following example:

```
 package samples;

public class ClientHandler implements javax.xml.rpc.handler.Handler {
     public ClientHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleRequest");
   return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleResponse");
   return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleFault");
   return true;  }

public void init(javax.xml.rpc.handler.HandlerInfo config) { }

public void destroy() {
  }

public javax.xml.namespace.QName[] getHeaders() {
   return null; }
}
```

The code for the server handler class is illustrated in the following example:

```
 package sample;
public class ServerHandler implements javax.xml.rpc.handler.Handler {
  public ServerHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleRequest");
   return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleResponse");
   return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleFault");
   return true; }

public void init(javax.xml.rpc.handler.HandlerInfo config) { }

public void destroy() {  }

public javax.xml.namespace.QName[] getHeaders() {
   return null;  }
}
```

1. Compile these classes using:
2. Open an assembly tool and import the two sample enterprise archive (EAR) files:

-
-

3. Import the compiled handler classes into the projects for the sample modules:
   - Import sample.ClientHandler into the **appClientModule** directory of the **AddressBookClient** project.
   - Import sample.ServerHandler into the **ejbModule** directory of the **AddressBookW2JE** project.
4. Configure the client deployment descriptor for handler classes.

   This topic explains how to configure the client deployment descriptor for user-provided handler classes.
5. Configure the webservices.xml deployment descriptor for handler classes.

   This topic explains how to configure the `webservices.xml` deployment descriptor for user-provided handler classes.
6. Save your changes and export the EAR files.
7. Uninstall the `WebServicesSamples.ear` application from your server if it is already installed.
8. Install the new `WebServicesSamples.ear` application.
9. Start the server.
10. Run the client:

    launchClient ApplicationClients.ear -CCjar=AddressBookClient.jar

    When the client runs, the console output looks like the following example. The messages from the handlers are shown in bold.

```
IBM WebSphere Application Server
J2EE Application Client Tool
Copyright IBM Corp., 1997-2003
WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client
Environment.
WSCL0035I: Initialization of the J2EE Application Client
Environment has completed.
WSCL0014I: Invoking the Application Client class
com.ibm.websphere.samples.webservices.addr.AddressBookClient
>> Querying address for 'Purdue Boilermaker' using port
AddressBookW2JE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       1 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       2 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       3 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port AddressBookW2JB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
```

```
>> Response is:
       4 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
```

For the client, the handler class is configured for each service reference, not for each port. The AddressBook sample has four ports, but only one service reference, therefore the ClientHandler handles requests and responses on all ports.

When the server log file is examined, it contains the following data:

```
[9/24/03 16:39:22:661 CDT] 4deec1c6 WebGroup      I SRVE0180I:
[HTTP router for AddressBookW2JE.jar] [/AddressBookW2JE] [Servlet.LOG]:
AddressBook: init
[9/24/03 16:39:23:161 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleRequest
[9/24/03 16:39:23:211 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleResponse
```

Results

The deployment descriptors for handler classes are configured. Deployment descriptors are required so that so that WebSphere Application Server can process the incoming Web services requests.

What to do next

Deploy the EAR file that has been configured and enabled for Web services. Then you can test the application to make sure it runs within the WebSphere Application Server environment.

Configuring the JAX-RPC Web services client bindings in the ibm-webservicesclient-bnd.xmi deployment descriptor

You can configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file with assembly tools.

Before you begin

You can configure deployment descriptors with assembly tools provided with the application server.

You must configure the assembly tool before you can use it. Read about configuring the assembly tool in the Rational Application Developer documentation.

About this task

Now that you have assembled the client module, complete this step to configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor. Deployment descriptors are required so that so that WebSphere Application Server can process the incoming Web services requests.

Configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file with the following steps provided:

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. Switch to the Java EE Perspective.
   a. Click **Window** > **Open Perspective** > **Other** > **Java EE**.
3. Open the Project Explorer.
   a. Click **Window** > **Show View** > **Other** > **Project Explorer**.
4. Locate the deployment descriptor file for the module. Hint: Deployment Descriptor: <*module*>
5. Double-click the deployment descriptor file to open the Deployment Descriptor editor.
   a. Select the **WS Binding** tab at the bottom of the editor window to open the Web Services Client Bindings editor.
6. Verify the serviceRefLink element settings.

a. Open the **Web Services Client Bindings** editor.
b. Click the **Services References** tab.
c. Click **Add**.
d. Select the service references defined in the client deployment descriptor file from the list.

7. Verify the deployedWSDLFile element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Select the service reference.
   c. Expand the **Service Reference Details** section.
   d. Click **Browse** that is located to the right of the Deployed WSDL file field.
   e. Select the new Web Services Description Language (WSDL) file.
   f. Click **OK**.

   You can also change the deployedWSDLFile element of a deployed Web service using the administrative console. Click **Enterprise Applications** > *application* > **Web module** or **EJB module** > *module* >**Web services client bindings**.
8. Verify the `defaultMappings` element settings.
   a. Open the **Web services client bindings** editor.
   b. Click **Default mappings**.
   c. Click **Add**.
   d. Edit the entries in the newly added row to establish a mapping between a *portType* and a *port* in the WSDL file. Only one entry is supported for each *portType*.
   e. Click **OK**.

   You can also change the defaultMappings element of a deployed Web service using the administrative console. Click **Enterprise Applications** > *application* > **Web module** or **EJB module** > *module* >**Web services client bindings**.
9. Access the Web services client Port bindings editor through the **Port qualified name binding details** section at the bottom of the editor pane.
10. Verify the syncTimeout element settings.
    a. Create a Port qualified name bindings for the port.
    b. Open the **Web services client bindings** editor.
    c. Confirm that a service reference is selected in either the **Component-scoped references** or the **Service references** section.
    d. Expand the **Port qualified name binding** section.
    e. Click **Add**. The **Add port qualified name binding** dialog opens.
    f. Type the *namespace* of the WSDL file port you want to configure, in the **Port namespace link** field.
    g. Type the *local_name* of the WSDL file port you want to configure in the **Port local name link** field. The name displayed in the **Port qualified name binding** list is the local name of the WSDL file port.
    h. Click **OK**.
    a. Configure the syncTimeout property by locating the Synchronization timeout field and enter the desired value. The default is 300 seconds.
11. Verify the basicAuth element settings.
    a. Locate the **HTTP basic authentication** field in the **Port qualified name binding details** section.
    b. Type the desired value in the User ID and Password fields.
    c. Click **OK**.
12. Verify the sslConfig element settings.
    a. Locate the **SSL configuration** field in the **Port qualified name binding details** section.

b. Type the desired value in the **Name** field.
c. Click **OK**.

13. After editing the properties, type **ctrl-s** on your keyboard to save the changes.

Results

You have configured the `ibm-webservicesclient-bnd.xmi` deployment descriptor. If you have configured all of the client deployment descriptors, test the Web services client. If you have not configured all of the client deployment descriptors, complete the configurations and then test the Web services client.

Related concepts

Assembly tools
WebSphere Application Server supports *assembly tools* that you can use to develop, assemble, and deploy Java Platform, Enterprise Edition (Java EE) modules.

ibm-webservicesclient-bnd.xmi assembly properties for JAX-RPC applications

The `ibm-webservicesclient-bnd.xmi` deployment descriptor file contains information for the Web services run time that is WebSphere product-specific. This deployment descriptor file is used with Java API for XML-based RPC (JAX-RPC) Web services. This binding file is not applicable for Java API for XML-Based Web Services (JAX-WS) Web services.

You can configure deployment descriptors with assembly tools provided with WebSphere Application Server. Read about configuring the JAX-RPC Web services client bindings in the ibm-webservicesclient-bnd.xmi deployment descriptor to learn more about configuring this deployment descriptor.

Assembly properties

The following user-definable assembly properties are supported:

- **componentNameLink**

  An attribute of the componentScopedRefs element. When a Web service is implemented by an Enterprise JavaBeans (EJB) implementation, each <componentScopedRefs> element contains assembly properties for an individual enterprise bean. The componentNameLink attribute of the <componentScopedRefs> element identifies the enterprise bean that the assembly properties apply to by specifying the <ejb-name>. This property is used only when the Web service client is an enterprise bean.
- **serviceRefLink**

  An attribute of the serviceRefs element. Specifies the link to the <service-ref-name> in the <service-ref> element in the client deployment descriptor. The client deployment descriptor is either `ejb-jar.xml`, `web.xml` or `application-client.xml`.
- **deployedWSDLFile**

  An attribute of the serviceRefs element is optional. Permits an alternate Web Services Description Language (WSDL) file to use other than that specified in the <wsdl-file> element of the <service-ref> element in the client deployment descriptor. If an attribute is specified, the alternate WSDL file must be packaged in the same module and must be compatible with the development WSDL file. The deployedWSDLFile property supplies a new WSDL file containing a different endpoint Web address than the original WSDL file.
- **defaultMappings** element

  Identifies which port to use for a given portType when one is not selected by the client. This element has the following attributes: portTypeNamespace, portTypeLocalName, portNamespace, portLocalName. These attributes identify which wsdl:port is used for a wsdl:portType.
- **syncTimeout**

  An attribute of the portQnameBindings element. Specifies how long, in seconds, to wait for a response from a synchronous call. The default is 300 seconds.
- **basicAuth**

An element of the portQnameBindings element. Authenticates a service client to the service endpoint, independent of the underlying transport that includes, HTTP, HTTPS, and Java Message Service (JMS). Set the user ID and password attributes as needed.

- **sslConfig**

  An element of the portQnameBindings element. Specifies the Secure Sockets Layer (SSL) configuration of an HTTPS outbound request. The name attribute is the name of an SSL configuration entry or alias that is defined in the SSL configuration repertoire. This attribute is used only when the client is running in the WebSphere Application Server.

  For WebSphere Application Server for z/OS, some digital certificate and keyring management is required. Refer to Creating Secure Sockets Layer digital certificates and System Authorization Facility keyrings applications to initiate HTTPS requests for more information.

A bindings file example

The following example demonstrates the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute whose value is a unique string. Review the template `xmi` files generated by the **WSDL2Java** command for examples of ID strings. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

```
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">

 <componentScopedRefs componentNameLink="myComponent ref"/>

 <serviceRefs serviceRefLink="myService ref" deployedWSDLFile="META-INF/wsdl/alternate.wsdl">
  <defaultMappings portTypeLocalName="AddressBook" portTypeNamespace="http://www.com.ibm"
portLocalName="AddressBookPort" portNamespace="http://www.com.ibm"/>
   <portQnameBindings portQnameNamespaceLink="http://www.com.ibm"
portQnameLocalNameLink="AddressBookPort" syncTimeout="99">
    <basicAuth userid="myId" password="myPassword"/>
    <sslConfig name="mynode/DefaultSSLSettings"/>
   </portQnameBindings>
  </serviceRefs>
 </com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Related tasks

Viewing Web services deployment descriptors in the administrative console
You can view the Web services client and server deployment descriptors for a deployed Web services application. You can view the bindings in the deployment descriptors.

“Configuring the JAX-RPC Web services client bindings in the ibm-webservicesclient-bnd.xmi deployment descriptor” on page 618
You can configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file with assembly tools.

“Assembling Web services applications” on page 727
You can assemble Java-based Web services applications using assembly tools.

Related reference

“WSDL2Java command for JAX-RPC applications” on page 591
Run the WSDL2Java command-line tool against the WSDL file to create Java APIs and deployment descriptor templates.

Implementing extensions to JAX-RPC Web services clients

WebSphere Application Server provides extensions to Web services clients using the Java API for XML-based RPC (JAX-RPC) programming model.

About this task

You can customize Web services by using the following extensions to the JAX-RPC client programming model.

- Set the **REQUEST_SOAP_HEADERS** and **RESPONSE_SOAP_HEADERS** properties in a JAX-RPC client Stub to enable a Web services client to send or retrieve implicit SOAP headers.

  An implicit SOAP header is a SOAP header that is not explicitly defined in the WSDL file. An implicit SOAP header file fits one of the following descriptions:

  – A message part that is declared as a SOAP header in the binding in the WSDL file, but the message definition is not referenced by a portType within a WSDL file.

  – An element that is not contained in the WSDL file.

  Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

  See “Sending implicit SOAP headers with JAX-RPC” on page 632 or “Receiving implicit SOAP headers with JAX-RPC” on page 633 to learn how to modify your client code to send or retrieve transport headers.

- Set the **REQUEST_TRANSPORT_PROPERTIES** and **RESPONSE_TRANSPORT_PROPERTIES** properties to enable a Web services client to send or retrieve transport headers.

  Set the properties on the Stub or Call object.

  By modifying your client code to send or retrieve transport headers, you can send or receive specific information within the transport headers of outgoing requests or incoming responses from the server. For requests or responses that use the HTTP transport, the information is sent or retrieved in an HTTP header. Similarly, for a request or response that uses the Java Message Service (JMS) transport, the information is sent or retrieved in a JMS message property.

  See “Sending transport headers with JAX-RPC” on page 634 or “Retrieving transport headers with JAX-RPC” on page 635 to learn how to modify your client code to send or retrieve transport headers. See “Transport header properties best practices” on page 551 to learn how to enable a Web services client to send or retrieve transport headers.

- Implement support for **javax.xml.rpc.ServiceFactory.loadService()** methods.

  The loadService methods create an instance of the generated service implementation class in an implementation-specific manner. The loadService methods are new for JAX-RPC 1.1 and include three signatures:

  – **public.javax.xml.rpc.Service loadService (Class serviceInterface)**

    As documented in the JAX-RPC specification, this method returns the generated service implementation for the service interface. See the Web services specifications and API documentation to review the JAX-RPC specification.

  – **public.javax.xml.rpc.Service loadService (URL wsdlDocumentLocation, Class serviceInterface, Properties properties)**

    This method behaves like the loadService (Class serviceInterface) because the following parameters are ignored:

    - wsdlDocumentLocation
    - properties

  – **public.javax.xml.rpc.Service loadService (URL wsdlDocumentLocation, QName serviceName, Properties properties)**

    This method returns the generated service implementation for the specified service by using optional namespace-to-package mapping information.

    - wsdlDocumentLocation - ignored
    - serviceName - QName (namespace, localpart) of the service

- properties - If this parameter is non-null, it contains namespace-to-package mapping entries. Each Property entry key is a String corresponding to the namespace. Each Property entry value is a String corresponding to the Java package name.

  If the properties argument contains an entry with a key (namespace) that matches the namespace portion of the QName serviceName argument, the entry value (`javaPackage`) is used as the package name when trying to locate the service implementation.

For more information on these methods, see the JAX-RPC specification.

- Implement the **CustomBinder interface** to provide concrete custom data binders for a specific XML schema type **(JAX-RPC applications only)**.

  Custom data binders are used to map XML schema types with Java objects. Custom data binders provide bindings for XML schema types that are not supported by the current Java API for XML-based Remote Call Procedure (JAX-RPC) specification. WebSphere Application Server provides an extension to the Web Services for Java Platform, Enterprise Edition (Java EE) programming model called the CustomBinder interface that implements these custom bindings for a specific XML schema type. The CustomBinder interface has three properties, in addition to deserialize and serialize methods:

  – QName for the XML schema type
  – QName scope
  – Java type

  The custom data binder defines serialize and deserialize methods to convert between a Java object and a SOAPElement interface. A custom data binder is added to the runtime system and interacts with the Web services runtime using a SOAPElement. They are added to the runtime by using custom binding providers. Read about the custom data binders and the custom binding provider to learn more. See the CustomBinder interface documentation to learn more about how you can implement this interface to provide concrete custom data binders for a specific XML schema type.

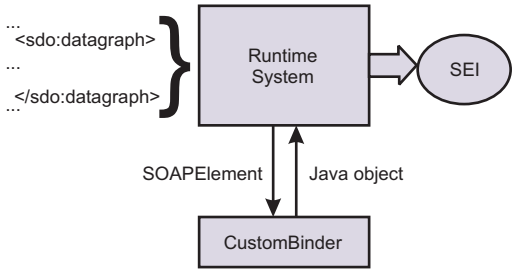
Custom data binders for JAX-RPC applications

A *custom data binder* is used to map XML schema types with Java objects. Custom data binders provide bindings for XML schema types that are not supported by the current Java API for XML-based Remote Call Procedure (JAX-RPC) specification.

The custom data binder defines serialize and deserialize methods to convert between a Java object and a SOAPElement interface. A custom data binder is added to the run time system and interacts with the Web services runtime using a SOAPElement. Unlike conventional deserializers, custom data binders do not rely on the low-level parsing events from the run time to build the Java object, such as Simple API for XML (SAX). Instead, the run time builds the custom data binder by rendering the incoming SOAP message into a SOAPElement. The SOAPElement that contains the message is passed to the customer data binder. For example, if the incoming message contains a Service Data Object (SDO) datagraph, the run time system processes as follows:

1. The run time system recognizes the `<sdo:Datagraph>` code.
2. The run time queries the type mapping system to locate the custom data binder for the datagraph data, for example SDOCustomBinder.
3. A SOAPElement is created that represents the incoming SDO datagraph.
4. The run time passes the SOAPElement to the SDOCustomBinder.

Within the deserialized method, the SDOCustomBinder extracts the content from the SOAPElement and builds a concrete DataGraph object with a `commonj.sdo.DataGraph` type. The figure displays the Web services runtime flow and a custom data binder.



When a Java object is serialized, a similar process occurs. The run time locates a custom data binder and converts the Java object to a SOAPElement. The runtime serializes the SOAPElement to the raw message that is transported in the output stream.

The following is an example of an XML schema that is defined by the SDO specification:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sdo="commonj.sdo"
  targetNamespace="commonj.sdo">

  <xsd:element name="datagraph" type="sdo:DataGraphType"/>

  <xsd:complexType name="DataGraphType">
  <xsd:complexContent>
    <xsd:extension base="sdo:BaseDataGraphType">
      <xsd:sequence>
        <xsd:any minOccurs="0" maxOccurs="1"
                 namespace="##other" processContents="lax"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="BaseDataGraphType" abstract="true">
  <xsd:sequence>
    <xsd:element name="models" type="sdo:ModelsType" minOccurs="0"/>
    <xsd:element name="xsd" type="sdo:XSDType" minOccurs="0"/>
    <xsd:element name="changeSummary"
                 type="sdo:ChangeSummaryType" minOccurs="0"/>
   </xsd:sequence>
  <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

  <xsd:complexType name="ModelsType">
  <xsd:sequence>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
             namespace="##other" processContents="lax"/>
  </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="XSDType">
  <xsd:sequence>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
        namespace="http://www.w3.org/2001/XMLSchema" processContents="lax"/>
  </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ChangeSummaryType">
  <xsd:sequence>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
             namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:attribute name="create" type="xsd:string"/>
```

```
  <xsd:attribute name="delete" type="xsd:string"/>
  </xsd:complexType>

</xsd:schema>
```

WebSphere Application Server defines the CustomBinder interface that implements concrete custom bindings for a specific XML schema type.

The custom binding provider is used to import the custom bindings into the run time. To learn how to plug your custom data binders into the WSDL2Java command-line tool for development, read about custom binding providers. You can also read about usage patterns for deploying custom data binders to learn more about how to deploy the provider package to your runtime, as well as the roles involved in the custom binding process.

Custom binding providers for JAX-RPC applications

A *custom binding provider* is the packaging of custom data binder classes for with a declarative metadata file. The main purpose of a custom binding provider is to aggregate related custom data binders to support particular user scenarios. The custom binding provider is used to plug the custom data binders into the emitter tools and the run time system so that the emitter tools can generate the appropriate artifacts and the run time system can augment its existing type mapping system to reflect the applied custom data binders and invoke them.

A custom binding provider works with a specific XML schema type, while applications involve a few related XML schema types. You need a mechanism to aggregate and declare various custom data binders to provide a complete binding solution. The concept of the custom binding provider defines a declarative model that can be used to plug in a set of custom data binders to either emitter tools or the run time system.

You can review information in Custom data binders to learn more about custom data binders and CustomBinder interface, which is the API included in WebSphere Application Server to define the custom data binders. After you have reviewed these articles you are ready to deploy the custom binder package. To learn how to deploy this package, see Usage patterns for deploying custom data binders.

The declarative metadata file, `CustomBindingProvider.xml`, is an XML file that is packaged with the custom provider classes in a single Java archive (JAR) file and located in the `/META-INF/services/directory`. After a provider JAR file is packaged, the binary information and the metadata file located in the JAR file can be used by the WSDL2Java command-line tool and the run time system.

The following example is the XML schema for the `CustomBindingProvider.xml` file. The top level type is the providerType that contains a list of mapping elements. Each mapping element defines the associated custom data binder and properties, including xmlQName, javaName and qnameScope. You can read more about these properties in CustomBinder interface. The providerType also has an attribute called scope that has a value of *server*, *application* or *module*. The scope attribute is used by the server deployment to resolve the conflict and to realize a custom binding hierarchy.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    targetNamespace=
       "http://www.ibm.com/webservices/customdatabinding/2004/06"
    xmlns:customdatabinding=
        "http://www.ibm.com/webservices/customdatabinding/2004/06"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="qualified">

   <xsd:element name="provider" type="customdatabinding:providerType"/>

   <xsd:complexType name="providerType">
   <xsd:sequence>
     <xsd:element name="description" type="xsd:string" minOccurs="0"/>
     <xsd:element name="mapping" minOccurs="0" maxOccurs="unbounded">
```

```
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="description" type="xsd:string" minOccurs="0"/>
          <xsd:element name="xmlQName" type="xsd:QName"/>
          <xsd:element name="javaName" type="xsd:string"/>
          <xsd:element name="qnameScope"
                       type="customdatabinding:qnameScopeType"/>
          <xsd:element name="binder" type="xsd:string"/>
        </xsd:sequence>
       /xsd:complexType>
      </xsd:element>
       <xsd:attribute name="scope"
             type="customdatabinding:ProviderScopeType" default="module"/>
       </xsd:sequence>
   </xsd:complexType

  <xsd:simpleType name="qnameScopeType">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="simpleType"/>
         <xsd:enumeration value="complexType"/>
         <xsd:enumeration value="element"/>
      </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ProviderScopeType">
     <xsd:restriction base="xsd:string">
         <xsd:enumeration value="server"/>
         <xsd:enumeration value="application"/>
         <xsd:enumeration value="module"/>
     </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

The following is an example of the `CustomBindingProvider.xml` file for the SDO DataGraph schema that was introduced in CustomBinder interface. The example displays the mapping between a schema type, DataGraphType, and a Java type, commonj.sdo.DataGraph. The binder that represents this mapping is called `test.sdo.SDODataGraphBinder`.

```
<cdb:provider
 xmlns:cdb="http://www.ibm.com/webservices/customdatabinding/2004/06"
   xmlns:sdo="commonj.sdo">
   <cdb:mapping>
     <cdb:xmlQName>sdo:DataGraphType</cdb:xmlQName>
     <cdb:javaName>commonj.sdo.DataGraph</cdb:javaName>
     <cdb:qnameScope>complexType</cdb:qnameScope>
     <cdb:binder>test.sdo.SDODataGraphBinder</cdb:binder>
   </cdb:mapping>
</cdb:provider>
```

You need to import your custom data binders into the WSDL2Java command-line tool for development purposes. The custom data binders affect how the development artifacts, including the Service Endpoint Interface and the JSR 109 mapping data, are generated from the Web Services Description Language (WSDL) file. The WSDL2Java command-line tool ships with WebSphere Application Server and uses the custom binder Java archive file, or custom binder package, to generate these the development artifacts.

The following example is a WSDL file that references the SDO DataGraph schema that is introduced in the CustomBinder interface topic.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://sdo.test"
   xmlns:impl="http://sdo.test"
   xmlns:intf="http://sdo.test"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sdo="commonj.sdo">
```

```
<wsdl:types>
 <schema elementFormDefault="qualified" targetNamespace="http://sdo.test"
      xmlns="http://www.w3.org/2001/XMLSchema" xmlns:sdo="commonj.sdo">
  <import namespace="commonj.sdo" schemaLocation="sdo.xsd"/>
 </schema>
</wsdl:types>

<wsdl:message name="echoResponse">
 <wsdl:part element="sdo:datagraph" name="return"/>
</wsdl:message>

<wsdl:message name="echoRequest">
  <wsdl:part element="sdo:datagraph" name="parameter"/>
</wsdl:message>

<wsdl:portType name="EchoService">
  <wsdl:operation name="echo">
    <wsdl:input message="impl:echoRequest" name="echoRequest"/>
    <wsdl:output message="impl:echoResponse" name="echoResponse"/>
   </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="EchoServiceSoapBinding" type="impl:EchoService">
  <wsdlsoap:binding style="document"
                   transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="echo">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:input name="echoRequest">
        <wsdlsoap:body use="literal"/>
       </wsdl:input>

      <wsdl:output name="echoResponse">
         <wsdlsoap:body use="literal"/>
      </wsdl:output>
   </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="EchoServiceService">
    <wsdl:port binding="impl:EchoServiceSoapBinding" name="EchoService">
      <wsdlsoap:address location="http://<uri>"/>
   </wsdl:port>
 </wsdl:service>

</wsdl:definitions>
```

If you run the WSDL2Java command without the custom data binding package, the following Service Endpoint Interface is generated with a parameter type, as dictated by the JAX-RPC specification:

```
public interface EchoService extends java.rmi.Remote {
  public javax.xml.soap.SOAPElement
   echo(javax.xml.soap.SOAPElement parameter)
    throws java.rmi.RemoteException;
}
```

When you run the WSDL2Java command with the custom data binding package, the custom data binders are used to generate the parameter types. To apply the custom data binders, use the -classpath option on the WSDL2Java tool. The tool searches its classpath to locate all the files with the same file path of `/META-INF/services/CustomBindingProvider.xml`. The following is an example how you can use the command to generate a Service Endpoint Interface with the parameter type of `commonj.sdo.Datagraph`:

```
WSDL2Java -role develop-server -container web classpath sdobinder.jar echo.wsdl
```

The Service Endpoint Interface that is generated looks like the following:

```
public interface EchoService extends java.rmi.Remote {
  public commonj.sdo.DataGraph
   echo(commonj.sdo.DataGraph parameter)
    throws java.rmi.RemoteException;
}
```

The custom binder packaged JAR file has to be made available at runtime to make sure the Web service client is invoked, regardless if it is a stub-based client or a Dynamic Invocation Interface (DII) client. The same applies to the service.

CustomBinder interface for JAX-RPC applications

WebSphere Application Server defines a CustomBinder interface that you can implement for Java API for XML-based Remote Call Procedure (JAX-RPC) applications to provide concrete custom data binders for a specific XML schema type.

The CustomBinder interface has three properties, in addition to deserialize and serialize methods. These properties are QName for the XML schema type, the QName scope, and the Java type that the schema type maps to. The properties are accessible through the corresponding getter methods.

getQName

The getQName method returns the QName of the target XML schema type. Custom data binders only work with the root level schema type.

For anonymous types, the getQName method returns the QName of the containing element.

For named types, the getQName method returns the QName of the complexType or the simpleType.

getQNameScope

The getQNameScope method returns the binder qnameScope property that indicates whether the schema type is a named type or an anonymous type. The qnameScope property value can be *complexType* for an <xsd:complexType>, *simpleType* for an <xsd:simpleType> or *element* for an <xsd:element> that is defined with an anonymous type.

In the following schema, data1 is an element that is defined with an anonymous type. The element, data2, is defined using the named type, data2Type.

```
<xsd:element name="data1">
 <xsd:complexType>
  ...
 </xsd:complexType>
</xsd:element>

<xsd:element name= "data2" type="data2Type"/>
<xsd:complexType name="data2Type">
 ...
</xsd:complexType>
```

The anonymous type, data1, has a qNameScope of element and a qName of data1. The type, data2Type, has a qNameScope of complexType and a qName of data2Type.

The element, data2, is not represented in the custom data binder. The custom data binder only processes types and not elements.

getJavaName

The getJavaName method returns the fully-qualified class name for the Java type that is mapped to the named or anonymous type. The class can be an interface or a concrete class. The object returned from

the deserialize method has a type that is compatible with the Java type that is returned by the getJavaName method.

serialize

The serialize method returns the SOAPElement that the custom data binder builds from the Java object. The Java object is passed from the run time system and is expected to match what is returned from the getJavaName method. The SOAPElement parameter does not have child elements, but it does have a valid QName. This parameter is a reference for the binder to create the final SOAPElement.

In most cases, the binder implementation appends the child elements to the root SOAPElement. The run time system guarantees that the SOAPElement QName is correct. Therefore, the custom data binder for named types keeps the QName of the root element because the binder does not know the enclosing element. The binder implementation for an anonymous type should always include the QName in the returned SOAPElement that matches the defined schema type. WebSphere Application Server does not have concrete methods in the CustomBindingContext parameter.

deserialize

The deserialize method returns a Java object that the custom data binder builds from the passed root SOAPElement. The object type of the returned Java object must match what is returned from the getJavaName method. Unlike the parameter serialize method, the passed SOAPElement contains the original XML data with the necessary namespace declarations.

The following is an example of an implementation of the SDO DataGraph binder, where the convertToSDO and convertToSAAJ utility methods convert between SOAPElement and an SDO object.

```
package test.sdo.binder;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;

import com.ibm.wsspi.webservices.binding.CustomBinder;
import com.ibm.wsspi.webservices.binding.CustomBindingContext;

public class DataGraphBinder implements CustomBinder {
 public QName getQName() {
  return new QName("commonj.sdo", "DataGraphyType");
}
public String getJavaName() {
 return CustomBinder.QNAME_SCOPE_COMPLEXTYPE;
}
public String getJavaName() {
 return commonj.sdo.DataGraph.class.getName();
}
public javax.xml.soap.SOAPElement serialize(
  Object bean,
  SOAPElement rootNode,
  CustomBindingContext context)
  throws javax,xml.soap.SOAPException {
 // convertToSAAJ is a utility method to convert
 // the SDO DataGraph to the SOAPElement
 return convertToSAAJ(bean, rootNode);

public Object deserialize(
  SOAPElement source,
  CustomBindingContext context)
  throws javax.xml.soap.SOAPException {
  // convertToSDO is a utility method to convert
  // the SOAPElement to the SDO DataGraph
  return convertToSDO(source);
 }
}
```

To learn more about custom data binders, see “Custom data binders for JAX-RPC applications” on page 623. To learn how to plug your custom data binders into the WSDL2Java command-line tool for development, see Custom binder providers.

Usage patterns for deploying custom data binders for JAX-RPC applications

Custom data binders are used to map XML schema types with Java objects. Custom data binders provide bindings for XML schema types that are not supported by the current Java API for XML-based Remote Call Procedure (JAX-RPC) specification. WebSphere Application Server provides an extension to the Web Services for Java Platform, Enterprise Edition (Java EE) programming model called the CustomBinder interface that implements these custom bindings for a specific XML schema type. The custom binding provider is the package for the custom data binders that is imported into the runtime.

You can learn about the CustomBinder API in the topic CustomBinder interface. The topic Custom data binders includes general information about custom binders and the topic Custom binding providers reviews how they are packaged for development.

This usage pattern reviews how to deploy the provider package to your runtime, as well as the roles involved in the custom binding process.

Roles involved in custom data binding

Four roles are involved with custom data binding. These roles that are defined by the Web Services for Java Platform, Enterprise Edition (Java EE) specification are as follows:

- **Custom binding provider** is responsible for implementing the required custom data binders, declaring these binders in a `CustomBindingProvider.xml` file and packaging the binding classes into a Java archive (JAR) file.
- **Application developer** is responsible for applying the custom binding provider JAR file and generating the development artifacts.
- **Application assembler** needs to understand the application requirements in terms of the custom data binding and decides how to package the custom provider JAR file as a part of the application.
- **Application deployer** configures the shared libraries to make custom data binding support available to the applications. This needs to be done if the custom provider JAR file is not packaged with the application. If the application is not deployed, the deployer has to run the Web services deployment tools after the application is installed.

Common usage patterns

The custom binder provider package can be deployed in various ways to provide flexibility beyond the standard JAX-RPC mapping standards. Three primary deployment usage patterns are as follows:

- **Deploy the custom data binders at the server level**

  This pattern ensures that all the applications that are running on the server are affected by the custom data binders and is useful if fundamental XML types are introduced but are not supported by the standard JAX-RPC mapping rules.

  This type of situation occurs frequently for new Web services specifications that define new schema types. For example, the WS-Addressing specification defines an EndpointReferenceType schema type that is not supported by the JAX-RPC mapping rules. Because this pattern requires augmenting the server classpath, it has a significant impact on the server runtime and affects the installed applications. This pattern is most suitable for WebSphere Application Server internal components.
- **Deploy the custom binders for one or more application**

  Use this pattern if you only want specified applications to be affected by the custom data binders and if relevant XML schema types apply to a set of applications. You can share the custom data binders within a set of applications while achieving isolation between different sets of applications.
- **Deploy the custom binders for a specific Web module within an application**

Using this pattern ensures that a specific Web module is affected by the deployed custom data binders. This pattern is useful when fine granularity for custom binding is required. You cannot use this pattern with EJB modules because the module and its referenced library belong to the entire application.

Usage patterns

This section reviews deploying custom data binders using one of the three patterns:

- **Server level deployment**

  If you deploy the custom data binders at the server level, you need to set the scope attribute of the declared binding provider as *server*. Setting the value to *server* guarantees a higher priority for declared binders if there are conflicts between the server and applications. The custom binding provider JAR file needs to be in the appropriate place to be picked up by the server runtime. Configure the server path and make the custom binding provider JAR file a part of the server classpath. To learn about values used in configuring the server classpath see Java virtual machine settings.

- **Deploying custom data binders for one or more applications**

  To deploy custom data binders for one or more applications, set the scope attribute of the declared custom binding provider as *application*. Setting the value to application guarantees higher priority binders in case of conflicts between the application and the module. If the custom data binders are used by more than one application, configure a shared library for the applications to reference. To learn about values used in configuring the shared libraries path see Managing shared libraries.

- **Deploy the custom data binders for a specific Web module within an application**

  To deploy custom data binders for a specific Web module within an application, set the scope attribute of the declared custom binding provider to the value *module*. The only way to apply the custom data binder for this pattern is to pre-package the custom binding provider JAR file with the Web module, for example, place the JAR file in the `/WEB-INF/lib` directory.

Example: Using JAX-RPC properties to send and receive SOAP headers

WebSphere Application Server provides extensions to the Java API for XML-based RPC (JAX-RPC) and Web Services for Java Platform, Enterprise Edition (Java EE) client programming models, including the REQUEST_SOAP_HEADERS and RESPONSE_SOAP_HEADERS Stub properties. This is an example of how these two properties are used.

The following programming example illustrates how to send two request SOAP headers and receive one response SOAP header within a Web services request and response:

```
1 //Create the request and response hashmaps.
2 HashMap requestHeaders=new HashMap();
3 HashMap responseHeaders=new HashMap();
4
5 //Add "AtmUuid1" and "AtmUuid2" to the request hashmap.
6 requestHeaders.put(new QName("com.rotbank.security", "AtmUuid1"),
7   "<AtmUuid1 xmlns=\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid1>");
8 requestHeaders.put(new QName("com.rotbank.security", "AtmUuid2"),
9  ((IBMSOAPFactory)SOAPFactory.newInstance()).createElementFromXMLString(
10   "x:AtmUuid2 xmlns:x=\"com.rotbank.security\"><x:uuid>ROTB-0A01254385FCA09
     </x:uuid><x:AtmUuid2>"));
11
12 //Add "ServerUuid" to the response hashmap.
13 //If "responseHeaders" is empty, all the SOAP headers are
14 //extracted from the response message.
15 responseHeaders.put(new QName("com.rotbank.security","ServerUuid"), null);
16
17 //Set the properties on the Stub object.
18 stub.setProperty(Constants.REQUEST_SOAP_HEADERS.requestHeaders);
19 stub.setProperty(Constants.RESPONSE_SOAP_HEADERS.responseHeaders);
20
21 //Call the operationon the Stub.
22 stub.foo(parm2, parm2);
23
```

```
24 //Retrieve "ServerUuid" from the response hashmap.
25 SOAPElement serverUuid =
26   (SOAPElement) responseHeaders.get(new QName("com.rotbank.security","ServerUuid"));
27
28 //Note: "serverUuid" now equals a SOAPElement object that represents the
29 //following code:
30//"<y:ServerUuid xmlns:y=\"com.rotbank.security\"><:uuid>ROTB-0A03519322FSA01
   </y:uuid></y:ServerUuid.");
```

On lines 2-3, new HashMaps are created that are used for the request and response SOAP headers.

On lines 6-10, the AtmUuid1 and AtmUuid2 headers elements are added to the request HashMap.

On line 15, the ServerUuid header element name, along with a null value, is added to the response HashMap.

On line 18, the request HashMap is set as a property on the Stub object. This causes the AtmUuid1 and AtmUuid2 headers to be added to each request message that is associated with an operation that is invoked on the Stub object.

On line 19, the response HashMap is set as a property on the Stub object. This causes the ServerUuid header to be extracted from each response message that is associated with an operation that is invoked on the Stub object.

On line 22, the Web service operation is invoked on the Stub object.

On lines 25-26, the ServerUuid header is retrieved from the response HashMap. The header was extracted from the response message and inserted into the HashMap by the Web services engine.

Sending implicit SOAP headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to send values in implicit SOAP headers. By modifying your client code to send implicit SOAP headers, you can send specific information within an outgoing Web service request.

Before you begin

To complete this task, you need a Web services client that you can enable to send implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

You cannot manipulate protected SOAP headers. A SOAP header that is declared protected by its owning component, for example, Web Services Security, is not accessible to client applications. An exception occurs if you try to manipulate protected SOAP headers.

About this task

The client application sets properties on the Stub or Call object to send and receive implicit SOAP headers.

1. Create a java.util.HashMap object.

2. Add an entry to the HashMap object for each implicit SOAP header that the client wants to send. The HashMap entry key is the QName of the SOAP header. The HashMap entry value is either an SAAJ SOAPElement object or a String that contains the XML text of the entire SOAP header element.
3. Set the HashMap object as a property on the Stub or Call object. The property name is com.ibm.websphere.webservices.Constants.REQUEST_SOAP_HEADERS. The value of the property is the HashMap.
4. Issue the remote method calls using the Stub or Call object. The headers within the HashMap object are sent in the outgoing message.

   A `JAXRPCException` error can occur if any of the following are true:

   - The HashMap object contains a key that is not a QName object or if the HashMap object contains a value that is not a String or a SOAPElement object.
   - The HashMap object contains a key that represents a SOAP header that is declared protected by the owning component.

Results

You have a JAX-RPC Web services client that is configured to send implicit SOAP headers.

Receiving implicit SOAP headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to receive values from implicit SOAP headers. By modifying your client code to receive implicit SOAP headers, you can receive specific information within an incoming Web service response.

Before you begin

To complete this task, you need a Web services client that you can enable to receive implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

You cannot manipulate protected SOAP headers. A SOAP header that is declared protected by its owning component, for example, Web Services Security, is not accessible to client applications. An exception occurs if you try to manipulate protected SOAP headers.

About this task

The client application sets properties on the Stub or Call object to send and receive implicit SOAP headers.

1. Create a java.util.HashMap object
2. Add an entry to the HashMap object for each implicit SOAP header that the client wants to receive. The HashMap entry key is the QName of the SOAP header. The HashMap entry value is null.
3. Set the HashMap entry on the Stub or Call object. The property name is com.ibm.websphere.webservices.Constants.RESPONSE_SOAP_HEADERS. The value of the property is the HashMap.

4. Issue remote method calls against the Stub or Call object. The Web services engine extracts the specified response headers from the Web services response message and inserts them into the HashMap. After the remote method returns, the response headers are accessible from the HashMap object.

   A JAXRPCException error can occur if any of the following are true:

   - The HashMap contains a key that is not a QName.
   - The HashMap contains a key that represents a SOAP header that is declared protected by the owning component.

Results

You have a JAX-RPC Web services client that can receive values from implicit SOAP headers.

Sending transport headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to send application-defined information along with your Web services requests by using transport headers.

Before you begin

You need a JAX-RPC Web services client that you can enable to send transport headers.

Sending transport headers is supported only by Web services clients, and only supported for the HTTP and JMS transports. The Web services client must call the JAX-RPC APIs directly and not through any intermediary layers, such as a gateway function. Sending and retrieving transport headers on the Web services server is done through non-Web services APIs.

About this task

When using the JAX-RPC programming model, the client must set a property on the Stub or Call object to send values in transport headers. After you set the property, the values are set in all the requests for subsequent remote method invocations against that Stub or Call object until the associated property is set to null or the Stub or Call object is discarded.

To send values in the transport headers on outbound requests, modify the client code as follows:

1. Create a java.util.HashMap object that contains the transport header identifiers.
2. Add an entry to the HashMap object for each transport header that you want the client to send.
   a. Set the HashMap entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made as to the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class.
   b. Set the HashMap entry value to a string that contains the value of the transport header.
3. Set the HashMap entry on the Stub or Call object using the com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES property. When the REQUEST_TRANSPORT_PROPERTIES property value is set, that HashMap is used on subsequent invocations to set the header values in the outgoing requests. If the REQUEST_TRANSPORT_PROPERTIES property value is set to null, no HashMap is used on subsequent invocations to set header values in outgoing requests. To learn more about these properties, see the transport header properties documentation.
4. Issue remote method calls against the Stub or Call object. The headers and the associated values from the HashMap are added to the outgoing request for each method invocation. If the invocation

uses HTTP, then the transport headers are sent as HTTP headers within the HTTP request. If the invocation uses JMS, then the transport headers are sent as JMS message properties.

If the property is not set correctly, you might experience API usage errors that result in a JAXRPCException error. The following requirements must be met, or the process fails:

- The property value that is set on the Stub or Call object must be a HashMap object or null.
- The HashMap must not be empty.
- Each key in the HashMap must be a String object.
- Each value in the HashMap must be a String object.

Results

You have a JAX-RPC Web services client that is configured to send transport headers.

Retrieving transport headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to retrieve values from transport headers. For a request that uses HTTP, the transport headers are retrieved from HTTP headers found in the HTTP response message. For a request that uses Java Message Service (JMS), the transport headers are retrieved from the JMS message properties found on the JMS response message.

Before you begin

You need a Web services client that you can enable to retrieve transport headers.

Retrieving transport headers is supported only by Web services clients, and only supported for the HTTP and JMS transports. The Web services client must call the JAX-RPC APIs directly and not through any intermediary layers, such as a gateway function. Sending and retrieving transport headers on the Web services server is done through non-Web services APIs.

About this task

When using the JAX-RPC programming model, the client must set a property on the Stub or Call object in order to retrieve values from the transport headers. After you set the property, values are read from responses for the subsequent method invocations against that Stub or Call instance until the associated property is set to null or the Stub or Call object is discarded.

To retrieve values from the transport headers on inbound responses, modify the client code.

1. Create a java.util.HashMap object that contains the names of the transport headers to be retrieved from incoming response messages.
2. Add an entry to the HashMap for each header that you want to retrieve a value from every incoming response message.
   a. Set the HashMap entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user-defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made to confirm the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class. The HashMap entry value is ignored and does not need to be set. An empty HashMap, for example, one that is non-null, but does not contain any keys, causes all the transport headers in the response to be retrieved.
3. Set the HashMap entry on the Stub or Call object using the com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES property. When the HashMap is set, the RESPONSE_TRANSPORT_PROPERTIES property is used in subsequent

invocations to retrieve the headers from the responses. If you set the property to null, no headers are retrieved from the response. To learn more about these properties, see the transport header properties documentation.

4. Issue remote method calls against the Stub or Call object. The values from the specified transport headers are retrieved from the response message and placed in the HashMap.

   If the property is not set correctly, you might experience API usage errors that result in a JAXRPCException error. The following requirements must be met, or the process fails:

   - The property value that is set on the Stub or Call object must be either null or an instance of a HashMap.
   - All the HashMap keys must be a string data type, and the keys must not be null.

Results

You have a JAX-RPC Web service that can receive transport headers from incoming response messages.

Running an unmanaged Web services JAX-RPC client

WebSphere Application Server Version 7.0 and the Application Client for WebSphere Application Server Version 7.0 provides an unmanaged client implementation that is based on the Java API for XML-based RPC (JAX-RPC) 1.1 specification. The Thin Client for JAX-RPC with WebSphere Application Server is an unmanaged and stand-alone Java client environment that enables you to run JAX-RPC Web services client applications to invoke Web services that are hosted by the application server.

Before you begin

Before you can set up a JAX-RPC unmanaged client environment you will need to obtain the Thin Client for JAX-RPC Java archive (JAR) file. To obtain the Thin Client for JAX-RPC, you must either install the application server or the application client.

The Thin Client for JAX-RPC JAR file, `com.ibm.ws.webservices.thinclient_7.0.0.jar`, is located in the *app_server_root*`\runtimes` directory. Refer to the license agreements to ensure correct usage and for limitations on copies of the Thin Client for JAX-RPC outside of the WebSphere environment.

The Thin Client for JAX-RPC runs on distributed operating systems with IBM Software Development Kits (SDKs) Version 6.0 and Sun Java Development Kit (JDK) Version 6.0 that are provided by IBM. The Thin Client for JAX-RPC is supported on non-IBM SDKs Version 6.0 with this limitation:

- Xerces limitation on non-IBM SDKs

  If you are using a non-IBM SDK, because of dependencies with the Xerces implementation, you will need to download Xerces-J version 2.6.2 and set it in the classpath before attempting to run the Thin Client for JAX-RPC.

About this task

Set up a Thin Client for JAX-RPC environment by completing the following steps.

1. Configure the path. You can add the Java `bin` directories to your path by typing:

   ```
   set PATH=<your_JDK_bin_directory>;%PATH%
   ```

2. Configure the classpath.

   ```
   set CLASSPATH=.;<your_web_services_thin_client_install_directory>\com.ibm.ws.webservices.thinclient_7.0.0.jar;
   <your_application_jars>;%CLASSPATH%
   ```

   - If you are using a non-IBM SDK, obtain a Xerces xml-apis.jar and xercesImpl.jar from the Xerces Web site and configure the classpath definition.

     ```
     set CLASSPATH=.;<your_Xerces_install_directory>\xml-apis.jar;<your_Xerces_install_directory>
     \xercesImpl.jar;%CLASSPATH%
     ```

3. Configure SSL for the client.
   a. Add the following system properties to the Java command:

      ```
      -Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props
      ```

      You can obtain the ssl.client.props file from the WebSphere Application Server installation and modify the file to suit your environment. You must, at a minimum, update the location of the com.ibm.ssl.keyStore and com.ibm.ssl.trustStore key files in the ssl.client.props file to the match location of your target environment. For example, use these SSL configuration settings when running the application with a Sun JRE:

      ```
      com.ibm.ssl.protocol=SSL
      com.ibm.ssl.trustManager=SunX509
      com.ibm.ssl.keyManager=SunX509
      com.ibm.ssl.contextProvider=SunJSSE

      com.ibm.ssl.keyStoreType=JKS
      com.ibm.ssl.keyStoreProvider=SUN
      com.ibm.ssl.keyStore=/home/user1/etc/key.jks

      com.ibm.ssl.trustStoreType=JKS
      com.ibm.ssl.trustStoreProvider=SUN
      com.ibm.ssl.trustStore=/home/user1/etc/trust.jks
      ```

      The key store file and trust store file must be created using the Java keytool utility before the application runs. The automatic key file generation is not supported with a non-IBM product JRE.

      You must override the default ORB implementation of the non-IBM product JRE with the com.ibm.ws.orb_7.0.0.jar file, or add it to the classpath.
4. Enter the following command to run your client application:

   ```
   %JAVA_HOME%/bin/java -Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props <your_client_application>
   ```

Results

You have set up an unmanaged JAX-RPC client runtime environment that can be used to invoke Web services hosted on a WebSphere Application Server.

Related tasks

"Task overview: Implementing Web services applications" on page 415
Use this topic as an introduction to using Web services. WebSphere Application Server supports Web services that are developed and implemented based on a variety of Java programming models. Use Web services when operating across a variety of platforms, including Java Platform, Enterprise Edition (Java EE) and non-Java EE platforms.

"Installing Application Client for WebSphere Application Server" on page 369
This topic describes how to install the Application Client for WebSphere Application Server using the installation image on the product CD-ROM.

"Developing and deploying JAX-RPC Web services clients" on page 608
You can develop Web services clients based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification and the Java API for XML-based RPC (JAX-RPC) programming model.

Related information

Xerces Web site

Using HTTP to transport Web services

You can develop an HTTP accessible Web service when you have an existing JavaBeans object to enable as a Web service.

Before you begin

Develop a Web Services Description Language (WSDL) file. You need a WSDL file to use Web services. You can develop your own WSDL file or get one from a Web services provider through e-mail, downloading, or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own.

About this task

The application server supports the use of HTTP to transport Web services client requests. With HTTP, your Web services clients and servers can communicate through SOAP messages. SOAP is the underlying communication protocol that is used in Web services that support the Web Services for Java Platform, Enterprise Edition (Java EE), the Java API for XML-Based Web Services (JAX-WS), and the Java API for XML-based remote procedure call (JAX-RPC) specifications.

HTTP is the most commonly used transport for Web services.

1. Depending on your application programming model,
   - Use HTTP to transport Web Services requests for JAX-WS applications.
   - Use HTTP to transport Web Services requests for JAX-RPC applications.
2. Deploy the Web services application.
3. Configure security for the HTTP connection.

Results

You have a JavaBeans object that uses HTTP to transport Web services client requests.

What to do next

Publish the WSDL file.

Using HTTP to transport Web services requests for JAX-WS applications

You can develop an HTTP accessible Java API for XML-Based Web Services (JAX-WS) Web service when you have an existing JavaBeans object to enable as a Web service.

Before you begin

You must have an annotated JAX-WS JavaBeans object to enable as a Web service. Optionally, you can run the **wsgen** command to create a Web Service Description Language (WSDL) file from your annotated JAX-WS JavaBeans component. You must specify the `-wsdl` option with the **wsgen** command to create the WSDL file.

For example:

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

To learn about developing a JAX-WS Web service using annotations, read about developing Java artifacts for JAX-WS applications using JavaBeans.

About this task

The application server supports the use of HTTP to transport Web services client requests. With HTTP, your Web services clients and servers can communicate through SOAP messages. SOAP is the underlying communication protocol that is used in Web services that support the Web Services for Java Platform, Enterprise Edition (Java EE) and the Java API for XML-Based Web Services (JAX-WS) specifications.

HTTP is the most commonly used transport for Web services.

1. Add an HTTP binding and a SOAP address to the WSDL file.

   The WSDL file of a Web service must include an HTTP binding and a SOAP address, which specifies an HTTP endpoint URL string, that is accessible on the HTTP transport. An HTTP binding is a wsdl:binding element that contains a soap:binding element with a transport attribute that ends in `soap/http`.

   In addition to the HTTP binding, a wsdl:port element that references the HTTP binding must be included in the wsdl:service element within the WSDL file. The wsdl:port element contains a soap:address element with a location attribute that specifies an HTTP endpoint URL string.

   When you develop the Web service, you can use a placeholder such as `file:unspecified_location` for the endpoint URL string.

   Note: If you deploy a JAX-WS JavaBeans component as a Web service without a WSDL file, a WSDL file is automatically generated for the component.
2. For JAX-WS Web services applications, no HTTP transport configuration is needed. The HTTP transport settings are generated dynamically by the application server. The Web archive {WAR) file only needs the JavaBeans object along with the optional WSDL file properly installed.
3. Deploy the Web services application.
4. Configure security for the HTTP transport.

   To configure a secure HTTP transport, attach the SSLTransport policy to the application. To specify the basic authentication transport token, use the administration console to set the user ID and the password attributes in the HTTPTransport binding.
5. (Optional) Configure HTTP session management.

   HTTP session management enables JAX-WS Web Services applications to appear dynamic to application users.
6. (Optional) Configure the asynchronous response listener for JAX-WS clients.

   You can use the asynchronous response listener within the Thin Client for JAX-WS and application client environments to receive responses for requests that are invoked asynchronously.
7. Configure the endpoint URL information for HTTP bindings.

   The WSDL publisher uses this partial URL string to produce the actual HTTP URL for each port component defined in the enterprise archive (EAR) file. The published WSDL file can be used by clients, that need to invoke the Web service.

Results

You have a JavaBeans object that uses HTTP to transport JAX-WS Web services client requests.

What to do next

Publish the WSDL file.

Using the asynchronous response servlet

Java API for XML-Based Web Services (JAX-WS) includes an asynchronous response servlet, which is used within the application server environment to receive responses for JAX-WS requests that are invoked asynchronously.

Before you begin

JAX-WS provides support for invoking Web services using an asynchronous client invocation by using either a callback or polling model. Both the callback model and the polling model are available on the Dispatch client and the dynamic proxy client. When a JAX-WS client that is running within the application server environment uses an asynchronous client invocation, the responses are received by the asynchronous response servlet. To learn how to use the asynchronous client invocation model, read about invoking JAX-WS Web services asynchronously.

About this task

The asynchronous response servlet is used within an application server to handle incoming asynchronous responses. The servlet uses the same secure and unsecure HTTP ports assigned to the application server. The servlet starts automatically when the application server starts. Because the asynchronous response servlet does not perform role-based authorization checks, only user authentication checks are performed.

The asynchronous response servlet supports both the HTTP and HTTPS protocols. Since the servlet inherits the SSL configuration of the application server, configuring the application server also configures the servlet. The asynchronous response servlet is not affected by the custom HTTP and SSL port properties used by the asynchronous response listener and only runs on the application ports for the application server.

1. Determine if you want the JAX-WS client to use the HTTP or HTTPS transport mechanism.
2. Configure the Web container transport chains to modify the SSL configuration of the application server. The servlet inherits these settings. Read about configuring transport chains to learn how to configure the Web container transport chains.

Results

The asynchronous response servlet is configured to enable your JAX-WS clients to receive asynchronous responses on the HTTP or HTTPS transport protocol.

Using the asynchronous response listener

Java API for XML-Based Web Services (JAX-WS) includes an asynchronous response listener, which is used within the Thin Client for JAX-WS and application client environments to receive responses for requests that are invoked asynchronously.

Before you begin

JAX-WS provides support for invoking Web services using an asynchronous client invocation by using either a callback or polling model. Both the callback model and the polling model are available on the Dispatch client and the dynamic proxy client. When the JAX-WS client uses an asynchronous client invocation, the responses are received by the asynchronous response listener. To learn how to use the asynchronous client invocation model, read about invoking JAX-WS Web services asynchronously.

About this task

The asynchronous response listener is used within a Web services client to handle incoming asynchronous responses. You can use the listener in Thin Client for JAX-WS environments and application

client environments. By default, the listener opens a random port to listen for asynchronous responses or you can optionally configure a specific port for the listener to use. The listener starts automatically in the JAX-WS run time when the JAX-WS client is configured to expect an asynchronous response.

There are two versions of the asynchronous response listener. The unsecure version of the asynchronous response listener supports the HTTP protocol, and the secure version of the asynchronous response listener supports the HTTPS protocol. The correct asynchronous response listener is automatically started based on the particular transport used by the JAX-WS client. To ensure that the correct Secure Sockets Layer (SSL) handshaking occurs between the asynchronous response listener and the application server, configure the SSL properties using the SSL transport policy or the Java system properties.

For Web services clients running in the application server environment, use the asynchronous response servlet for receiving asynchronous responses.

1. Determine if you want the JAX-WS client to use the HTTP or HTTPS transport mechanism.
2. Configure the asynchronous response listener for unsecure communication using HTTP.

   You can configure the HTTP port for the asynchronous response listener as a Java system property or as a custom property within the transport policy. Properties that are defined in the policy set binding files override any Java system property that might have been defined.

   a. Define the `com.ibm.websphere.webservices.http.listenerPort` property as a Java system property. If this property is set as a Java system property, then all asynchronous response listeners within that Java Virtual Machine (JVM) are affected.
   b. Define the `com.ibm.websphere.webservices.http.listenerPort` property within the `HTTPTransport` transport policy set bindings files. If this property is set as a custom property within a transport policy set binding, then only the services for which the policy set has been configured are affected.
3. Configure the asynchronous response listener for secure communication using HTTPS.

   You can configure the HTTPS port for the asynchronous response listener as a Java system property or as a custom property within the transport policy.

   a. Define the `com.ibm.websphere.webservices.https.listenerPort` property as a Java system property. If this property is set as a Java system property, thenall asynchronous response listeners within that JVM are affected.
   b. Define the `com.ibm.websphere.webservices.https.listenerPort` property within the `SSLTransport` transport policy set bindings files. If this property is set as a custom property within a transport policy set binding, then only the services for which the policy set has been configured are affected.

Results

Your JAX-WS Web services client is configured to use the asynchronous response listener to receive incoming asynchronous responses.

Example

The following examples demonstrate how to enable the asynchronous response listener when defining the custom port of 9999:

Use the following Java command to configure the custom HTTP port for the asynchronous response listener in a thin client environment:

```
- java.exe -Dcom.ibm.websphere.webservices.http.listenerPort=9999 com.ibm.websphere.my_program
```

Use the following launchClient command to configure the custom HTTP port for the asynchronous response listener in an application client container:

```
- launchClient.bat MyClient.ear -CCDcom.ibm.websphere.webservices.http.listenerPort=9999
```

The following is an excerpt from an HTTPTransport policy binding.xml file that includes the asynchronous response listener properties:

```
</wsp:Policy>
   </wsp:ExactlyOne>
 </wsp:All>
    <wshttp:outAsyncResponseProxy>
     <wshttp:connectInfo host="" port=""></wshttp:connectInfo>
                  <wshttp:basicAuth userid="" password=""></wshttp:basicAuth>
    </wshttp:outAsyncResponseProxy>
    <wshttp:properties>
  <wshttp:customProperty name="com.ibm.websphere.webservices.http.listenerPort" value="9999" />
   </wshttp:properties>
 </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

What to do next

Run the JAX-WS client with the specified asynchronous response listener options.

Using HTTP session management support for JAX-WS applications

HTTP session management is performed in the HTTP transport layer by using either cookies or URL rewriting. By providing multiple options for tracking a series of requests, HTTP session management enables Java API for XML-Based Web Services (JAX-WS) applications to appear dynamic to application users.

Before you begin

Develop a JAX-WS dynamic proxy or Dispatch client. To learn more about developing JAX-WS clients, read about developing a JAX-WS client from a Web Services Description Language (WSDL) file or developing a dynamic client using JAX-WS APIs.

About this task

You can use HTTP session management to maintain user state information on the server, while passing minimal information back to the user to track the session. You can implement HTTP session management on the application server using either session cookies or URL rewriting.

The interaction between the browser, application server, and application is transparent to the user and the application program. The application and the user are typically not aware of the session identifier provided by the server.

Session cookies

The HTTP maintain session feature uses a single cookie, JSESSIONID, and this cookie contains the session identifier. This cookie is used to associate the request with information stored on the server for that session. On subsequent requests from the JAX-WS application, the session ID is transmitted as part of the request header, which enables the application to associate each request for a given session ID with prior requests from that user. The JAX-WS client applications retrieve the session ID from the HTTP response headers and then use those IDs in subsequent requests by setting the session ID in the HTTP request headers.

URL rewriting

URL rewriting works like a redirected URL as it stores the session identifier in the URL. The session identifier is encoded as a parameter on any link or form that is submitted from a Web page. This encoded URL is used for subsequent requests to the same server.

1. Configure the server to enable session tracking. Specify either session cookies or URL rewriting for your session tracking mechanism using the administration console. Read about configuring session tracking to learn how to track sessions with cookies or with URL rewriting.
2. Enable HTTP session management for the JAX-WS client. You can manually set the maintainSession property to **true** in the policy set binding.
   - Enable session management on the client using a HTTP transport policy set file. You can configure the HTTP transport policy using the administration console. On the HTTP transport setting page, select the **Session enable** checkbox to enable an HTTP session. Read about configuring the HTTP transport property to learn how to set this property. You can also manually set the `maintainSession` property to `yes` to enable HTTP session management.
   - Enable session management on the client by setting the JAX-WS property, `javax.xml.ws.session.maintain`, to `true`.

Results

You have enabled HTTP session management for your JAX-WS application.

Example

The following example is an excerpt from a HTTP Transport policy set that demonstrates how to configure the `maintainSession` property:

```
<!-- This is the PolicyType for HTTP transport -->
        <wsp:ExactlyOne>
                <wsp:All>
                        <wshttp:readTimeout>300</wshttp:readTimeout>
                        <wshttp:writeTimeout>300</wshttp:writeTimeout>
                        <wshttp:connectTimeout>180</wshttp:connectTimeout>
                        <wshttp:persistConnection>yes</wshttp:persistConnection>
                        <wshttp:messageResendOnce>no</wshttp:messageResendOnce>
                        <wshttp:chunkTransferEnc>no</wshttp:chunkTransferEnc>
                        <wshttp:acceptRedirectedURL>no</wshttp:acceptRedirectedURL>
                        <wshttp:sendExpectHeader>no</wshttp:sendExpectHeader>
                        <wshttp:maintainSession>yes</wshttp:maintainSession>
                        <wshttp:compressRequest>
                                <wshttp:compressType name="none"></wshttp:compressType>
                        </wshttp:compressRequest>
                        <wshttp:compressResponse>
                                <wshttp:compressType name="none"></wshttp:compressType>
                        </wshttp:compressResponse>
                        <wshttp:protocolVersion>HTTP/1.1</wshttp:protocolVersion>
                </wsp:All>
        </wsp:ExactlyOne>
```

The following code example is demonstrates how to programmatically enable session management on the client by setting the javax.xml.ws.session.maintain property on the correct JAX-WS object.

```
Map<String, Object> rc = ((BindingProvider) port).getRequestContext();
...
...
rc.put(BindingProvider.SESSION_MAINTAIN_PROPERTY, Boolean.TRUE);
...
...
```

Using HTTP to transport Web services requests for JAX-RPC applications

You can develop an HTTP accessible Java API for XML-based remote procedure call (JAX-RPC) Web service when you already have a JavaBeans object to enable as a Web service.

Before you begin

Run the **Java2WSDL** command to create a Web Services Description Language (WSDL) file. When you run the **Java2WSDL** command, use the -bindingsTypes option, along with `http`, to set the HTTP transport bindings. For example:

```
java2wsdl -bindingTypes http,jms -implClass my.pkg.MyEJBClass my.pkg.MySEI
```

To learn more about using the **Java2WSDL** command, see the Java2WSDL command for JAX-RPC applications documentation.

The Java2WSDL command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the Java2WSDL command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

About this task

The application server supports the use of HTTP to transport Web services client requests. With HTTP, your Web services clients and servers can communicate through SOAP messages. SOAP is the underlying communication protocol that is used in Web services that support the Web Services for Java Platform, Enterprise Edition (Java EE) and the Java API for XML-based remote procedure call (JAX-RPC) specifications.

HTTP is the most commonly used transport for Web services.

To develop an HTTP-accessible Web service from an existing an existing JavaBean object:

1. Add an HTTP binding and a SOAP address to the WSDL file.

   The WSDL file of a Web service must include an HTTP binding and a SOAP address, which specifies an HTTP endpoint URL string, to be accessible on the HTTP transport. An HTTP binding is a wsdl:binding element that contains a wsdlsoap:binding element with a transport attribute that ends in `soap/http`.

   In addition to the HTTP binding, a wsdl:port element that references the HTTP binding must be included in the wsdl:service element within the WSDL file. The wsdl:port element contains a wsdlsoap:address element with a location attribute that specifies an HTTP endpoint URL string.

   When you develop the Web service, a placeholder such as `file:unspecified_location` can be used for the endpoint URL string.
2. Add the HTTP endpoints to your enterprise archive (EAR) file using the **endptEnabler** command, if your application includes enterprise beans.

   By default, the **endptEnabler** command adds only HTTP endpoints.
3. Deploy the Web services application.
4. Configure security for the HTTP connection.

   For a secure HTTP connection, add the basicAuth assembly property to the `ibm-webservicesclient-bnd.xmi` deployment descriptor file. Set the user ID and the password attributes.
5. Configure the endpoint URL information for HTTP bindings.

   The WSDL publisher uses this partial URL string to produce the actual HTTP URL for each port component defined in the EAR file. The published WSDL file can be used by clients, that need to invoke the Web service.

Results

You have a JavaBean object that uses HTTP to transport Web services client requests.

What to do next

Publish the WSDL file.

Related concepts

Assembly tools
WebSphere Application Server supports *assembly tools* that you can use to develop, assemble, and deploy Java Platform, Enterprise Edition (Java EE) modules.

Related reference

"Java2WSDL command for JAX-RPC applications" on page 577
The Java2WSDL command-line tool maps Java classes to a WSDL file for Java API for XML-based RPC (JAX-RPC) applications.

Using SOAP over Java Message Service to transport Web services

You can use the SOAP over Java Message Service (JMS) transport protocol as an alternative to SOAP over HTTP for communicating SOAP messages between clients and servers.

Before you begin

A Web service must be implemented as an enterprise bean for accessibility through the JMS transport.

About this task

Note: WebSphere Application Server Version 7.0 introduces support for an emerging industry standard SOAP over JMS protocol. The SOAP over JMS specification provides a standard set of interoperability guidelines for using a JMS-compliant transport with SOAP messages to enable interoperability between the implementations of different vendors. Using this standard, a mixture of client and server components from different vendors can interoperate when exchanging SOAP request and response messages over the JMS transport for both Java API for XML Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) Web services. By using the JMS transport, your enterprise beans based Web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections.

Note:

The benefits of using JMS include:
- Reliable messaging transport for communicating request and response messages.
- Flexible one-way requests for clients and servers. For example, the server does not have to be active when the client sends the one-way request. Simultaneous one-way requests can be sent to multiple servers through the use of a topic.
- Synchronous two-way requests are supported for both JAX-WS and JAX-RPC clients.
- Asynchronous requests are supported for JAX-WS clients.

The SOAP over JMS specification defines a JMS endpoint URI syntax for specifying JMS destinations. A JMS endpoint URL is used to access JAX-WS or JAX-RPC Web services with the JMS transport. This URL specifies the JMS destination and connection factory, as well as the port component name for the Web service request. This endpoint URL is similar to the HTTP endpoint URL, which specifies the host and port as well as the context root and port component name.

1. Develop the enterprise bean that you intend to use as your service implementation bean.
   - For JAX-WS applications, develop an enterprise beans- based JAX-WS service implementation bean.
   - For JAX-RPC applications, develop an enterprise beans-based JAX-RPC service implementation bean.

2. Develop Java artifacts for JAX-WS applications. Although a Web Services Description Language (WSDL) file is typically optional when developing a JAX-WS service implementation bean, it is required if your JAX-WS endpoints are exposed using the SOAP over JMS transport and you are publishing your WSDL file. Therefore, if you are developing a JAX-WS web service using the bottom-up approach, use the wsgen tool to generate a WSDL file from your JAX-WS implementation bean, and then package the WSDL file along with any associated schema files with your application.
3. Modify your WSDL file so that the ports that are accessible using the SOAP over JMS transport have an endpoint location URL that starts with the `jms:` prefix. For example:

```
<service name="MyService">
    <port name="MyJMSPort" binding="tns:MyJMSPortBinding">
      <soap:address location="jms:my_actual_endpoint_URL"/>
    </port>
</service>
```

   Using the `jms:` prefix enables the WSDL publisher to identify the `MyJMSPort` port as a JMS port, and a JMS-style endpoint location URL is used when publishing the WSDL.
4. Assemble the enterprise beans.
   a. Assemble a JAR file that is enabled for Web services from an enterprise bean. You can assemble the artifacts that are required to enable the enterprise beans module for Web services into a Java archive (JAR) file.
   b. Assemble a Web services-enabled enterprise bean JAR file into an enterprise archive (EAR) file. You can assemble the artifacts that are required to enable the Web services-enabled JAR file into an EAR file.
5. Enable the enterprise beans-based endpoints by using the endptEnabler command. Use the `-transport jms` option to request that the endptEnabler command create a message-driven bean (MDB) listener for each Enterprise JavaBeans (EJB) JAR file that contains Web services implementation beans. This message-driven bean serves as a listener for requests that are associated with the Web service endpoints contained in the EJB JAR file.
6. Decide on the names and the types of the JMS objects that your application uses.

   Before you install your application, you need to:

   - Decide whether your Web service receives requests from a queue or a topic.
     - Decide whether to use a secure destination or a nonsecure destination.
     - Decide the names for your queues and topics, connection factory and activation specification.

   Use the following guidelines for deciding JMS object names and types. In typical situations, you can use a queue for receiving Web services requests.

   - Queue
     - A queue receive all types of requests. Valid requests include one-way, two-way, and synchronous. Asynchronous requests are only valid with JAX-WS Web services.
     - A queue is used only by a single EJB JAR file for receiving the requests for Web services endpoints contained within that EJB JAR file.
   - Topic
     - A topic is used to receive only one-way requests.
     - You can share a topic among multiple EJB JAR files. Each request message that is sent to a topic is processed by each of the MDB listeners that are configured to listen on that topic. This means that each request message is processed by each EJB JAR file associated with that specific topic.

   The following example describes a typical configuration for a single EJB JAR file containing Web services endpoints:

   - Suppose the EJB JAR file is StockQuoteEJB.jar and contains one or more Web services endpoints related to the StockQuote service.

- You have a single queue, StockQuote_Q, with the JNDI name, `jms/StockQuote_Q`, that is used to receive requests.
- You have a connection factory, StockQuote_CF, with JNDI name, `jms/StockQuote_CF`, that can be used by clients to connect to the JMS provider.
- You also have a connection factory, StockQuote_ReplyCF, with JNDI name, `jms/StockQuote_ReplyCF`, that is used by the MDB listener for the EJB JAR file to connect to the JMS provider when sending reply messages.
- You have an activation specification, StockQuote_AS, with JNDI name, `jms/StockQuote_AS`, that is used to associate the StockQuoteEJB.jar's MDB listener with the queue named StockQuote_Q.

7. Define the JMS administered objects.

   After you decide on the names and types of the JMS objects, use the administrative console or the wsadmin scripting tool to define the JMS objects. There are multiple ways to administer JMS resources depending on what type of JMS provider is being used. Read about choosing a messaging provider to learn more about administering JMS resources.

8. Deploy the Web services application.

   During the installation process you are prompted for two types of information for each enterprise bean JAR file that is enabled for Web services and is contained in your EAR file:
   - The JNDI name of the connection factory that is used by the MDB listener to use for sending reply messages.

     If your Web service contains two-way operations, the MDB listener that is defined by the endptEnabler command needs to access a queue connection factory to add a reply message to the reply queue. The MDB listener uses a resource environment reference of `java:comp/env/jms/WebServicesReplyQCF`. Therefore, during the application installation process, you must provide the actual JNDI name of the connection factory for the MDB listener to use for that Web service. Using the previous example, the JNDI name is `jms/StockQuote_ReplyCF`.
   - The name of the activation specification for the MDB listener to use.

     An activation specification is an object that is used to associate a JMS connection factory with a JMS destination (queue or topic). When deployed, an MDB is configured with the correct activation specification so that messages from the queue or topic are properly delivered to the MDB. During deployment, you can modify the name of the activation specification that is associated with each MDB listener. The activation specification name contained in the input EAR file is displayed as a default value. If you specify the correct activation specification name to the endptEnabler command, you can accept the default value. Otherwise, enter the correct activation specification name.

9. Decide whether to use the new industry standard SOAP over JMS protocol or the IBM proprietary SOAP/JMS protocol.

   Note: It is a best practice to use the industry standard SOAP/JMS protocol. The IBM proprietary SOAP/JMS protocol has been deprecated with this release. However, if your application needs to interoperate with previous versions of the product, then use the proprietary protocol.
   - If you use the industry standard SOAP over JMS protocol, use the Configure endpoint URL information for JMS bindings task to specify a JMS endpoint URL prefix that adheres to the JMS endpoint URI syntax that is associated with the standard; for example:

     ```
     jms:jndi:jms/StockQuote_Q&jndiConnectionFactoryName=jms/StockQuote_CF
     ```
   - If you use the IBM proprietary SOAP over JMS protocol, use the Configure endpoint URL information for JMS bindings task to specify a JMS endpoint URL prefix that adheres to the IBM proprietary SOAP over JMS protocol; for example:

     ```
     jms:/queue?destination=jms/StockQuote_Q&connectionFactory=jms/StockQuote_CF
     ```

10. (Optional) Configure endpoint URL information for JMS bindings.

    Use the administrative console to configure a JMS endpoint URL prefix that you can associate with each EJB JAR module in your application. The WSDL publisher uses this partial URL string to

produce the actual JMS URL for each port component that is defined in the enterprise bean JAR file. The clients that need to invoke the Web service can use the published WSDL file.

Perform this step only if you are publishing the WSDL file for your application.

11. (Optional) Publish the WSDL file for your application.

    Publishing the WSDL file, this produces WSDL documents that you can use to develop your client applications. The publishing process produces fully-resolved endpoint location URLs within the WSDL files.

    Perform this step only if the published WSDL files are needed to develop your client applications.

Results

You have a Web service that is configured to use SOAP over JMS to transport the requests.

What to do next

Develop a Web services client.

SOAP over Java Message Service (JMS) protocol

The Web services engine supports the use of an emerging industry standard SOAP over Java Message Service (JMS)-compliant messaging transport as an alternative to HTTP for communicating SOAP messages between clients and servers.

Note: WebSphere Application Server Version 7.0 introduces support for an emerging industry standard SOAP over JMS protocol. The SOAP over JMS specification provides a standard set of interoperability guidelines for using a JMS-compliant transport with SOAP messages to enable interoperability between the implementations of different vendors. Using this standard, a mixture of client and server components from different vendors can interoperate when exchanging SOAP request and response messages over the JMS transport for both Java API for XML Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) Web services. By using the JMS transport, your enterprise beans based Web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections.

IBM and other vendors have been working on the proposed SOAP over JMS specification since 2005. The specification has been submitted to W3C and a working group is established. The current member submission of this draft specification was jointly published in October, 2007. Refer to the SOAP over JMS specification for details of this industry standard.

This topic provides a summary of the emerging industry-standard SOAP over JMS protocol. Use this SOAP over JMS transport protocol if you need to provide implementations for the client or server components. Also, you need to make sure that the implementations are interoperable with the client and server components provided by the Web services engine in WebSphere Application Server.

Client responsibilities

The client component is responsible for sending SOAP request messages and receiving SOAP response messages while adhering to the following protocol constraints:

- The client must use a BytesMessage, for example, javax.jms.BytesMessage, to transmit the SOAP request message to the server.
- The client must set the following properties on the JMS request message before sending the message to the destination queue or topic:
  - **SOAPJMS_contentType**: This property is similar to the Content-Type header found in an HTTP message and is used to describe the content type of the message. A text-only SOAP message, for example, a message with no attachments, has the following value set for this JMS message property:

```
text/xml; charset="UTF-8"
```

For a SOAP message containing attachments, use the following code to set the SOAPJMS_contentType property on the JMS message:

```
multipart/related; type="text/xml"; start="<...content-id_of_first_ part...>"
```

This example represents a multipart message, where the first part is of type `text/xml` and contains the SOAP envelope. The other parts of the multipart message contain various attachments. The HTTP 1.1 specification contains more information about the Content-Type header.

- **enableTransaction**: Set this optional property to true on an outbound JMS request message if you want the server component to process the Web service request under the same transaction that was used to receive the message from the destination queue or topic.

  Note: For client components, only set this property to true for one-way or two-way asynchronous requests to avoid synchronization problems that can occur with two-way synchronous Web service requests. If this property is not set or is set to the default value of false, the server suspends the transaction that was used to receive the request message from the destination queue or topic prior to invoking the Web services engine to process the request.
- **SOAPJMS_requestIRI**: You must set this property to the JMS endpoint URL associated with the request.
- **SOAPJMS_soapAction**: This optional property is set on an outbound JMS request message to indicate the soapAction value associated with the Web services request. This property is similar to the SOAPAction HTTP header used when transporting Web service requests over an HTTP transport. The value of the soapAction property is a URI identifying the intent of the SOAP request. If the SOAPJMS_soapAction property is specified, it is used by the server component to determine the target of the request. The SOAP specification places no restrictions on the format or specificity of the URI nor does the specification require that the URI is resolvable. Typically, this property is set to the soapAction value from the WSDL document.
- **SOAPJMS_targetService**: You must set this property on an outbound JMS request message, and the value must match the targetService property value that is found in the JMS endpoint URL for the request. This value is used by the server component to determine the port component to which the request is dispatched.
- **SOAPJMS_bindingVersion**: This property indicates the version number of the protocol used by the client and server. Set the value to 1.0.

- If the request message represents a two-way request, meaning that a reply is expected, the client component must set the JMS message JMSReplyTo property to specify the queue that is used for the reply message. The JMS message setJMSReplyTo method is used to specify the queue. You can benefit from configuring a permanent reply queue on the client to prevent the client from having to create a temporary queue each time a Web service request is made. Read about configuring a permanent reply queue for Web services using SOAP over JMS to learn more about creating this special queue.
- If the SOAP request message represents a one-way request, meaning that a reply message is not expected, the client component must not set the JMS message JMSReplyTo property.
- The client component can assume that a reply message is a JMS BytesMessage object.
- The client component can assume that the reply message correlation ID matches the message ID of the original request message.

Server responsibilities

The server component is responsible for receiving the SOAP request messages and sending the SOAP response messages, while adhering to the following protocol constraints:

- The server component can expect to receive a JMS BytesMessage. If something other than a BytesMessage is received by the server component, then a fault with the subcode, unsupportedJMSMessageFormat, is returned to the client if a reply is expected.

- The server component must process the SOAP request properly to produce an appropriate SOAP reply message.
- The server component must send a reply message back to the client only if the JMS request message's JMSReplyTo property is set. The JMS message getJMSReplyTo method is used to retrieve the JMSReplyTo property value from the JMS message.
- The server component must set the following properties in the JMS reply message before sending the message to the reply queue:
  - **SOAPJMS_contentType**: This property is used to describe the content type of the message. See the description for this property in the client responsibilities section in this topic.
  - **correlation ID**: Set the correlation ID property of the JMS reply message to the message ID of the original JMS request message. This correlation is done by calling the JMS message setJMSCorrelationID method.
  - **SOAPJMS_bindingVersion**: This property indicates the version number of the protocol used by the client and server. Set the value to 1.0.

Example: SOAP request without attachments

The following example displays the results from calling the JMS message toString method for a request message without attachments:

```
  JMSMessage class: jms_bytes
  JMSType:          null
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     null
  JMSTimestamp:     -1
  JMSCorrelationID: null
  JMSDestination:   null
  JMSReplyTo:       queue://_Q_7D6C2035383215AB00000000000F4241?busName=WsFvtBus
  JMSRedelivered:   false
    JMS_IBM_MsgType: 1
    SOAPJMS_contentType: text/xml; charset=UTF-8
    SOAPJMS_targetService: MaelstromWsEndpoint
    SOAPJMS_requestIRI: jms:jndi:jms/MyRequestQueue?jndiConnectionFactoryName=jms/MyConnFactory&targetService=MyPort1
    SOAPJMS_soapAction: "getQuote"
    SOAPJMS_bindingVersion: 1.0
3c3f786d6c2076657273696f6e3d22312e302220656e636f64696e673d227574662d38223f3e3c73
6f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f7363
68656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c6e733a
7873643d22687474703a2f2f7777772e77332e6f72672f323030312f584d4c536368656d61222078
...
```

Example: SOAP request with attachments

The following example displays the results from calling the JMS message toString method for a request message with attachments:

```
  JMSMessage class: jms_bytes
  JMSType:          null
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     null
  JMSTimestamp:     -1
  JMSCorrelationID: null
  JMSDestination:   null
  JMSReplyTo:       queue://_Q_F0940794C5CC2F84000000000044AA21?busName=WsFvtBus
  JMSRedelivered:   false
    JMS_IBM_MsgType: 1
    SOAPJMS_contentType: multipart/related;
  boundary=MIMEBoundaryurn_uuid_B6BAFEADB1886ADC241205525550237;
```

```
  type="text/xml"; start="<0.urn:uuid:B6BAFEADB1886ADC241205525550238@apache.org>"
    SOAPJMS_targetService: MaelstromWsEndpoint
    SOAPJMS_requestIRI: jms:jndi:jms/WebSvcsJMSQ?jndiConnectionFactoryName=
   jms/WebSvcsJMS_CF&targetService=MaelstromWsEndpoint
    SOAPJMS_soapAction: attachment
    SOAPJMS_bindingVersion: 1.0
2d2d4d494d45426f756e6461727975726e5f757569645f42364241464541444231383836414443332
343132303535323535303233370d0a436f6e74656e742d547970653a20746578742f786d6c3b20
636861727365743d5554462d380d0a436f6e74656e742d5472616e736665722d456e636f64696e67
3a20386269740d0a436f6e74656e742d49443a203c302e75726e3a757569643a4236424146454144
42313838364144433234313230353532353535303233384061706163686526e6f72673e0d0a0d0a3c
736f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f73
6368656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c6e73
3a7873643d22687474703a2f2f7777772e77332e6f72672f323030312f584d4c536368656d612220
786d6c6e733a7873693d22687474703a2f2f7777772e77332e6f72672f323030312f584d4c536368
656d612d696e7374616e63652220786d6c6e733a736f6170656e633d22687474703a2f2f73636865
...
```

SOAP response

The following example displays the results from calling the JMS message toString method for a SOAP reply message:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     null
JMSTimestamp:     0
JMSCorrelationID: ID:cdddb857f078a266eb9a972f110a134f0000000000000001
JMSDestination:   null
JMSReplyTo:       null
JMSRedelivered:   false
contentType:
  multipart/related;
  type="text/xml";
  start="<961368106530.1092112854745.IBM.WEBSERVICES@yackerjr>";
  boundary="----=_Part_0_1655006754.1092112854745"
0d0a2d2d2d2d2d2d3d5f506172745f305f313635353030363735342e313039323131323835343734
350d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d38
...
```

JMS endpoint URL syntax

As part of an emerging industry-standard SOAP over JMS protocol, a Java Message Service (JMS) endpoint URL syntax has been defined. A JMS endpoint URL is used to access Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) Web services with the JMS transport. This URL specifies the JMS destination and connection factory, as well as the port component name for the Web service request. This endpoint URL is similar to the HTTP endpoint URL, which specifies the host and port as well as the context root and port component name.

Note: WebSphere Application Server Version 7.0 introduces support for an emerging industry standard SOAP over JMS protocol. The SOAP over JMS specification provides a standard set of interoperability guidelines for using a JMS-compliant transport with SOAP messages to enable interoperability between the implementations of different vendors. Using this standard, a mixture of client and server components from different vendors can interoperate when exchanging SOAP request and response messages over the JMS transport for both Java API for XML Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) Web services. By using the JMS transport, your enterprise beans based Web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections.

IBM and other vendors have been working on the W3C SOAP over JMS specification since 2005. The specification has been submitted to W3C and a working group is established. The current member submission of this document was jointly published in October, 2007. The application server supports the current draft specification from W3C.

Note: A JMS endpoint URL has the following general form:

```
jms:jndi:<destination-jndi-name>?<property>=<value>&<property>=<value>&...
```

The URL consists of the jms: transport type, followed by the jndi: variant type, followed by the JNDI name of the destination queue or topic, followed by the query string containing a list of property and value pairs that are used to specify various JMS endpoint information. The jndi: variant means that JNDI is used to locate object names in the endpoint URL string.

The properties supported in the URL string are described in the following tables:

Destination-related properties (required)

Property name	Description
jndiConnectionFactoryName	Specifies the JNDI name of the connection factory that is used by the client runtime to establish a connection to the JMS messaging engine.
targetService	Specifies the name of the port component to which the request is dispatched.

JNDI-related properties (optional)

Property name	Description
jndiInitialContextFactory	Specifies the name of the initial context factory class to use. This value maps to the java.naming.factory.initial property.
jndiURL	Specifies the JNDI provider URL. This value maps to the java.naming.provider.url property.

JMS-related properties (optional)

Property name	Description
deliveryMode	Indicates whether the request message is persistent or not. The valid values are PERSISTENT and NON_PERSISTENT. The default value is NON_PERSISTENT.
timeToLive	Specifies the lifetime, in milliseconds, of the request message. A value of 0 indicates an infinite lifetime. If this parameter is not specified, then the JMS-defined default value is used.
priority	Specifies the JMS priority associated with the request message. Specify this value as a positive integer from 0, the lowest priority, to 9, the highest priority. If this parameter is not specified, then the JMS-defined default value is used.
replyToName	Specifies the JNDI name of the JMS destination to which the response message is sent. Using this optional property enables the client to use a previously defined, permanent queue rather than a temporary queue, for receiving replies.

The required properties jndiConnectionFactoryName and targetService must be in the JMS endpoint URL string. The remaining properties are optional.

If you set values for the deliveryMode, timeToLive, and priority properties on the JMS request, these values are propagated from the JMS request message to the corresponding JMS reply message.

See the SOAP over Java Message Service specification in the Web services specifications and APIs documentation to learn more about this industry standard.

IBM proprietary SOAP over JMS protocol (deprecated)

You can use a SOAP over Java Message Service (JMS) transport as an alternative to HTTP for communicating SOAP messages between clients and servers. The Web services engine supports the use of an IBM proprietary implementation as well as the industry standard implementation.

Note: In earlier versions of the application server, an IBM proprietary SOAP over JMS protocol was supported for Java API for XML-based RPC (JAX-RPC) applications. In WebSphere Application Server 7.0, this proprietary SOAP over JMS protocol is now deprecated in favor of an emerging industry standard SOAP over JMS protocol. You can use the IBM proprietary SOAP over JMS protocol with your Java API for XML Web Services (JAX-WS) or JAX-RPC Web services; however, take advantage of the emerging standard SOAP over JMS protocol. If your client application invokes enterprise beans-based Web services that are supported by an earlier version of the WebSphere Application Server, you must continue to use the IBM proprietary SOAP over JMS protocol to access those Web services.

You can use a SOAP over JMS transport if you need to provide implementations for the client or server components, and you need to make sure that the implementations are interoperable with the client and server components provided by the Web services engine in the application server. The IBM proprietary SOAP over JMS protocol describes specific message exchange requirements for client and server components so they can exchange SOAP request and response messages through the use of the JMS APIs supported by the application server.

Client responsibilities

The client component is responsible for sending SOAP request messages and receiving SOAP response messages while adhering to the following protocol constraints:

- The client must use either a JMS TextMessage object, for example, javax.jms.TextMessage, or a BytesMessage object, for example, javax.jms.BytesMessage, to transmit the SOAP request message to the server. If the request message contains attachments, a BytesMessage object must be used. If the request message does not contain attachments, the client can use a TextMessage or a BytesMessage object. The application server client implementation uses only a BytesMessage object for the request message due to the potential need to transmit attachments.
- The client must set the following properties on the JMS request message before sending the message to the destination queue or topic:
  - **contentType**: This property is similar to the Content-Type header found in an HTTP message and is used to describe the content type of the message. A text-only SOAP message, for example, a message with no attachments, is written as follows:

    ```
    text/xml; charset="UTF-8"
    ```

    The **contentType** property in a SOAP request message that contains attachments must be set as follows:

    ```
    multipart/related; type="text/xml"; start="<...content-id of first part...>"
    ```

    This example represents a multi-part message, where the first part is of type `text/xml` that contains the SOAP message. The other parts of the multi-part message contain various attachments. The HTTP 1.1 specification contains more information about the Content-Type header.
  - **enableTransaction**: Set this optional property to `true` on the outgoing SOAP over JMS request message if the server component should process the Web service request under the same transaction that was used to receive the message from the destination queue or topic. The client component should only set this property to `true` for a one-way request to avoid synchronization problems that can occur with a two-way Web service request. If this property is not set or is set to the default value of `false`, then the server will suspend the transaction that was used to receive the request message from the destination queue or topic prior to invoking the Web services engine to process the request.

  - **endpointURL**: This property must be set to the JMS endpoint URL associated with the request.
  - **soapAction**: This optional property is set on an outgoing SOAP over JMS request message to indicate the soapAction value associated with the Web services request. This property is similar to the SOAPAction HTTP header used when transporting Web service requests over an HTTP transport. The value of the soapAction property is a URI identifying the intent of the SOAP request. If the soapAction property is specified, it is used by the server component to determine the target of the request. The SOAP specification places no restrictions on the format or specificity of the URI or that it is resolvable. Typically, this property is set to the soapAction value from the WSDL document.
  - **targetService**: This property must be set to the targetService property value that is found in the JMS-style endpoint location URL for the request. This value is used by the server component to determine the port component in the target when dispatching the request.
  - **transportVersion**: This property indicates the version number of the protocol used by the client and server. Set the value to 1 (one).
- If the SOAP request message represents a two-way request, the client component must set the JMS message's replyTo property to specify the queue that is used for the reply message. The JMS message `setJMSReplyTo` method is used for this. It can be beneficial to configure a permanent **replyTo** queue on the client to prevent the client from having to set the JMS message's replyTo property each time a Web service request is made.
- If the SOAP request message represents a one-way request, the client component must not set the JMS message's replyTo property.
- The client component must be prepared to handle a reply message that is a BytesMessage or a TextMessage object, regardless of the type of JMS message used to transmit the SOAP request. The application server component responds with the same type of JMS message that is received from the client, unless the response contains attachments and a BytesMessage object must be used.
- The client component can assume that the reply message correlation ID matches the original request message ID.

Server responsibilities

The server component is responsible for receiving the SOAP request messages and sending the SOAP response messages while adhering to the following protocol constraints:

- The server must be prepared to receive a TextMessage or a BytesMessage. If the request contains attachments, a ByteMessage must be used. The WebSphere product implementation of the server component responds in kind when sending the reply message back to the client, unless the response contains attachments and a BytesMessage is used.
- The server component must process the SOAP request properly to produce an appropriate SOAP reply message.
- The server component must send a reply message back to the client only if the JMS request message's replyTo property is set.
- The server component must set the following properties in the JMS reply message before sending the message to the replyTo queue:
  - **contentType**: See the description for this property in the client responsibilities section in this article.
  - Set the **correlation ID** of the JMS reply message to the message ID of the original JMS request message. This is done by calling the JMS message `setJMSCorrelationID` method.
  - **transportVersion**: This property indicates the version number of the protocol used by the client and server. Set the value to 1 (one).

Example: SOAP request without attachments

The following example displays the results from calling the JMS message `toString` method for a request message without attachments:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     ID:d438eebf04cb124aa25c5821110a134f0000000000000001
JMSTimestamp:     1092110476167
JMSCorrelationID: null
JMSDestination:   topic://NewsGroupTopic?topicSpace=FvtTopicSpace
JMSReplyTo:       null
JMSRedelivered:   false
JMS_IBM_System_MessageID: 6B6765B36943A18C_11000001
transportVersion: 1
JMSXUserID:
targetService: NGConsumerJMS
JMSXAppID: Service Integration Bus
endpointURL: jms:/topic?destination=jms/NewsGroupTopic&connectionFactory;
  =jms/NewsGroupTCF&targetService;=NGConsumerJMS

contentType: text/xml; charset=utf-8
3c736f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f
736368656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c6e
...
```

The following SOAP Version 1.1 example displays the payload from the previous message example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<postMessage><ngName xsi:type="xsd:string">news.current.events</ngName>
<msg xsi:type="xsd:string">This is a sample news item.</msg>
</postMessage>
</soapenv:Body>
</soapenv:Envelope>
```

For SOAP Version 1.2, the `encodingStyle` parameter is not supported, so the example looks similar to the following:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<postMessage><ngName xsi:type="xsd:string">news.current.events</ngName>
<msg xsi:type="xsd:string">This is a sample news item.</msg>
</postMessage>
</soapenv:Body>
</soapenv:Envelope>
```

Example: SOAP request with attachments

The following example displays the results from calling the JMS message `toString` method for a request message with attachments:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  1
JMSExpiration:    1092086312310
JMSPriority:      4
JMSMessageID:     ID:4bb64ed64e7d813d59ba5fec110a134f0000000000000001
JMSTimestamp:     1092086012310
JMSCorrelationID: null
JMSDestination:   queue://Logger_Q
JMSReplyTo:       queue://_Q_6B6765B36943A18C_00000385
JMSRedelivered:   false
```

```
JMS_IBM_System_MessageID: 6B6765B36943A18C_10000001
transportVersion: 1
JMSXUserID:
targetService: WSLoggerJMS
JMSXAppID: Service Integration Bus
endpointURL: jms:/queue?
destination=jms/Logger_Q&connectionFactory=jms/Logger_CF&targetService=WSLoggerJMS
contentType: multipart/related; type="text/xml";
start="<945414389.1092086011970.IBM.WEBSERVICES@myhost1>";
boundary="----=_Part_0_247953397.1092086011970"
0d0a2d2d2d2d2d2d3d5f506172745f305f323437393533333939372e313039323038363031313937330
0d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d380d
...
```

The following displays the payload from the previous message example:

```
Content-Type: multipart/related; type="text/xml";

       start="<945414389.1092086011970.IBM.WEBSERVICES@myhost1>";

       boundary="----=_Part_0_247953397.1092086011970"



------=_Part_0_247953397.1092086011970
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <945414389.1092086011970.IBM.WEBSERVICES@myhost1>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header>
<p499:InternationalizationContext soapenv:mustUnderstand="0"
 xmlns:p499="http://www.ibm.com/webservices/InternationalizationContext">
<Locales>
 <Locale>
  <LanguageCode>en</LanguageCode>
  <CountryCode>US</CountryCode>
 </Locale>
</Locales>
<TimeZoneId>America/Chicago</TimeZoneId>
</p499:InternationalizationContext>
</soapenv:Header>
<soapenv:Body>
 <sendJpegImage/>
</soapenv:Body>
<soapenv:Envelope>
------=_Part_0_247953397.1092086011970
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <jpegImageRequest=81380956150.1092086011880.IBM.WEBSERVICES@myhost1>
<...contents of jpeg image file...>
```

SOAP response

The following example displays the results from calling the JMS message `toString` method for a SOAP reply message:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     null
JMSTimestamp:     0
JMSCorrelationID: ID:cdddb857f078a266eb9a972f110a134f0000000000000001
```

```
JMSDestination:   null
JMSReplyTo:       null
JMSRedelivered:   false
contentType:
  multipart/related;
  type="text/xml";
  start="<961368106530.1092112854745.IBM.WEBSERVICES@yackerjr>";
  boundary="----=_Part_0_1655006754.1092112854745"
0d0a2d2d2d2d2d2d3d5f506172745f305f313635353030363735342e313039323131323835343734
350d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d38
...
```

IBM proprietary JMS endpoint URL syntax (deprecated)

A Java Message Service (JMS) endpoint URL is used to access Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) Web services with the JMS transport. This proprietary URL specifies the Java Message Service (JMS) destination and connection factory, as well as the port component name for the Web service request. This endpoint URL is similar to the HTTP endpoint URL, which specifies the host and port as well as the context root and port component name.

Note: In earlier versions of the application server, an IBM proprietary SOAP over JMS protocol was supported for Java API for XML-based RPC (JAX-RPC) applications. In WebSphere Application Server 7.0, this proprietary SOAP over JMS protocol is now deprecated in favor of an emerging industry standard SOAP over JMS protocol. You can use the IBM proprietary SOAP over JMS protocol with your Java API for XML Web Services (JAX-WS) or JAX-RPC Web services; however, take advantage of the emerging standard SOAP over JMS protocol. If your client application invokes enterprise beans-based Web services that are supported by an earlier version of the WebSphere Application Server, you must continue to use the IBM proprietary SOAP over JMS protocol to access those Web services.

Note: A JMS endpoint URL has the following general form:

`jms:/[`*queue*`|`*topic*`]?<`*property*`>=<`*value*`>&<`*property*`>=<`*value*`>&...`

The URL consists of the `jms:` transport type, followed by either `/queue` or `/topic` to indicate the JMS destination type, followed by the query string containing a list of property and value pairs that are used to specify the JMS endpoint information.

The properties supported in the URL string are described in the following tables:

Destination-related properties (required)

Property name	Description
destination	Specifies the Java Naming and Directory Interface (JNDI) name of the destination queue or topic.
connectionFactory	Specifies the JNDI name of the connection factory.
targetService	Specifies the name of the port component to which the request is dispatched.

JNDI-related properties (optional)

Property name	Description
initialContextFactory	Specifies the name of the initial context factory to use which is mapped to the java.naming.factory.initial property.
jndiProviderURL	Specifies the JNDI provider URL, which is mapped to the java.naming.provider.url property.

JMS-related properties (optional)

Property name	Description
deliveryMode	Indicates whether the request message is persistent or not. The valid values are `1` for nonpersistent and `2` for persistent. The default value is `1`.
timeToLive	Specifies the lifetime, in milliseconds, of the request message. A value of `0` indicates an infinite lifetime.
priority	Specifies the JMS priority associated with the request message. Valid values are between `0` to `9`. The default value is `4`. A value of `0` is the lowest priority and a value of `9` is the highest priority.
replyToDestination	Specifies the JNDI name of a queue to be used to receive reply messages. Using this optional property enables the client to use a permanent queue, rather than a temporary queue, for receiving replies.

If you set values for the deliveryMode, timeToLive, and priority properties on the JMS request, these values are propagated from the JMS request message to the corresponding JMS reply message.

The required properties, destination, connectionFactory, and targetService must be contained in the JMS endpoint URL string. The rest of the properties are optional.

You can set any of the properties on the client Stub object. The various properties can be specified by including them as part of the endpoint URL or you can set these properties programmatically by the client on the Stub object. Properties specified on the client Stub object take precedence over properties that are specified as part of a JMS endpoint URL string.

Using the JMS asynchronous response message listener

Java API for XML-Based Web Services (JAX-WS) includes a Java Message Service (JMS) asynchronous response message listener, which is used to receive responses to asynchronous JAX-WS requests that use the JMS transport. The JMS asynchronous response message listener is used in the application server and application client environments.

Before you begin

JAX-WS provides support for invoking Web service operations asynchronously by using either a callback or a polling model. When the JAX-WS client uses the JMS transport to invoke asynchronous operations, the responses are received by the asynchronous response message listener. To learn how to use the JAX-WS asynchronous client invocation model, read about invoking JAX-WS Web services asynchronously.

About this task

The JMS asynchronous response message listener is used within the Web services client environment to receive incoming asynchronous responses when the client application is using the JMS Transport. The listener requires a connection factory and a queue to function correctly. Begin by configuring the connection factory and queue, and then specify the JNDI names of the connection factory and queue to the listener by setting Java system properties. The environment in which the client is running determines how the system properties are set.

The JMS asynchronous response message listener is started automatically by the Web services client runtime environment when the client invokes the first asynchronous JAX-WS operation using the JMS transport.

The connection factory and the queue configured with the asynchronous response message listener is used for all requests that are invoked within a particular Java process such as for the application server or an application client container. You can share the connection factory among different Java processes. However, you cannot share a queue among Java processes.

1. Determine if you want the JAX-WS client to use the JMS transport mechanism.
2. For each Java process that will use JMS as a transport for asynchronous JAX-WS requests, configure the connection factory and queue that are used by the JMS asynchronous response listener for that process. You can share a connection factory among multiple Java processes, but you cannot share a queue among Java processes.
3. For each Java process, set the com.ibm.websphere.webservices.AsyncReplyQueueName and com.ibm.websphere.webservices.AsyncReplyCFName Java system properties to specify the JNDI names of the queue and connection factory that are used by the JMS asynchronous response message listener for that process.

   If the JNDI name of the queue is the default value, jms/DefaultAsyncReplyQueue, then you do not need to set the AsyncReplyQueueName property. Likewise, if the JNDI name of the connection factory is the default value, jms/DefaultAsyncReplyCF, then you do not need to set the AsyncReplyCFName property as well.

   If your client runs within the application server environment, then set the properties as application server system properties by using the administrative console or the wsadmin command.

   If your client runs within the application client container environment, then you should set the properties by using the –CCD option on the launchClient command line.

Results

Your JAX-WS Web services client is configured to use the JMS asynchronous response message listener to receive asynchronous response messages when using the JMS transport.

Example

Suppose that you have a JAX-WS Web services client that runs in the application client container environment and uses the JMS transport to communicate with the server. Suppose also that the client invokes asynchronous JAX-WS operations. You can create a connection factory with the JNDI name, jms/MyAppCF, and a queue with the JNDI name, jms/MyAppAsyncReplyQueue. When you invoke the client with the launchClient command, specify the JNDI names of the queue and connection factory as illustrated in the following command:

```
launchClient MyAppClient.ear \
   -CCDcom.ibm.websphere.webservices.AsyncReplyQueueName=jms/MyAppReplyQueue \
   -CCDcom.ibm.websphere.webservices.AsyncReplyCFName=jms/MyAppCF \
   <application arguments>
```

Configuring a permanent reply queue for Web services using SOAP over JMS

When using two-way Web service communications using the industry standard SOAP over JMS protocol, you can benefit from configuring a permanent reply queue on a Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) Web services client. The use of a permanent reply queue can improve performance because this reply queue prevents the client from having to create a temporary reply queue each time a Web services request is invoked.

About this task

A permanent reply queue is configured on the Web services client in one of the following ways:

- Specify the optional `replyToName` property in the JMS endpoint URL.
- Set the reply queue programmatically.
  - For a JAX-WS Web services client, set the JNDI name of the reply queue programmatically on the client RequestContext object. Setting the reply queue JNDI name on the RequestContext object affects all subsequent requests that are invoked using that RequestContext object.
  - For a JAX-RPC Web services client, set the JNDI name of the reply queue programmatically on the client Stub or Call object. Setting the reply queue JNDI name as a Stub or a Call property affects all subsequent requests that are invoked using that Stub or Call object.
- Set the reply queue as a Java virtual machine (JVM) system property. Setting the reply queue as a JVM system property affects all of your Web services clients running in the particular JVM. If there are multiple clients running in the same JVM that need to use a different reply queue, then this option does not work. Instead, use one of the other two options.

To set the permanent reply queue using any of these options, only client-side configuration is necessary. There is no configuration necessary for the Web service provider.

Use the typical administrative functions of the JMS messaging provider to create the permanent reply queue prior to configuring the reply queue with the Web services client.

Configure the JNDI name of the permanent reply queue using one of the following ways:

- Specify the optional `replyToName` property in the JMS endpoint URL; for example:

  ```
  jms:jndi:jms/MyRequestQueue&jndiConnectionFactoryName=jms/MyCF&replyToName=jms/MyReplyQueue
  ```

- Set the reply queue programmatically on the client.

  The value of the property is a String and represents the JNDI name of the reply queue.

  - For JAX-WS Web services clients, set the com.ibm.wsspi.webservices.Constants.JMS_REPLY_QUEUE_JNDI_NAME property on the client JAX-WS RequestContext object; for example:

    ```
    ((BindingProvider) port).getRequestContext().put
    (com.ibm.wsspi.webservices.Constants.JMS_REPLY_QUEUE_JNDI_NAME, "jms/MyReplyQueue");
    ```

  - For JAX-RPC Web services clients, set the com.ibm.wsspi.webservices.Constants.JMS_REPLY_QUEUE_JNDI_NAME property on the client JAX-RPC Stub or Call object; for example:

    ```
    ((javax.xml.rpc.Stub) stub)._setProperty(com.ibm.wsspi.webservices.Constants.JMS_REPLY_QUEUE_JNDI_NAME,
    "jms/MyReplyQueue");
    ```

- Set the JNDI name of the reply queue as a JVM system property.
  - For a Java client invocation, enter the following code at a command prompt:

    ```
    java -Dcom.ibm.websphere.webservices.JMSReplyQueueJndiName=jms/MyReplyQueue
    ```

  - For a JVM running on the application server, perform the following actions:
    - Set a JVM system property using the administrative console for the application server that runs the Web service client application.

      To set custom properties, log on to the administrative console, and navigate to the Java virtual machine custom properties panel.

      1. Click **Servers > Server Types > WebSphere application servers > *server_name*> Java and Process Management > Process Definition > Java Virtual Machine > Custom Properties> New**
      2. Set the **Name** property to: com.ibm.websphere.webservices.JMSReplyQueueJndiName
      3. Set the **Value** property to: jms/Permanent_Q
      4. Click OK to save your changes.

5. Click **Synchronize changes with Nodes** and click **Save**.
6. Restart the application server.

Results

Your Web services client can now receive SOAP overJMS messages from a permanent reply queue.

Configuring a permanent replyTo queue for JAX-RPC Web services using SOAP over JMS (deprecated)

When using two-way Web service communications using the IBM proprietary SOAP over JMS transport, you can benefit from configuring a permanent replyTo queue on Java API for XML-based RPC (JAX-RPC) Web services client to prevent the client from having to create a temporary reply queue each time a Web service request is made.

About this task

Note: In WebSphere Application Server 7.0, the IBM proprietary SOAP over JMS protocol is now deprecated in favor of the emerging industry standard protocol. You can use the IBM proprietary SOAP over JMS protocol with your Java API for XML Web Services (JAX-WS) or JAX-RPC Web services, however, you are encouraged to take advantage of the SOAP over JMS protocol standard. This task describes configuring a permanent replyTo queue when using the IBM proprietary SOAP over JMS transport. To learn more about the SOAP over JMS standard, see the using SOAP over JMS to transport Web services documentation.

A permanent replyTo queue is configured on the Web services client in one of the following ways:

- Specify the optional `replyToDestination` property in the JMS endpoint URL.
- Set the replyTo queue programmatically on the client JAX-RPC Stub or Call object. Setting the replyTo queue as a Stub or a Call property affects all requests that are invoked using that Stub or Call object.
- Set the replyTo queue as a Java virtual machine (JVM) system property. Setting the replyTo queue as a JVM system property affects all of your SOAP over JMS clients running in the particular JVM. If there are multiple clients in the same application that need to use a different replyTO queue, then the best option is to set the property programmatically.

Note: To set the permanent replyTo queue using any of these options, only client-side configuration is necessary. There is no configuration necessary on the Web service provider side.

- Specify the optional `replyToDestination` property in the JMS endpoint URL.

  ```
  jms:/queue?destination=jms/MyRequestQueue&connectionFactory=
  jms/MyCF&replyToDestination=jms/MyReplyQueue&targetService=MyService
  ```

- Set the replyTo queue programmatically on the client JAX-RPC Stub or Call object.

  The client uses the JAX-RPC Stub or Call object to invoke the Web service.

  ```
  ((javax.xml.rpc.Stub)stub)._setProperty(com.ibm.wsspi.webservices.Consta
  nts.JMS_REPLY_QUEUE_JNDI_NAME,
  "jms/Permanent_Q");
  ```

  The value of the property is a String.

- Set the replyTo queue as a JVM system property.
  - For a Java client invocation, enter at a command prompt:

    ```
    java -Dcom.ibm.websphere.webservices.JMSReplyQueueJndiName=jms/Permanent_Q
    ```

  - For a JVM running on the application server:
    - Set a JVM system property using the administrative console for the application server which runs the Web service client application.

      To set custom properties, connect to the administrative console and navigate to the Java virtual machine custom properties panel.

1. Click **Servers > Server Types > WebSphere application servers > *server_name*> Java and Process Management > Process Definition > Java Virtual Machine > Custom Properties> New**
2. Set the **Name** property to: `com.ibm.websphere.webservices.JMSReplyQueueJndiName`
3. Set the **Value** property to: `jms/Permanent_Q`
4. Click OK to save your changes.
5. Click **Synchronize changes with Nodes** and click **Save**.
6. Restart the application server.

Results

Your Web services client can now receive IBM proprietary SOAP over JMS messages from a permanent replyTo queue.

Invoking Web service requests transactionally using SOAP over JMS transport

Use the enableTransactionalOneWay property to ensure that one-way and two-way asynchronous Web service requests using the industry standard SOAP over JMS transport will be sent to the destination queue or topic transactionally.

About this task

When using JMS to transport Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) Web service requests, the default behavior is for the SOAP message to be added to the destination queue or topic non-transactionally or outside of the client application's transaction. Adding the SOAP message to the destination queue or topic is done outside of the transaction to avoid synchronization problems that can occur with two-way synchronous Web service requests. However, you can choose to enable one-way and two-way asynchronous requests to be processed as part of a transaction. You can use the enableTransactionalOneWay property to ensure that one-way and two-way asynchronous Web service requests that use the JMS transport are sent to the destination queue or topic transactionally. When the client application invokes the Web service request, the resulting SOAP request message is added to the destination queue or topic as part of the client application's transaction.

Use one of the following ways to enable the enableTransactionalOneWay property.

- Set the enableTransactionalOneWay property programmatically. The value of the property is a Boolean.
  - For JAX-WS clients, set the property on the client JAX-WS RequestContext object. For example:

    ```
    ((BindingProvider) port).getRequestContext().put
    (com.ibm.websphere.webservices.Constants.ENABLE_TRAN_ONEWAY,
    new Boolean(true));
    ```

  - For JAX-RPC clients, set the property on the client JAX-RPC Stub or Call object. For example:

    ```
    stub._setProperty(com.ibm.websphere.webservices.Constants.ENABLE_TRAN_ONEWAY,
    new Boolean(true));
    ```

- For JAX-RPC clients, set the `enableTransactionalOneWay` property as a custom property in the `ibm-webservicesclient-bnd.xmi` deployment descriptor file by using the wsadmin command.

  For more information about the wsadmin tool options, review Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands.

  Use the `$AdminApp` object along with the `-WebServicesClientCustomProperty` option to set the value of the property within the client binding file, `ibm-webservicesclient-bnd.xmi`. The value of the custom property, `enableTransactionalOneWay` , is either `true` or `false`.
  - Using Jacl:

    ```
    $AdminApp edit MyApplication {-WebServicesClientCustomProperty {{MyEJBJar.jar MyEJB
    service/MyServiceRef MyPort enableTransactionalOneWay true}}}
    ```

  - Using Jython:

```
AdminApp.edit('MyApplication', ['-WebServicesClientCustomProperty', [['MyEJBJar.jar',
'MyEJB', 'service/MyServiceRef ', 'MyPort', 'enableTransactionalOneWay', 'true']]])
```

Results

You have a Web services client application that is configured to invoke one-way and two-way asynchronous requests transactionally when using the JMS transport.

What to do next

After you have enabled the `enableTransactionalOneWay` property, run the client application.

Invoking one-way JAX-RPC Web service requests transactionally using the JMS transport (deprecated)

Use the enableTransactionalOneWay property to ensure that one-way JAX-RPC Web service requests using the IBM proprietary JMS transport will be sent to the destination queue or topic transactionally.

About this task

Note: In WebSphere Application Server 7.0, the IBM proprietary SOAP over JMS protocol is now deprecated in favor of the emerging industry standard protocol. You can use the IBM proprietary SOAP over JMS protocol with your Java API for XML Web Services (JAX-WS) or JAX-RPC Web services, however, you are encouraged to take advantage of the SOAP over JMS protocol standard. This task describes configuring a permanent replyTo queue when using the IBM proprietary SOAP over JMS transport. To learn more about the SOAP over JMS standard, see the using SOAP over JMS to transport Web services documentation.

When using JMS to transport Web service requests, the default behavior is for the SOAP message to be added to the destination queue or topic non-transactionally or outside of the client application's transaction. Adding the SOAP message to the destination queue or topic is done outside of the transaction to avoid synchronization problems that can occur with two-way Web service requests. However, you can choose to enable one-way requests to be processed as part of the transaction. The `enableTransactionalOneWay` property can be used to ensure that one-way Web service requests that use the JMS transport will be sent to the destination queue or topic transactionally. When the client application invokes the one-way Web service request, the resulting SOAP request message is added to the destination queue or topic as part of the client application's transaction.

Use one of the following ways to enable the `enableTransactionalOneWay` property.

- Set the `enableTransactionalOneWay` property programmatically on the client JAX-RPC Stub or Call object.

  When using a static Stub to invoke the Web service operation, set the `enableTransactionalOneWay` property on the Stub object before invoking the Web service method. When using a Call object to invoke the Web service operation, set the `enableTransactionalOneWay` property on the Call object before invoking the `invokeOneWay()` method.

  ```
  Service service = /* Obtain the desired service */
  MyStub stub = service.getPort();

  /* Set enableTransactionalOneWay property on Stub */
  stub._setProperty(com.ibm.websphere.webservices.Constants.ENABLE_TRAN_ONEWAY, new Boolean(true));

  /* Invoke the one-way operation */
  stub.myOneWayOperation("Parm1");
  ```

  The value of the property is Boolean.
- Set the `enableTransactionalOneWay` property as a custom property in the `ibm-webservicesclient-bnd.xmi` deployment descriptor file by using the wsadmin command.

For more information about the wsadmin tool options, review Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands.

Use the `$AdminApp` object along with the `-WebServicesClientCustomProperty` option to set the value of the property within the client binding file, `ibm-webservicesclient-bnd.xmi`. The value of the custom property, `enableTransactionalOneWay` , is either `true` or `false`.

– Using Jacl:

```
$AdminApp edit MyApplication {-WebServicesClientCustomProperty {{MyEJBJar.jar MyEJB
service/MyServiceRef MyPort enableTransactionalOneWay true}}}
```

– Using Jython:

```
 AdminApp.edit('MyApplication', ['-WebServicesClientCustomProperty', [['MyEJBJar.jar',
'MyEJB', 'service/MyServiceRef ', 'MyPort', 'enableTransactionalOneWay', 'true']]])
```

Results

You have a Web service client application that is configured to invoke one-way requests transactionally while using the JMS transport.

What to do next

After you have enabled the `enableTransactionalOneWay` property, run the client application.

Developing applications that use Web Services Addressing

Web Services Addressing (WS-Addressing) aids interoperability between Web services by defining a standard way to address Web services and to provide addressing information in messages. This task describes the steps that are required to create a Web service that is accessed using a WS-Addressing endpoint reference.

About this task

Perform these tasks if you are using endpoint references in your Web service application logic to address Web service endpoints, or if you are creating a Web service that complies with the WS-Addressing interoperability protocol.

- To perform the basic WS-Addressing development activities that are required by Web services developers, such as creating a Web service that is referenced by an endpoint reference, refer to “Using the Web Services Addressing APIs: Creating an application that uses endpoint references” on page 682.
- To perform more advanced WS-Addressing functions, such as setting or retrieving message-addressing properties, refer to “Using the IBM proprietary Web Services Addressing SPIs: Performing more advanced Web Service Addressing tasks” on page 699.
- To configure a service or a client to use the WS-Addressing support, refer to “Enabling Web Services Addressing support for JAX-RPC applications” on page 712.

Web Services Addressing support

The Web Services Addressing (WS-Addressing) support in this product provides the environment for Web services that use the Worldwide Web Consortium (W3C) WS-Addressing specifications. This family of specifications provide transport-neutral mechanisms to address Web services and to facilitate end-to-end addressing.

Note: This release provides support for the Java API for XML-Based Web Services (JAX-WS) 2.1 API, which you can use to create portable applications that use WS-Addressing. Existing JAX-WS applications will work as before.

You do not normally need to be aware of the underlying WS-Addressing support because WebSphere Application Server ensures that your Web service applications are WS-Addressing compliant when required. Read this topic only if you need to use the WS-Addressing support directly. For example, if you have one of the following roles:

- A Web service developer who needs to use the WS-Addressing application programming interfaces (APIs) to create endpoint references within an application, and then use these references to target Web service resource instances.
- A system programmer who needs to use the IBM proprietary WS-Addressing system programming interfaces (SPIs) to perform more advanced WS-Addressing operations, such as specifying message-addressing properties on Web services messages.
- An administrator who is configuring policy sets for JAX-WS applications.

The WS-Addressing support for developers consists of two sets of programming interfaces: the JAX-WS 2.1 standard interfaces, and the IBM proprietary implementation of the WS-Addressing specification.

Features of the JAX-WS 2.1 WS-Addressing support

This product provides support for the JAX-WS 2.1 WS-Addressing APIs, which you can use to perform basic addressing functions such as creating an endpoint reference, enabling WS-Addressing support, and specifying the action URIs that are associated with the WSDL operations of the Web service. Use these APIs if you want to perform simple WS-Addressing functions and create JAX-WS applications that are portable.

The JAX-WS 2.1 WS-Addressing APIs provide the following features for core WS-Addressing application development:

- Java representations of WS-Addressing endpoint references.
  - You can create Java endpoint reference instances for the application endpoint, or other endpoints in the same application, at run time. You do not have to specify the URI of the endpoint reference.
  - You can create Java endpoint reference instances for endpoints in other applications by specifying the URI of the endpoint reference.
  - On services, you can use annotations to specify whether WS-Addressing support is enabled and whether it is required.
  - On clients, you can use features to specify whether WS-Addressing support is enabled and whether it is required.
  - You can configure client proxy or Dispatch objects using endpoint references.
- Java support for endpoint references that represent Web Services Resource (WS-Resource) instances.
  - You can associate reference parameters with an endpoint reference at the time of its creation, to correlate it with a particular resource instance.
  - In targeted Web services, you can extract the reference parameters of an incoming message, so that the Web service can route the message to the appropriate WS-Resource instance.

Features of the IBM proprietary WS-Addressing support

This product provides an IBM proprietary implementation of the WS-Addressing specification, which you can use with JAX-RPC applications as well as JAX-WS applications, to perform more advanced functions such as creating endpoint references that represent highly available objects, or directly setting message addressing properties in the SOAP header. Use these APIs and SPIs if you want to create JAX-RPC applications that use addressing, or you want to perform more advanced functions that are not possible with the JAX-WS 2.1 APIs.

The IBM proprietary API provides the following basic features:

- You can easily create Java endpoint reference instances to represent any endpoint in the server, based on the deployment environment of the application. You do not have to specify the URI of the endpoint reference. Additionally, endpoint references can represent highly available or workload-managed objects.
- You can configure client JAX-WS BindingProvider request context objects, or JAX-RPC Stub or Call objects, with a WS-Addressing endpoint reference. Future invocations on these objects are targeted at the endpoint that is represented by the endpoint reference. The invocations also automatically conform to the WS-Addressing specification (namespace) that is associated with that endpoint reference.

The IBM proprietary WS-Addressing SPIs provide support for extended WS-Addressing system development using the following features:

- Reasoning and manipulation of endpoint references beyond what is available at the application programming level.
  - You can manipulate the contents of the endpoint reference, as specified by the WS-Addressing specification.
  - You can associate a WS-Addressing namespace, and therefore specification behavior, with an endpoint reference.
- Java representations of the WS-Addressing message-addressing properties.
  - You can specify WS-Addressing message-addressing properties for outbound Web service messages. In the targeted Web service, you can extract message addressing properties from inbound Web service messages.
  - You can specify the WS-Addressing namespace of an outbound WS-Addressing message, although in most cases the namespace is automatically derived based on the target endpoint reference. In a targeted Web service, you can acquire the WS-Addressing namespace of an incoming message.

Support for WS-Addressing specifications and interoperability

By default, this product supports the W3C WS-Addressing 1.0 Core and SOAP Binding specifications that are identified by the http://www.w3.org/2005/08/addressing namespace. Unless otherwise stated, WS-Addressing semantics that are described in this documentation refer to these specifications.

For interoperability, other levels of the WS-Addressing specification are supported in this version of the product; in particular, the WS-Addressing W3C submission with the namespace http://schemas.xmlsoap.org/ws/2004/08/addressing.

For JAX-WS applications, this product supports the WS-Addressing metadata specification identified by the http://www.w3.org/2007/05/addressing/metadata namespace. This specification supersedes the WS-Addressing Web Services Description Language (WSDL) binding specification identified by the http://www.w3.org/2006/05/addressing/wsdl namespace.

In addition, this product supports the following features from the WS-Addressing WSDL binding specification:

- The wsaw:UsingAddressing extensibility element, on the WSDL Binding element only. The supported namespaces for this element are the http://www.w3.org/2006/05/addressing/wsdl namespace and the http://www.w3.org/2006/02/addressing/wsdl namespace (deprecated).
- The wsaw:Action extensibility element. The supported namespaces for this element are the http://www.w3.org/2006/05/addressing/wsdl namespace, the http://www.w3.org/2006/02/addressing/wsdl namespace (deprecated), and the http://schemas.xmlsoap.org/ws/2004/08/addressing namespace.

Web Services Addressing overview

Web Services Addressing (WS-Addressing) is a Worldwide Web Consortium (W3C) specification that aids interoperability between Web services by defining a standard way to address Web services and provide addressing information in messages. The WS-Addressing specification introduces two primary concepts:

endpoint references, and message-addressing properties. This topic contains an overview of each concept; for further details, refer to the WS-Addressing specifications.

Endpoint references

Endpoint references provide a standard mechanism to encapsulate information about specific endpoints. Endpoint references can be propagated to other parties and then used to target the Web service endpoint that they represent. The following table summarizes the information model for endpoint references.

Table 11. Information model for endpoint references

Abstract Property name	Property type	Multiplicity	Description
[address]	xs:anyURI	1..1	The absolute URI that specifies the address of the endpoint.
[reference parameters]*	xs:any	0..unbounded	Namespace qualified element information items that are required to interact with the endpoint.
[metadata]	xs:any	0..unbounded	Description of the behavior, policies and capabilities of the endpoint.

The following prefix and corresponding namespace is used in the previous table.

Table 12. Prefix and corresponding namespace

Prefix	Namespace
xs	http://www.w3.org/2001/XMLSchema

The following XML fragment illustrates an endpoint reference. This element references the endpoint at the URI http://example.com/fabrikam/acct, has metadata specifying the interface to which the endpoint reference refers, and has application-defined reference parameters of the http://example.com/fabrikam namespace.

```
<wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/addressing"
                       xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                       xmlns:fabrikam="http://example.com/fabrikam"
                       xmlns:wsdli="http://www.w3.org/2005/08/wsdl-instance"
                       wsdli:wsdlLocation="http://example.com/fabrikam
                       http://example.com/fabrikam/fabrikam.wsdl">
    <wsa:Address>http://example.com/fabrikam/acct</wsa:Address>
    <wsa:Metadata>
        <wsaw:InterfaceName>fabrikam:Inventory</wsaw:InterfaceName>
    </wsa:Metadata>
    <wsa:ReferenceParameters>
        <fabrikam:CustomerKey>123456789</fabrikam:CustomerKey>
        <fabrikam:ShoppingCart>ABCDEFG</fabrikam:ShoppingCart>
    </wsa:ReferenceParameters>
</wsa:EndpointReference>
```

Message-addressing properties

Message addressing properties (MAPs) are a set of well defined WS-Addressing properties that can be represented as elements in SOAP headers and provide a standard way of conveying information, such as the endpoint to which message replies should be directed, or information about the relationship that the message has with other messages. The MAPs that are defined by the WS-Addressing specification are summarized in the following table.

Table 13. Message-addressing properties defined by the WS-Addressing specification

Abstract WS-Addressing MAP name	MAP content type	Multiplicity	Description
[action]	xs:anyURI	1..1	An absolute URI that uniquely identifies the semantics of the message. This proprety corresponds to the [address] property of the endpoint reference to which the message is addressed. This value is required.
[destination]	xs:anyURI	1..1	The absolute URI that specifies the address of the intended receiver of this message. This value is optional because, if not present, it defaults to the anonymous URI that is defined in the specification, indicating that the address is defined by the underpinning protocol.
[reference parameters]*	xs:any	0..unbounded	Correspond to the [reference parameters] property of the endpoint reference to which the message is addressed. This value is optional.
[source endpoint]	EndpointReference	0..1	A reference to the endpoint from which the message originated. This value is optional.
[reply endpoint]	EndpointReference	0..1	An endpoint reference for the intended receiver of replies to this message. This value is optional.
[fault endpoint]	EndpointReference	0..1	An endpoint reference for the intended receiver of faults relating to this message. This value is optional.
[relationship]*	xs:anyURI plus optional attribute of type xs:anyURI	0..unbounded	A pair of values that indicate how this message relates to another message. The content of this element conveys the [message id] of the related message. An optional attribute conveys the relationship type. This value is optional.
[message id]	xs:anyURI		An absolute URI that uniquely identifies the message. This value is optional.

The abstract names in the previous tables are used to refer to the MAPs throughout this documentation.

The following example of a SOAP message contains WS-Addressing MAPs:

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
            xmlns:wsa="http://www.w3.org/2005/08/addressing"
            xmlns:fabrikam="http://example.com/fabrikam">
    <S:Header>
        ...
        <wsa:To>http://example.com/fabrikam/acct</wsa:To>
                  <wsa:ReplyTo>
             <wsa:Address> http://example.com/fabrikam/acct</wsa:address>
        </wsa:ReplyTo>
        <wsa:Action>...</wsa:Action>
        <fabrikam:CustomerKey wsa:IsReferenceParameter='true'>123456789</fabrikam:CustomerKey>
        <fabrikam:ShoppingCart wsa:IsReferenceParameter='true'>ABCDEFG</fabrikam:ShoppingCart>
        ...
    </S:Header>
    <S:Body>
    ...
    </S:Body>
</S:Envelope>
```

Web Services Addressing message exchange patterns:

The Worldwide Web Consortium (W3C) Web Services Addressing (WS-Addressing) specification explicitly defines the WS-Addressing core properties for the message exchange patterns (MEPs) that are defined by WSDL 1.0. These MEPs are summarized in this topic, illustrating the mandatory WS-Addressing properties for each pattern.

One-way MEP

This straightforward one-way message is defined in WSDL 1.0 as an input-only operation. The WSDL fragment for this operation has the following form:

```
<operation name="myOperation">
    <input message="tns:myInputMessage"/>
</operation>
```

The following WS-Addressing message addressing properties (MAPs) are automatically added to the message header of a one-way WS-Addressing input message by the client application server run time, to ensure compliance with the WS-Addressing specification.

Note: You can override these values using the IBM proprietary WS-Addressing system programming interfaces (SPIs).

Table 14.

Abstract WS-Addressing MAP name	Default value for a one-way input message
[action]	The WS-Addressing action that is generated in accordance with the version of the WS-Addressing specification that is in use.
[reply endpoint]	The WS-Addressing reply endpoint indicating that no replies are expected to this input message. The value of this MAP depends on the version of the WS-Addressing specification that is in use.
[message id]	A uniquely generated message identifier. Although not mandated by the specification, the WebSphere Application Server run time automatically sets this value.

Although the WSDL operation for this message exchange does not specify any responses, related messages can be sent as part of other message exchanges. In particular, applications can use the WS-Addressing reply endpoint or fault endpoint MAPs to indicate to the target of a one-way message where to send related messages. To propagate a reply endpoint or fault endpoint, associate the appropriate property with the JAX-WS BindingProvider object's request context, or with the JAX-RPC Stub or Call object, as described in "Specifying and acquiring message-addressing properties using the IBM proprietary Web Services Addressing SPIs" on page 700, to override the defaults.

Two-way request-response

This is a request-response MEP as defined in WSDL 1.1. The response part of the operation might be defined as an output message, or a fault message, or both. The following WSDL code extracts show the various forms of definition for such an operation:

```
<operation name="myOperation">
   <input message="tns:myInputMessage"/>
   <output message="tns:myOutputMessage"/>
   <fault="tns:myFaultMessage"/>
</operation>
```

```
<operation name="myOperation">
   <input message="tns:myInputMessage"/>
   <output message="tns:myOutputMessage"/>
</operation>
```

```
<operation name="myOperation">
   <input message="tns:myInputMessage"/>
   <fault="tns:myFaultMessage"/>
</operation>
```

The application server client run time ensures that the SOAP header of the outgoing request message contains the relevant WS-Addressing message information headers. The calling application does not have to set the WS-Addressing headers. A response is expected, therefore you must specify a reply endpoint or fault endpoint in the request message.

Note: In the 2005/08 specification, an unspecified reply endpoint is valid because it defaults to an endpoint reference that contains the anonymous URI.

The following table summarizes the MAPs that the product sets by default on a Web service request that uses WS-Addressing. You can override or specify other MAPs using the IBM proprietary WS-Addressing SPIs.

Table 15.

Abstract WS-Addressing MAP name	**Default value for the input message of a request-response operation**
[action]	The WS-Addressing action that is generated in accordance with the version of the WS-Addressing specification that is in use.
[message id]	A uniquely generated message identifier.

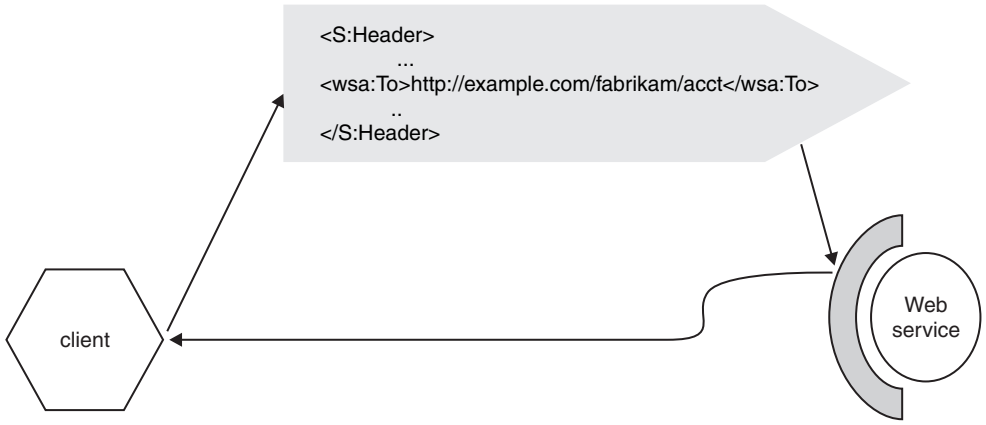
The following table summarizes the MAPs that are set by default by the product on a WS-Addressing response or fault message.

Table 16.

Abstract WS-Addressing MAP name	**Default value for the input message of a request-response operation**
[action]	The WS-Addressing action that is generated in accordance with the version of the WS-Addressing specification that is in use.
[relationship]*	A relationship set containing a reply relationship to the message id that is passed in the request message.
[message id]	A uniquely generated message identifier; although not mandated by the specification, the application server run time automatically sets this property.

Synchronous request-response

By default, if you do not use the IBM proprietary WS-Addressing SPI to set the reply endpoint or fault endpoint, the response part of a two-way message is returned according to the underlying protocol in use. In particular, in the case of an HTTP request, the response is returned synchronously in the HTTP response.



For JAX-WS synchronous invocations, if you set the reply endpoint or the fault endpoint, the endpoint reference that you set must use the anonymous URI. If the endpoint reference does not use the anonymous URI, a javax.xml.ws.WebServiceException exception is thrown. Although the endpoint reference uses the anonymous URI, you can use reference parameters within the endpoint reference to target the reply or fault endpoint.

For JAX-WS applications, you can specify a synchronous message exchange pattern by applying and configuring a WS-Addressing policy type. This exchange pattern is particularly useful in the following scenarios:

- You do not have WS-Security enabled, or have not used an assembly tool to specify that the ReplyTo and FaultTo elements of the SOAP message should be signed. In this situation, it is possible for a JAX-WS endpoint to be used to send messages to a third party, potentially taking part in a Denial of Service attack. To prevent such attacks, specify the synchronous message exchange pattern, and enable WS-Policy so that clients are aware of this requirement.
- A JAX-WS client is communicating through a NAT device. URIs in the ReplyTo or FaultTo elements of the SOAP message cannot be routed through such a device. In this situation, the client must use the anonymous URI defined by the WS-Addressing specification, and a synchronous message exchange pattern. To ensure that the client conforms to these requirements even if the server uses WS-Policy to request a non-anonymous URI in the ReplyTo element, specify the synchronous message exchange pattern on the client.

You can ensure that servers or clients are aware of the requirement for synchronous messaging by enabling WS-Policy.

Asynchronous request-response

The JAX-RPC 1.0 programming model does not allow for asynchronous replies or faults to a two-way request-response operation.

Responses to, or faults generated from, requests that are directed at endpoints hosted on WebSphere Application Server are targeted at the reply endpoint or fault endpoint, in accordance with the WS-Addressing specification. The connection with the requesting client will be closed with an HTTP 202 response.

For JAX-WS asynchronous invocations, the reply endpoint is generated automatically for use by the JAX-WS implementation. If you attempt to set either a reply endpoint or a fault endpoint , a javax.xml.ws.WebServiceException exception is thrown.

For JAX-WS applications, you can specify an asynchronous message exchange pattern by applying and configuring a WS-Addressing policy type. This exchange pattern is particularly useful if a JAX-WS endpoint has a long-running invocation time. Client and server resources are used to keep the connection open, but this use of resource might be impractical if the service takes a long time to respond. You can ensure that clients are aware of the requirement for asynchronous messaging by enabling WS-Policy.

Web Services Addressing version interoperability

The Web Services Addressing (WS-Addressing) support in this product can interoperate with various versions of the WS-Addressing specification.

Table 17. Supported set of WS-Addressing versions

Associated namespace	Specification download location	Details
http://www.w3.org/2005/08/addressing	http://www.w3.org/2002/ws/addr/	W3C Candidate Recommendation (CR) versions of the WS-Addressing core and SOAP specifications. These specifications are sometimes referred to collectively as the 2005/08 version of WS-Addressing.

Table 17. Supported set of WS-Addressing versions (continued)

Associated namespace	Specification download location	Details
http://www.w3.org/2007/05/addressing/metadata	http://www.w3.org/2002/ws/addr/	W3C final version of the WS-Addressing metadata specification. This specification defines WS-Addressing WSDL extensions and WS-Policy assertions. For JAX-WS applications, this specification supersedes the http://www.w3.org/2006/05/addressing/wsdl specification.
http://www.w3.org/2006/05/addressing/wsdl	http://www.w3.org/2002/ws/addr/	W3C Candidate Recommendation (CR) version of the WS-Addressing WSDL specification. This is the default namespace used by this product for the WSDL parts of the WS-Addressing specification, for JAX-RPC applications. For JAX-WS applications, this specification is superseded by the http://www.w3.org/2007/05/addressing/metadata specification.
http://www.w3.org/2006/02/addressing/wsdl	http://www.w3.org/2002/ws/addr/	W3C Last Call (LC) version of the WS-Addressing WSDL specification. Support for this namespace is deprecated.
http://schemas.xmlsoap.org/ws/2004/08/addressing	http://www.w3.org/Submission/ws-addressing/	W3C WS-Addressing Submission specification This specification is sometimes referred to as the 2004/08 specification. It combines the core, SOAP and WSDL aspects of WS-Addressing in a single specification.

This version of the product interoperates with each of the WS-Addressing specifications that are defined in the previous table. This interoperability results in the following behavior:

- Incoming Web service messages that contain WS-Addressing message addressing properties (MAPs) are appropriately bound to SOAP, and WS-Addressing SOAP elements are appropriately deserialized to their WS-Addressing programming model representations according to the namespace in use.
- WS-Addressing programming model artifacts are appropriately serialized into SOAP elements, and the MAPs are bound to SOAP according to the namespace in use.
- Differing WS-Addressing semantics are adhered to, according to the WS-Addressing version currently in use.

Determining the WS-Addressing namespace of inbound messages

The WS-Addressing namespace of incoming Web service messages is the namespace of the first WS-Addressing [action] MAP that is found. The runtime looks for an [action] MAP of the default namespace prior to searching for other namespaces on the inbound message, in an undefined order. The

namespace of the WS-Addressing core specification in use is available to the target endpoint through the message context.

Determining the WS-Addressing namespace of outbound messages

WS-Addressing messages that are issued from this version of the product adopt the namespace that is associated with the destination endpoint reference. If this namespace is unknown, the message adopts the default WS-Addressing namespace.

This product provides a proprietary system programming interface (SPI) to change the namespace that is associated with an endpoint reference to any namespace in the supported set.

The WS-Addressing specification to use

Note: In most cases, use the default WS-Addressing specification that is supported by the product. You do not need to perform any additional actions to use this specification. The following list gives examples of occasions where you must override the default namespace:

- When interoperating with an endpoint that does not support the default namespace, for example, an earlier version of the product.
- When a namespace other than the default is required. For example, when implementing a specification that uses a level of WS-Addressing other than the default.

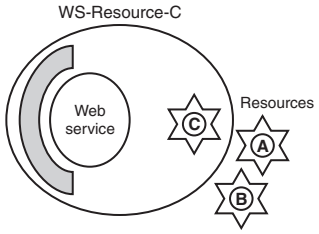
The W3C Last Call (LC) version of the WS-Addressing WSDL specification is deprecated. Use this specification only when you are interoperating with WebSphere Application Server 6.1 nodes that do not have fix pack V6.1.0.2 or later. Otherwise, use the W3C Candidate Recommendation version of the specification, or for JAX-WS applications, the WS-Addressing metadata specification.

Web Services Addressing application programming model

The Web Services Addressing (WS-Addressing) specification defines an endpoint reference that is represented in Extensible Markup Language (XML) by an EndpointReferenceType object that encapsulates information about the endpoint address as well as additional contextual information associated with the endpoint. Some services might be addressable using a simple URI address, as is most typical in Web services. Other services might require the use of an endpoint reference to address them, so that the additional contextual information associated with the endpoint is present in messages sent to the endpoint.

Examples of services that use WS-Addressing endpoint references include Web Services Resources and Web Services Notification message producers and message consumers, all of which have the notion of stateful resources associated with their endpoints. In these cases the endpoint reference not only contains the service address but also some data that is used to select the specific stateful resource instance for use in the processing of a Web services message.

A WS-Resource is defined as the combination of a resource and a Web service through which the resource is accessed. The following figure illustrates a Web service, at http://www.example.com/service, and three resources, A, B, and C, which are accessed through the Web service. Three WS-Resources are therefore illustrated in the figure:



A WS-Resource is referenced by a WS-Addressing endpoint reference that uniquely identifies the WS-Resource, typically by containing an identifier of the resource component of the WS-Resource inside the EndpointReference ReferenceParameter element. In the previous example, WS-Resource-C is the combination of the Web service and the resource that is identified by C, and a reference to WS-Resource-C might be as follows:

```
<wsa:EndpointReference>
    <wsa:Address>
        http://www.example.com/service
    </wsa:Address>
    <wsa:ReferenceParameters>
        <tns:SomeDisambiguatorElement>C</tns:SomeDisambiguatorElement>
    </wsa:ReferenceParameters>
    ...
</wsa:EndpointReference>
```

The WS-Addressing APIs provide the appropriate interfaces for implementing the previous pattern.

Web Services Addressing annotations

The WS-Addressing specification provides transport-neutral mechanisms to address Web services and to facilitate end-to-end addressing. If you have a JAX-WS 2.1 application you can use Java annotations in your code to specify WS-Addressing behavior at run time. Use WS-Addressing annotations to specify the actions that are associated with Web service operations, and also to enable or disable WS-Addressing support.

Note: This release provides support for the WS-Addressing annotations specified by the JAX-WS 2.1 standard. Use these annotations in the Java code for a Web service, to enable WS-Addressing support, to specify whether WS-Addressing information is required in incoming messages, and to specify actions to be associated with a Web service operation or fault response.

The following WS-Addressing annotations are supported in this product. These annotations are defined in the JAX-WS 2.1 specification unless otherwise stated.

javax.xml.ws.Action
: Specifies the action that is associated with a Web service operation. When you use this annotation with a particular method, and generate the corresponding WSDL document, the WS-Addressing Action extension attribute is added to the input and output elements of the WSDL operation that corresponds to that method. For this attribute to be added to the WSDL operation you must also specify the Addressing annotation on the implementation class. If you do not want to use the Addressing annotation you can supply your own WSDL document with the Action attribute already defined.

javax.xml.ws.FaultAction
: Specifies the action that is added to a fault response. When you use this annotation with a particular method, the WS-Addressing FaultAction extension attribute is added to the fault element of the WSDL operation that corresponds to that method. For this attribute to be added to the WSDL operation you must also specify the Addressing annotation on the implementation class. If you do not want to use the Addressing annotation you can supply your own WSDL document with the Action attribute already defined. This annotation must be contained within an Action annotation.

javax.xml.ws.soap.Addressing
: Specifies that this service wants to enable WS-Addressing support. You can use this annotation only on the service implementation bean; you cannot use it on the service endpoint interface.

com.ibm.websphere.wsaddressing.jaxws21.SubmissionAddressing
: This annotation is part of the IBM implementation of the JAX-WS 2.1 specification. This annotation specifies that this service wants to enable WS-Addressing support for the 2004/08 WS-Addressing specification. You can use this annotation only on the service implementation bean; you cannot use it on the service endpoint interface.

The following example code uses the Action annotation to define the invoke operation to be invoked (input), and the action that is added to the response message (output). The example also uses the FaultAction annotation to specify the action that is added to a response message if a fault occurs:

```
@WebService(name = "Calculator")
public interface Calculator {
   ...
    @Action(
      input="http://calculator.com/inputAction",
      output="http://calculator.com/outputAction",
      fault = { @FaultAction(className=AddNumbersException.class,
                value="http://calculator.com/faultAction")
      }
   )
   public int add(int value1, int value2) throws AddNumbersException {
      return value1 + value2;
   }
}
```

If you use a tool to generate service artefacts from code, the WSDL tags that are generated from the preceding example are as follows:

```
<definitions targetNamespace="http://example.com/numbers" ...>
   ...
   <portType name="AddPortType">
      <operation name="Add">
         <input message="tns:AddInput" name="Parameters"
                        wsaw:Action="http://calculator.com/inputAction"/>
         <output message="tns:AddOutput" name="Result"
                          wsaw:Action="http://calculator.com/outputAction"/>
         <fault message="tns:AddNumbersException" name="AddNumbersException"
                         wsaw:Action="http://calculator.com/faultAction"/>
      </operation>
    <portType>
    ...
<definitions>
```

Web Services Addressing security considerations

It is essential that communications using Web Services Addressing (WS-Addressing) are adequately secured and that a sufficient level of trust is established between the communicating parties. You can achieve secure communications through the signing of WS-Addressing message-addressing properties and the encryption of endpoint references.

Perform these actions for both the supported addressing namespaces, http://www.w3.org/2005/08/addressing and http://schemas.xmlsoap.org/ws/2004/08/addressing, even if you intend to use only one of those namespaces.

Signing of WS-Addressing message-addressing properties

You can use an assembly tool to specify the message-addressing properties, and therefore the WS-Addressing message elements, that require signing, or that require signature verification on inbound requests. The receiver of the message might rely on the presence of this verifiable signature to determine that the outbound message originated from a trusted source. Similarly, the lack of a verifiable signature that is associated with the specified inbound message addressing properties causes the rejection of the message with a SOAP fault.

Encryption of endpoint references

You can encrypt endpoint references as part of the SOAP header or SOAP body. Alternatively, you can remove the need for encryption by not including sensitive information in the [address] or [reference parameters] properties of the endpoint reference.

Use of the synchronous message exchange pattern

This method applies to JAX-WS applications only.

If you do not secure the WS-Addressing information in the SOAP message using one or more of the previous methods, and you do not have WS-Security enabled, the ReplyTo and FaultTo elements of the SOAP message could be used to send messages to a third party, potentially taking part in a Denial of Service attack. To prevent such attacks, apply a WS-Addressing policy type and configure it to specify synchronous messaging only. You should also enable WS-Policy so that this requirement is communicated to clients.

Web Services Addressing: firewalls and intermediary nodes

Using the Web Services Addressing (WS-Addressing) support in this product, you can create endpoint references that can be distributed across firewalls and intermediary nodes.

Using the WS-Addressing support, you can automatically generate endpoint references that represent endpoints on the node on which the references are generated. These endpoint references contain appropriate address information, based on the URL configured for the endpoint and any valid proxy configuration for the server on which the endpoint resides. Messages targeted at the endpoint reference from the client are routed to the endpoint through the appropriate intermediary node or nodes, as described in the following topology scenarios.

If you use the IBM proprietary API to create the endpoint reference, the topology of your system can also affect the type of endpoint reference that the WS-Addressing programming model generates. For example, if you use the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method to create an endpoint reference in a cluster environment, the endpoint reference, by default, represents an endpoint that is workload-managed in the cluster in which the endpoint was created, in accordance with the appropriate topologies in the following sections. This behavior therefore provides a performance enhancement for the application.

Note: If the requesting application component runs under a transaction or in an HTTP session, affinity constraints might apply to the workload-management of endpoints.

- Use the “Direct connection” topology for non-clustered configurations.
- Use the “HTTP server, such as IBM HTTP Server” on page 679, topology when endpoint references refer to services that:
  - are deployed in a workload-managed cluster
  - do not access any stateful information that is localized to a specific server
- Use the “Proxy Server for IBM WebSphere Application Server” on page 679 topology, or the “HTTP server with a Proxy Server for IBM WebSphere Application Server” on page 680 topology, when endpoint references refer to services that:
  - are deployed in a workload-managed cluster
  - optionally, access stateful information that is localized to a specific server
  - optionally, can be failed over in a highly-available configuration

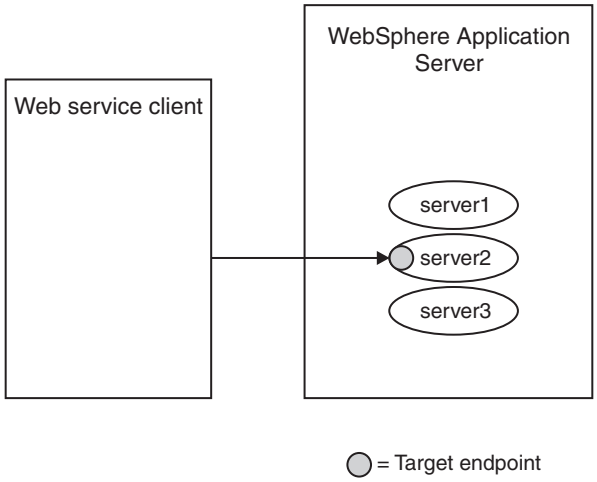
  The HTTP server with a Proxy Server for IBM WebSphere Application Server topology is useful when the HTTP server itself has no integrated capability for affinity-based routing to WS-Addressing endpoints.

For endpoint references that refer to services that do not access stateful information that is localized to a specific server, all the following topology scenarios are suitable.

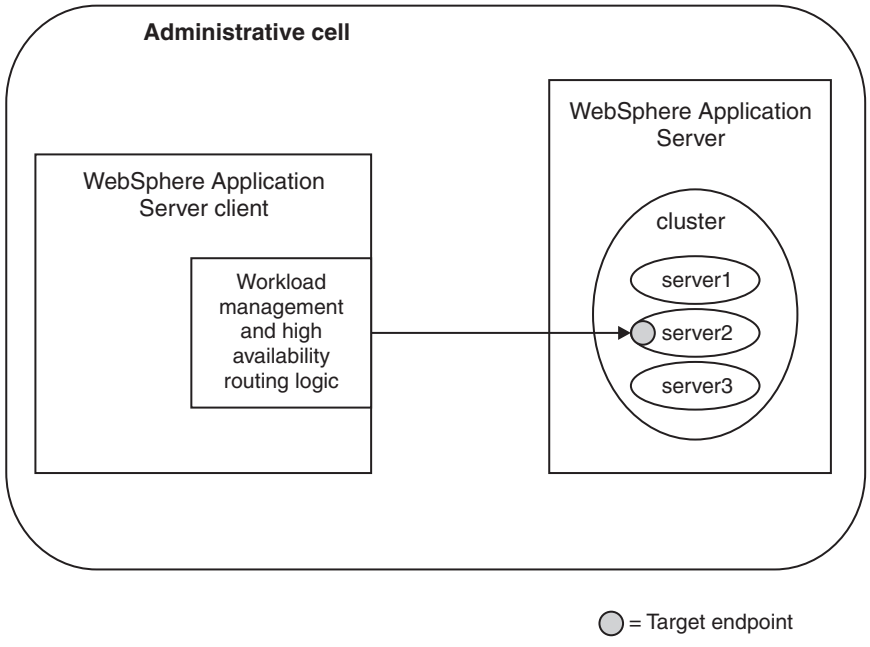
Direct connection

Use this topology for non-clustered configurations.

In this topology, there is no intermediary node. The client communicates directly with the server on which the target endpoint resides. In this topology, the WS-Addressing APIs automatically generate the appropriate endpoint reference address, based on the URL configured for the Web service module. This scenario is illustrated in the following diagram.



You can also use this topology when endpoint references created using the IBM proprietary API refer to services that are deployed in a workload-managed cluster. However, messages targeted at the endpoint reference are workload-managed only if the client targeting the endpoint reference is a WebSphere Application Server client, at Version 6.1 or later, that exists in the same administrative cell as the endpoint, as illustrated in the following diagram.

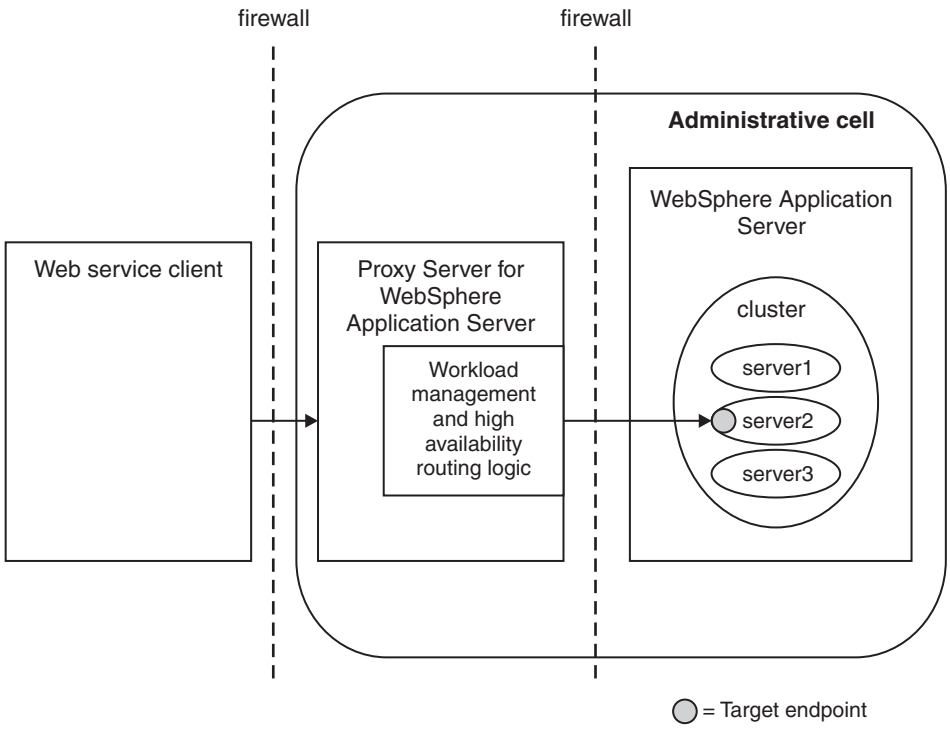


Endpoint references created using the standard JAX-WS API are not workload managed.

Proxy Server for IBM WebSphere Application Server

Use this topology when endpoint references refer to services that are deployed in a workload-managed cluster, optionally access stateful information that is localized to a specific server, and optionally can be failed over in a highly-available configuration.

In this topology, the WS-Addressing APIs automatically generate the appropriate endpoint reference address, based on the URL prefix of the Proxy Server for IBM WebSphere Application Server that is configured for the target Web service module. You must provide HTTP endpoint URL information, that is, configure the HTTP URL prefix for each deployment of each application. The client can exist outside the administrative cell that contains the proxy server and target server. The client communicates with the proxy server, which dynamically routes the client requests to the appropriate server in the cluster.

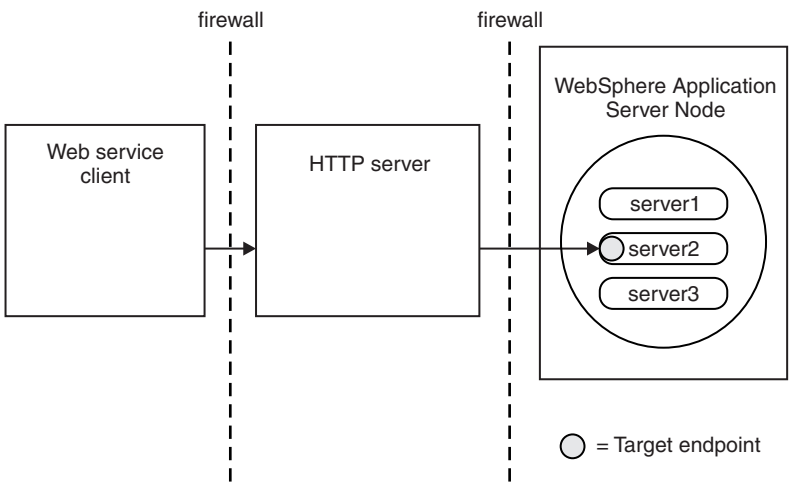


If the proxy that is addressed by the endpoint reference is a Proxy Server for IBM WebSphere Application Server, at Version 6.1 or later, that exists in the same administrative cell as the endpoint, messages targeted at a workload-managed endpoint reference are workload-managed, based on the cluster.

HTTP server, such as IBM HTTP Server

Use this topology when endpoint references refer to services that are deployed in a workload-managed cluster, and that do not access any stateful information that is localized to a specific server.

In this topology, the IBM WS-Addressing API automatically generates the appropriate endpoint reference address based on the URL prefix of the HTTP server that is configured for the target Web service module. You must provide HTTP endpoint URL information, that is, configure the HTTP URL prefix for each deployment of each application. The client communicates with the HTTP server, which then routes the client requests to a specific server based on the HTTP server configuration.

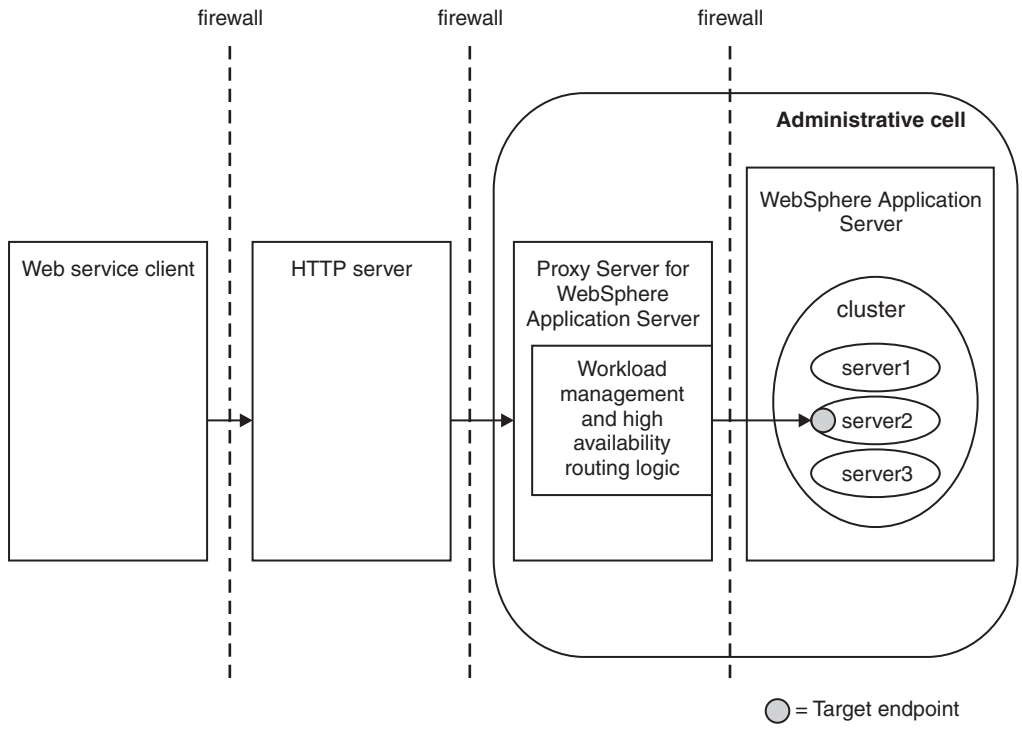


HTTP server with a Proxy Server for IBM WebSphere Application Server

Use this topology when endpoint references refer to services that are deployed in a workload-managed cluster, optionally access stateful information that is localized to a specific server, and optionally, can be failed over in a highly-available configuration. The topology is similar to the Proxy Server for IBM WebSphere Application Server topology, but supports the use of any HTTP server as the external reverse proxy.

In this topology, the WS-Addressing API automatically generates the appropriate endpoint reference address based on the URL prefix of the HTTP server that is configured for the target Web service module. You must provide HTTP endpoint URL information, that is, configure the HTTP URL prefix for each deployment of each application.

The client communicates with the HTTP server, which you configure, by routing requests from a plug-in to a proxy server, to forward the client requests to a Proxy Server for IBM WebSphere Application Server. The proxy then dynamically routes the requests to the appropriate server.



If the proxy that is addressed by the endpoint reference is Proxy Server for IBM WebSphere Application Server, at Version 6.1 or later, and exists in the same administrative cell as the endpoint, messages targeted at a workload-managed endpoint reference are workload-managed, based on the cluster.

Web Services Addressing and the service integration bus

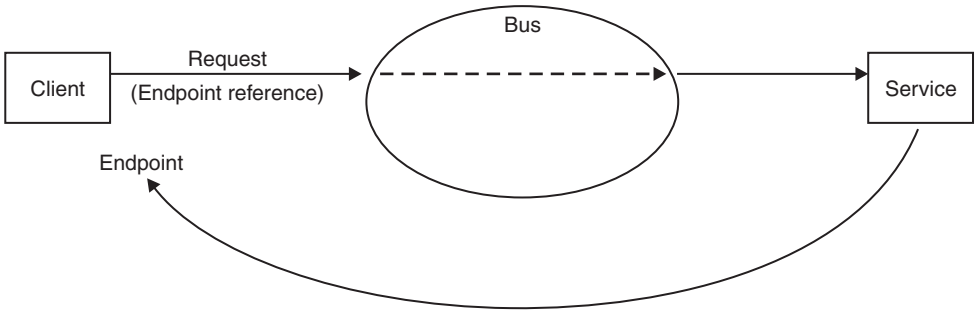
If you are using the Web Services Addressing (WS-Addressing) support, the presence of a service integration bus can affect the routing of messages. If you are also using a firewall, you might have to perform some additional configuration.

In the following scenarios, the client must conform to the WS-Addressing specification.

One-way messaging scenario

The path taken by one-way messages is as follows:

1. The client sends a request, containing an endpoint reference specifying the endpoint to which replies are sent, to the service integration bus. This request is a one way request, so the client does not wait for a response.
2. The bus passes the message intact to the Web service.
3. The Web service sends a response directly to the endpoint that is specified in the request.

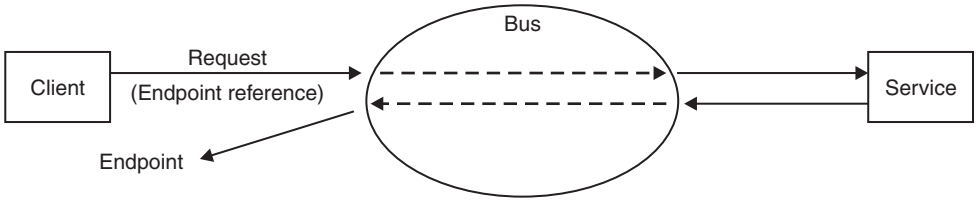


This scenario works if messages can flow directly from the Web service to the endpoint. If you have a configuration that does not support direct message flow, for example if you have a firewall, you must create handlers that can redirect the message as required.

Request-response messaging scenario

For request-response scenarios, the messages take the following path:

1. The client sends a request, containing an endpoint reference specifying the endpoint to which replies are sent, to the service integration bus.
2. The bus passes the message intact to the Web service, as a synchronous request. As the message leaves the bus, the endpoint reference is replaced with the anonymous URI listed in the WS-Addressing specification. This step ensures that the Web service does not send a response directly to the endpoint.
3. The Web service sends a response back to the bus, as part of the synchronous interaction.
4. As the message leaves the bus, the anonymous URI is replaced with the original endpoint reference, enabling the bus to pass the message to the endpoint.



Using the Web Services Addressing APIs: Creating an application that uses endpoint references

This product provides application programming interfaces for applications that need to create endpoint references and use those endpoint references to target Web service endpoints.

Before you begin

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Perform this task if you are a Web service developer who needs to create endpoint references within an application, and then use these references to target Web service resource instances. For example, a WSRF application developer.

1. Create a Web service that is referenced by an endpoint reference, and a client that accesses the Web service. For JAX-WS 2.1 applications, use the instructions in "Creating a JAX-WS Web service application that uses Web Services Addressing." For JAX-WS 2.0 or JAX-RPC applications, use the instructions in "Creating a JAX-RPC Web service application that uses Web Services Addressing" on page 688
2. Optional: You can extend the application that you created in the previous step so that it conforms to the Web Services Resource Framework (WSRF) specifications, by following the instructions in "Creating stateful Web services using the Web Services Resource Framework" on page 715.

Creating a JAX-WS Web service application that uses Web Services Addressing

Web Services Addressing (WS-Addressing) aids interoperability between Web services by defining a standard way to address Web services and provide addressing information in messages. This task describes the steps that are required to create a JAX-WS Web service that is accessed using a WS-Addressing endpoint reference. The task also describes the extra steps that are required to use stateful resources as part of the Web service.

Before you begin

The steps that are described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Perform this task if you are creating a JAX-WS Web service that uses the WS-Addressing specification. This task uses the JAX-WS 2.1 WS-Addressing APIs to create the required endpoint reference. Alternatively, you can create endpoint references using the IBM proprietary WS-Addressing API, and convert them into JAX-WS 2.1 API objects for use with the rest of the application.

1. Provide a Web service interface that returns an endpoint reference to the target service.

   The interface must return an endpoint reference, which it can do by using a factory operation or a separate factory service. The target service can front a resource instance, for example a shopping cart.
2. Implement the Web service created in the previous step. For the WS-Addressing portion of the implementation, perform the following steps:
   a. Include annotations to specify WS-Addressing behavior. See "Web Services Addressing annotations" on page 675 for more details.
   b. Optional: If your interface involves a Web service that fronts a resource instance, create or look up the resource instance.
   c. Optional: If you are using a resource instance, obtain the identifier of the resource. The resource identifier is application dependent and might be generated during the creation of the resource instance.

      Note: Do not put sensitive information in the resource identifier, because the identifier is propagated in the SOAP message.
   d. Create an endpoint reference that references the Web service by following the instructions in "Creating endpoint references using the JAX-WS 2.1 Web Services Addressing API" on page 686. If you are using a resource instance, pass in the resource identifier as a parameter.
   e. Return the endpoint reference.
3. If your Web service uses resource instances, extend the implementation to match incoming messages to the appropriate resource instances. Because you associated the resource identifier with the endpoint reference that you created earlier, any incoming messages targeted at that endpoint reference contain the resource identifier information as a reference parameter in the SOAP header of the message. Because the resource identifier is passed in the SOAP header, you do not need to expose it on the Web service interface. When WebSphere Application Server receives the message, it puts this information into the message context on the thread. Extend the implementation to perform the following actions:

a. Obtain the resource instance identifier from the message context.
   - If you are using the 2005/08 WS-Addressing namespace, use the REFERENCE_PARAMETERS property of the MessageContext class.
   - If you are using the 2004/08 WS-Addressing namespace, you must use the IBM WS-Addressing API, specifically the EndpointReferenceManager.getReferenceParameterFrom MessageContext(QName resource_id) method.

   Use the following method for the 2005/08 namespace:

```
...
List resourceIDList = (List)getContext().getMessageContext().get(MessageContext.REFERENCE_PARAMETERS);
...
```

   Use the following method for the 2004/08 namespace:

```
...

String resource_identifier =
        EndpointReferenceManager.getReferenceParameterFromMessageContext(PRINTER_ID_PARAM_QNAME);
...
```

b. Forward the message to the appropriate resource instance.

4. Optional: Configure a proxy client to communicate with the service.
   a. Use the wsimport or xjc tool to generate the artifacts required by the client.

      Note: If you want to use the 2004/08 WS-Addressing specification, specify the provided binding file, *app_server_root*/util/SubmissionEndpointReference.xjb, as the -b parameter of the tool. This parameter tells the tool to generate endpoint reference objects using the SubmissionEndpointReference class which is part of the IBM implementation of the standard JAX-WS 2.1 API. If you do not specify this bindings file, the resulting endpoint reference objects will not work with the standard JAX-WS 2.1 API.

   b. In the client code, create an instance of the service class.
   c. Obtain a proxy object from the service class. There are several ways to use the JAX-WS 2.1 API to obtain proxy objects. For example, there are several getPort methods on the Service class and one on the EndpointReference class. For more information, refer to the API documentation.
   d. Optional: Use the Addressing or SubmissionAddressing feature to enable WS-Addressing support. For example, create a proxy using a getPort method that accepts Web service features as a parameter. If you prefer, you can enable WS-Addressing support using another method, such as policy sets. For more information see “Enabling Web Services Addressing support for JAX-WS applications” on page 708.
   e. Use the proxy object to invoke the service method that returns the endpoint reference.

   The following sample code shows a client invoking a Web service to add two numbers together. The Web service issues a ticket (the resource identifier) to the client, and requires the client to use this ticket when invoking the Web service.

   The client creates two proxies. The first proxy obtains the ticket as an endpoint reference from the service. The second proxy uses the AddressingFeature class to enable WS-Addressing for the 2005/08 specification, and invokes the service to add the two numbers together.

```
...
CalculatorService service = new CalculatorService();
// Create the first proxy
Calculator port1 = service.getCalculatorServicePort();
// Obtain the ticket as an endpoint reference from the service
W3CEndpointReference epr = port1.getTicket();

// Create the second proxy, using an addressing feature to enable WS-Addressing
Calculator port2 = epr.getPort(Calculator.class, new AddressingFeature());
```

```
// Invoke the service to add the numbers
int answer = port2.add(value0, value1);
System.out.println("The answer is: " + answer);
...
```

Note: If the metadata of the endpoint reference conflicts with the information already associated with the outbound message, for example if the proxy object is configured to represent a different interface, a javax.xml.ws.WebServiceException exception is thrown on attempts to invoke the endpoint.

If you want to set message-addressing properties, such as a reply to endpoint, you must use the IBM proprietary WS-Addressing SPI and the BindingProvider class, as described in “Specifying and acquiring message-addressing properties using the IBM proprietary Web Services Addressing SPIs” on page 700.

5. Optional: Configure a Dispatch client to communicate with the service. You can configure a client in different ways; the following steps describe one example.
   a. Create an instance of the service.
   b. Add a port to the service object.
   c. Create an instance of the Dispatch class, passing in the endpoint reference.
   d. Create a Dispatch<T> object. Use the Service.createDispatch method with the following parameters:
      - The endpoint reference returned by the service, which represents the resource to forward messages to.
      - An array of Web service features. Include one or more WS-Addressing features to enable WS-Addressing. See “Enabling Web Services Addressing support for JAX-WS applications” on page 708 for more details.

      There are several variations of the Service.createDispatch method; see the API documentation for more details.
   e. Compose the client request message.
   f. Invoke the service endpoint with the Dispatch client.

   The following code shows an example fragment of a Dispatch client that enables 2004/08 WS-Addressing.

```
...
CalculatorService service = new CalculatorService();
Dispatch(<SOAPMessage> dispatch = service.createDispatch(
    endpointReference,
    SOAPMessage.class,
    Service.Mode.MESSAGE,
    new SubmissionAddressingFeature(true));
...
```

Results

The Web service and client are configured to use endpoint references through the WS-Addressing support. For a detailed example that includes code, see “Example: Creating a Web service that uses the JAX-WS 2.1 Web Services Addressing API to access a generic Web service resource instance” on page 694.

What to do next

- Refer to “Web Services Addressing security considerations” on page 676 for information about security with WS-Addressing.
- Deploy the application. If you used WS-Addressing annotations or features in the code, you do not have to take any additional steps to enable WS-Addressing support. For more information and for other scenarios that might require additional steps, for example enabling WS-Addressing support using policy sets, see “Enabling Web Services Addressing support for JAX-WS applications” on page 708.

Creating endpoint references using the JAX-WS 2.1 Web Services Addressing API:

Endpoint references are a primary concept of the Web Services Addressing (WS-Addressing) interoperability protocol, and provide a standard mechanism to encapsulate information about specific Web service endpoints. This product provides interfaces for you to create endpoint references using the standard JAX-WS 2.1 API.

About this task

This task is a subtask of “Creating a JAX-WS Web service application that uses Web Services Addressing” on page 683.

Perform this task if you are writing an application that uses the standard JAX-WS 2.1 WS-Addressing API. Such applications require endpoint references to target Web service endpoints. The standard JAX-WS API is designed to create only simple endpoint references, and therefore has the following restrictions:

- You cannot create highly available or workload managed endpoint references.
- You cannot create endpoint references that represent stateful session beans.
- You cannot use classes created using the JAX-WS 2.1 API with the IBM proprietary WS-Addressing SPI.

You can overcome these restrictions by using the IBM proprietary WS-Addressing API to create the endpoint references and then converting them into standard JAX-WS 2.1 endpoint references that can be used by the rest of the application. Refer to “Creating endpoint references using the JAX-WS 2.1 Web Services Addressing API” for instructions.

Note: Endpoint references that are created using this API contain metadata that complies with the WS-Addressing 1.0 Metadata specification. This behavior is not a requirement of the JAX-WS 2.1 specification, so endpoint references created using vendor software might contain metadata that complies with a different specification. This difference could cause problems if you are interoperating with an application created using vendor software.

- If an endpoint needs to create an endpoint reference that represents itself, use the getEndpointReference method of the Web service context object, passing in an Element object representing the reference parameters to be associated with the endpoint reference (or a null object if you do not want to specify any reference parameters).

  By default, this method creates a W3CEndpointReference object. If you want to create a SubmissionEndpointReference object, representing an endpoint that conforms to the 2004/08 WS-Addressing specification, pass the endpoint reference type as a parameter. For example, the following code fragment uses the getEndpointReference method to return a W3CEndpointRerence object that has a ticket ID associated with it:

```
...
@WebService(name="Calculator",
            targetNamespace="http://calculator.org")

public class Calculator {
    @Resource
    WebServiceContext wsc;


    ...
    // Create the ticket id
    element = document.createElementNS(
                      "http://calculator.jaxws.axis2.apache.org", "TicketId");
    element.appendChild( document.createTextNode("123456789") );
    ...

    public W3CEndpointReference getEPR() {
        // Get the endpoint reference and associate the ticket id
        // with it as a reference parameter
        W3CEndpointReference epr = (W3CEndpointReference)wsc.getEndpointReference(element);
```

```
        return epr;
    }
...
```

The following line of code shows how to create a 2004/08 endpoint reference for the preceding sample:

```
SubmissionEndpointReference epr = (SubmissionEndpointReference)
        wsc.getEndpointReference(SubmissionEndpointReference.class, element);
```

- If an endpoint needs to create an endpoint reference that represents a different endpoint, use either the W3CEndpointReferenceBuilder class or the SubmissionEndpointReferenceBuilder class, depending on the namespace that you want to use.
  1. Create an instance of the appropriate builder class. Use the W3CEndpointReferenceBuilder class if you want to create an endpoint reference that complies with the 2005/08 WS-Addressing specification. Use the SubmissionEndpointReferenceBuilder class if you want to create an endpoint reference that complies with the 2004/08 WS-Addressing specification.
  2. Set the following property or properties of the builder instance according to the location of the endpoint.
     - If the endpoint is in another module in this application, set the serviceName and endpointName properties to appropriate values. You must set the serviceName property before you set the endpointName property, otherwise the application throws an error. The endpoint reference that is returned contains a suitable address for the endpoint, as determined by the implementation.

       Note: This behavior differs from the IBM WS-Addressing API, in that creating an endpoint reference using the com.ibm.websphere.wsaddressing.EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method is not restricted to endpoints in the same application.
     - If the endpoint is in another Java EE application, set the address property to point to the endpoint.
  3. Optional: Set other properties of the builder instance as required. For example, if the Web service is used to access a resource instance, use the referenceParameter property to associate the identifier of the resource with the endpoint reference. For more information on the properties that you can set, see the API documentation.
  4. Invoke the build method on the builder instance to obtain the endpoint reference.

  For example, the following code fragment uses the W3CEndpointReferenceBuilder class to obtain an endpoint reference that complies with the 2005/08 specification, and points to an endpoint that is in another module in this application:

```
...
@WebService(name="Calculator", targetNamespace="http://calculator.org")
public class Calculator {

    public W3CEndpointReference getEPR() {
        ...
        // Create the builder object
        W3CEndpointReferenceBuilder builder = new
                                W3CEndpointReferenceBuilder();

        // Modify builder properties
        builder.address(calculatorServiceURI);

        // Create the endpoint reference from the builder object
        W3CEndpointReference epr = builder.build();
        return epr;
    }
...
```

The following code fragment uses the SubmissionEndpointReferenceBuilder class to obtain an endpoint reference that complies with the 2004/08 specification, and points to an endpoint that is in another application:

```
...
@WebService(name="Calculator", targetNamespace="http://calculator.org")
public class Calculator {

    public W3CEndpointReference getEPR() {
        ...
        // Create the builder object
        SubmissionEndpointReferenceBuilder builder = new
                          SubmissionEndpointReferenceBuilder();

        // Modify builder properties
        builder.serviceName(calculatorService);
        builder.endpointName(calculatorPort);

        // Create the endpoint reference from the builder object
        SubmissionEndpointReference epr = builder.build();
        return epr;
    }
...
```

Results

You created an endpoint reference for use by your application.

What to do next

1. If required, convert the endpoint reference to an instance of the com.ibm.websphere.wsaddressing.EndpointReference class, using the createIBMEndpointReference method. For example, on a client you might want to set the FaultTo message addressing property for outbound messages. You cannot set this property using the JAX-WS 2.1 API, so you must convert the endpoint reference representing the FaultTo endpoint to an instance of the com.ibm.websphere.wsaddressing.EndpointReference class, before setting it as a property on the BindingProvider object.
2. Continue with “Creating a JAX-WS Web service application that uses Web Services Addressing” on page 683.

Creating a JAX-RPC Web service application that uses Web Services Addressing

Web Services Addressing (WS-Addressing) aids interoperability between Web services by defining a standard way to address Web services and provide addressing information in messages. This task describes the steps that are required to create a JAX-RPC Web service that is accessed using a WS-Addressing endpoint reference. The task also describes the extra steps that are required to use stateful resources as part of the Web service.

Before you begin

The steps that are described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Perform this task if you are creating a Web service that uses the WS-Addressing specification.

1. Provide a Web service interface, by creating or generating a Web Services Description Language (WSDL) document for the Web service, that returns an endpoint reference to the target service. The interface must return an endpoint reference, which it can do by using a factory operation or a separate factory service. The target service can front a resource instance, for example a shopping cart.

2. Implement the Web service created in the previous step. For the WS-Addressing portion of the implementation, perform the following steps:
   a. Create an endpoint reference that references the Web service, by following the instructions in “Creating endpoint references using the IBM proprietary Web Services Addressing API” on page 690.
   b. Optional: If your interface involves a Web service that fronts a resource instance, create or look up the resource instance.
   c. Optional: If you are using a resource instance, obtain the identifier of the resource and associate it with the endpoint reference as a reference parameter, using the EndpointReference.setReferenceParameter(QName resource_id_name, String value) method. The resource identifier is application-dependent and might be generated during the creation of the resource instance.

      Note: Do not put sensitive information in the resource identifier, because the identifier is propagated in the SOAP message.

      The endpoint reference now targets the resource.
   d. Return the endpoint reference.
3. If your Web service uses resource instances, extend the implementation to match incoming messages to the appropriate resource instances. Because you associated the resource identifier with the endpoint reference that you created earlier, any incoming messages targeted at that endpoint reference contain the resource identifier information as a reference parameter in the SOAP header of the message. Because the resource identifier is passed in the SOAP header, you do not need to expose it on the Web service interface. When WebSphere Application Server receives the message, it puts this information into the message context on the thread. Extend the implementation to perform the following actions:
   a. Obtain the resource instance identifier from the message context, using the EndpointReferenceManager.getReferenceParameterFromMessageContext(QName resource_id_name) method.
   b. Forward the message to the appropriate resource instance.
4. To configure a client to communicate with the service, use the endpoint reference that is produced by the service in the first step to send messages to the endpoint.
   a. Obtain a Stub object (by looking up the service in the Java Naming and Directory Interface (JNDI)), or create an empty Call object.
   b. Associate the endpoint reference with the proxy object. Use the setProperty(String property_name, Object value) method of the Stub or Call object, using the WS-Addressing constant WSADDRESSING_DESTINATION_EPR as the property name, and the endpoint reference as the value.

      This procedure automatically configures the Stub or Call object, to represent the Web service (or resource instance if your interface uses a Web service that fronts a resource instance) of the endpoint reference. For Call objects, this process includes the configuration of the interface and endpoint metadata (portType and port elements) that are associated with the endpoint reference.

      Note: If the metadata of the endpoint reference conflicts with the information already associated with the outbound message, for example if the Stub object is configured to represent a different interface, a javax.xml.rpc.JAXRPCException exception is thrown on attempts to invoke the endpoint.

   Invocations on the Stub or Call object are now targeted at the Web service or resource instance that is defined by the endpoint reference. When an invocation occurs, the product adds appropriate message addressing properties, such as a reference parameter contained within the endpoint reference that identifies a target resource, to the message header.

Results

The Web service and client are configured to use endpoint references through the WS-Addressing support. For a detailed example that includes code, see “Example: Creating a Web service that uses the IBM proprietary Web Services Addressing API to access a generic Web service resource instance” on page 692.

What to do next

- Refer to “Web Services Addressing security considerations” on page 676 for information about security with WS-Addressing.
- Deploy the application. For this scenario, you do not have to take any additional steps to enable the WS-Addressing support in WebSphere Application Server because you specified a WS-Addressing property on the client. For more information, and for other scenarios which might require additional steps, see “Enabling Web Services Addressing support for JAX-RPC applications” on page 712.

Creating endpoint references using the IBM proprietary Web Services Addressing API:

Endpoint references are a primary concept of the Web Services Addressing (WS-Addressing) interoperability protocol, and provide a standard mechanism to encapsulate information about specific Web service endpoints. This product provides interfaces for you to create endpoint references using the IBM proprietary implementation of the WS-Addressing standard.

About this task

This task is a subtask of “Creating a JAX-RPC Web service application that uses Web Services Addressing” on page 688.

Perform this task if you are writing an application that uses the IBM proprietary WS-Addressing API. Such applications require endpoint references to target Web service endpoints. When you are writing the application, you might not know the address of the endpoint, because the address can change when the application is deployed. Using the IBM proprietary API, you can either specify the endpoint address, or allow the product to generate it for you at run time.

You can also specify the behavior of endpoint references in a cluster environment.

If you want to use endpoint reference objects from the standard JAX-WS 2.1 API instead of the IBM proprietary equivalents, but want the extra functionality provided by the IBM proprietary API, create the endpoint references using the methods described in this task and then convert them using the supplied converter classes. For example, you might want to perform such a conversion if you have a JAX-WS service application and you are creating endpoint references that represent stateful session beans, or that have an affinity to a particular server, or are workload managed. You cannot create such endpoint references using the JAX-WS 2.1 API.

- To create an endpoint reference with an address that you specify directly, use the WS-Addressing EndpointReferenceManager.createEndpointReference(URI address) method of the system programming interface (SPI) provided. This method is useful in test scenarios, where the address of the service does not change.
- To create an endpoint reference with an address that is automatically generated by the product, perform the following steps.
  1. If you created the Web service deployment descriptor file, webservices.xml, manually, ensure that the webservice-description-name in the file is the same as the local part of the Web Services Description Language (WSDL) service name. If you generated the webservices.xml file using the tools provided, the names match by default. This match is required for the generation of the correct URI for the endpoint reference.
  2. Create the endpoint reference using the method that is appropriate for the object that the reference will represent.

– If you are creating an endpoint reference to represent a stateful session bean that maintains in-memory state, create the endpoint reference using the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName, Remote statefulSessionBean) method of the application programming interface (API) provided. This method ensures that requests are targeted at the specific server that hosts the stateful session bean instance, and are not workload-managed.

  Also, if high availability for stateful session beans is specified, the endpoint reference remains valid even if the stateful session bean is failed over.

  Note: Affinity to stateful session beans using this method is not supported on the z/OS operating system. If a highly available stateful session bean fails over to another control region, or is passivated from one servant region and reactivated on another servant region in the same control region, the endpoint reference is no longer valid. To achieve affinity with a stateful session bean, run the application in a transaction to use transactional context affinity, or use HTTP session affinity.

– If you are creating an endpoint reference to represent any other object, create the endpoint reference using the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method of the API. The combination of service name and endpoint name must be unique in the server. If there is more than one Web service application with the same service name and endpoint name, the application server cannot generate a unique URI object for the endpoint. If you cannot ensure that the combination of service name and endpoint name is unique, use an SPI method to create the endpoint reference.

  Because the endpoint reference might be workload-managed at a later date, ensure that the endpoint does not contain any in-memory state. For JAX-WS applications, you can prevent the endpoint reference from being workload-managed by attaching a WS-Addressing policy set to the application and configuring the policy set binding to prevent the workload management of referenced endpoints in clusters.

  Note: You can use the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method to create an endpoint reference for any endpoint reference in the application server. This behavior differs from that of the JAX-WS 2.1 API, where creating an endpoint reference using the service name and endpoint name is restricted to endpoints within the same Java EE application.

When the application invokes either of the previous two methods, the product generates the address URI for the endpoint reference, and puts the service name and endpoint name into the metadata of the newly created endpoint reference.

Note: If you configured a virtual host for the server on which the endpoint is created, the URI of the endpoint reference refers to the virtual host of the HTTP server's configuration. You can override the HTTP endpoint URL information using the administrative console, see Provide HTTP endpoint URL information. The methods described previously will use the overridden value to generate the address URI for the endpoint reference.

Results

You created an endpoint reference for use by your application.

What to do next

1. If you want to convert the endpoint references from IBM proprietary WS-Addressing objects to standard JAX-WS 2.1 WS-Addressing objects, use one of the following methods of the com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceConverter class, depending on the namespace of the endpoint reference:
   - createW3CEndpointReference(EndpointReference epr): use this method if the EndpointReference object uses the 2005/08 specification. This method creates a W3CEndpointReference object.

- createSubmissionEndpointReference(EndpointReference epr): use this method if the EndpointReference object uses the 2004/08 specification. This method creates a SubmissionEndpointReference object.

2. Continue with “Creating a JAX-RPC Web service application that uses Web Services Addressing” on page 688, or if you converted the endpoint reference to the standard JAX-WS 2.1 API, continue with “Creating a JAX-WS Web service application that uses Web Services Addressing” on page 683.

Example: Creating a Web service that uses the IBM proprietary Web Services Addressing API to access a generic Web service resource instance

Consider an IT organization that has a network of printers that it wants to manage using Web services. The organization might represent each printer as a resource that is addressed through an endpoint reference. This example shows how to code such a service using the IBM proprietary Web Services Addressing (WS-Addressing) application programming interfaces (APIs) that are provided by WebSphere Application Server, and JAX-WS.

Providing a Web service interface that returns an endpoint reference to the target service

The IT organization implements a PrinterFactory service that offers a CreatePrinter portType element. This portType element accepts a CreatePrinterRequest message to create a resource that represents a logical printer, and responds with an endpoint reference that is a reference to the resource.

The WSDL definition for such a PrinterFactory service might include the following code:

```
<wsdl:definitions targetNamespace="http://example.org/printer" ...
                  xmlns:pr=" http://example.org/printer">
  <wsdl:types>
    ...
    <xsd:schema...>
      <xsd:element name="CreatePrinterRequest"/>
      <xsd:element name="CreatePrinterResponse"
                    type="wsa:EndpointReferenceType"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="CreatePrinterRequest">
    <wsdl:part name="CreatePrinterRequest"
               element="pr:CreatePrinterRequest" />
  </wsdl:message>
  <wsdl:message name="CreatePrinterResponse">
    <wsdl:part name="CreatePrinterResponse"
               element="pr:CreatePrinterResponse" />
  </wsdl:message>
  <wsdl:portType name="CreatePrinter">
    <wsdl:operation name="createPrinter">
      <wsdl:input name="CreatePrinterRequest"
                  message="pr:CreatePrinterRequest" />
      <wsdl:output name="CreatePrinterResponse"
                  message="pr:CreatePrinterResponse" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

The CreatePrinter operation in the previous example returns a wsa:EndpointReference object that represents the newly created Printer resource. The client can use this endpoint reference to send messages to the service instance that represents the printer.

Implementing the Web service interface

The createPrinter method shown in the following example creates an endpoint reference to the Printer service. The operation then obtains the identifier for the individual printer resource instance, and

associates it with the endpoint reference. Finally, the createPrinter method converts the EndpointReference object, which now represents the new printer, into a W3CEndpointReference object, and returns the converted endpoint reference.

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
import com.ibm.websphere.wsaddressing.EndpointReference;
import com.ibm.websphere.wsaddressing.jaxws.EndpointReferenceConverter;
import com.ibm.websphere.wsaddressing.jaxws.W3CEndpointReference;

import javax.xml.namespace.QName;

public class MyClass {

    // Create the printer
    ...

    // Define the printer resource ID as a static constant as it is required in later steps
    public static final QName PRINTER_ID_PARAM_QNAME = new QName("example.printersample",
                                                          "IBM_WSRF_PRINTERID", "ws-rf-pr" );
    public static final QName PRINTER_SERVICE_QNAME = new QName("example.printer.com", "printer", "...");
    public static final String PRINTER_ENDPOINT_NAME = new String("PrinterService");

    public W3CEndpointReference createPrinter(java.lang.Object createPrinterRequest)
    throws Exception {
     // Create an EndpointReference that targets the appropriate WebService URI and port name.
     EndpointReference epr =  EndpointReferenceManager.createEndpointReference(PRINTER_SERVICE_QNAME,
                                                                    PRINTER_ENDPOINT_NAME);

     // Create or lookup the stateful resource and derive a resource
     // identifier string.
     String resource_identifier = "...";

     // Associate this resource identifier with the EndpointReference as
     // a reference parameter.
     // The choice of name is arbitrary, but should be unique
     // to the service.
     epr.setReferenceParameter(PRINTER_ID_PARAM_QNAME,resource_identifier);
     // The endpoint reference now targets the resource rather than the service.
     ...

     return EndpointReferenceConverter.createW3CEndpointReference(epr);
    }
}
```

Extending the target service to match incoming messages to Web service resource instances

Because of the Web service implementation described previously, the printer resource instance now has a unique identifier embedded in its endpoint reference. This identifier becomes a reference parameter in the SOAP header of subsequent messages that are targeted at the Web service, and can be used by the Web service to match incoming messages to the appropriate printer.

When a Web service receives a message containing WS-Addressing message-addressing properties, the WebSphere Application Server processes these properties before the message is dispatched to the application endpoint, and sets them into the message context on the thread. The Printer Web service application accesses the reference parameters that are associated with the target endpoint from the WebServiceContext object, as illustrated in the following example:

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
...
 // Initialize the reference parameter name
 QName name = new QName(..);
 // Extract the String value.
 String resource_identifier =
        EndpointReferenceManager.getReferenceParameterFromMessageContext(PRINTER_ID_PARAM_QNAME);
```

The Web service implementation can forward messages based on the printer identity acquired from the getReferenceParameterFromMessageContext method to the appropriate printer instances.

Using endpoint references to send messages to an endpoint

The client creates a JAX-WS proxy for the printer, and converts the proxy into a BindingProvider object. The client then associates the EndpointReference object obtained previously with the request context of the BindingProvider object, as illustrated in the following example.

```
import javax.xml.ws.BindingProvider;
...

 javax.xml.ws.Service service= ...;
 Printer myPrinterProxy = service.getPort(portName, Printer.class);

 javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider)myPrinterProxy;

 // Retrieve the request context for the BindingProvider object
 Map myMap = myBindingProvider.getRequestContext();

 // Associate the endpoint reference that represents the new printer to the request context
 // so that the BindingProvider object now represents a specific printer instance.
 myMap.put(WSADDRESSING_DESTINATION_EPR, destinationEpr);

...
```

The BindingProvider object now represents the new printer resource instance, and can be used by the client to send messages to the printer through the Printer Web service. When the client invokes the BindingProvider object, WebSphere Application Server adds appropriate message-addressing properties to the message header, which in this case is a reference parameter contained within the endpoint reference that identifies the target printer resource.

Alternatively, the client can use a JAX-RPC Stub or Call object, which the client configures to represent the new printer. The use of the Call object is illustrated in the following example.

```
import javax.xml.rpc.Call;
...
 :
 // Associate the endpoint reference that represents the new printer with the call.
 call.setProperty(
        "com.ibm.websphere.wsaddressing.WSAConstants.
                            WSADDRESSING_DESTINATION_EPR ", epr);
```

From the perspective of the client, the endpoint reference is opaque. The client cannot interpret the contents of any endpoint reference parameters and should not try to use them in any way. Clients cannot directly create instances of endpoint references because the reference parameters are private to the service provider; clients must obtain endpoint references from the service provider, for example through a provider factory service, and then use them to direct Web service operations to the endpoint that is represented by the endpoint reference, as shown.

Example: Creating a Web service that uses the JAX-WS 2.1 Web Services Addressing API to access a generic Web service resource instance

Consider an IT organization that has a network of printers that it wants to manage using Web services. The organization might represent each printer as a resource that is addressed through an endpoint reference. This example shows how to code such a service using the JAX-WS 2.1 Web Services Addressing (WS-Addressing) application programming interfaces (APIs) that are provided by WebSphere Application Server.

Providing a Web service interface that returns an endpoint reference to the target service

The IT organization implements a PrinterFactory service that offers a CreatePrinter portType element. This portType element accepts a CreatePrinterRequest message to create a resource that represents a logical printer, and responds with an endpoint reference that is a reference to the resource.

The WSDL definition for such a PrinterFactory service might include the following code:

```
<wsdl:definitions targetNamespace="http://example.org/printer" ...
                  xmlns:pr=" http://example.org/printer">
  <wsdl:types>
    ...
    <xsd:schema...>
      <xsd:element name="CreatePrinterRequest"/>
      <xsd:element name="CreatePrinterResponse"
                   type="wsa:EndpointReferenceType"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="CreatePrinterRequest">
    <wsdl:part name="CreatePrinterRequest"
               element="pr:CreatePrinterRequest" />
  </wsdl:message>
  <wsdl:message name="CreatePrinterResponse">
    <wsdl:part name="CreatePrinterResponse"
               element="pr:CreatePrinterResponse" />
  </wsdl:message>
  <wsdl:portType name="CreatePrinter">
    <wsdl:operation name="createPrinter">
      <wsdl:input name="CreatePrinterRequest"
                  message="pr:CreatePrinterRequest" />
      <wsdl:output name="CreatePrinterResponse"
                  message="pr:CreatePrinterResponse" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

The CreatePrinter operation in the previous example returns a wsa:EndpointReference object that represents the newly created Printer resource. The client can use this endpoint reference to send messages to the service instance that represents the printer.

Implementing the Web service interface

The createPrinter method shown in the following example obtains the identifier for the individual printer resource instance. The operation then creates an endpoint reference to the Printer service, and associates the printer ID with the endpoint reference. Finally, the createPrinter method returns the endpoint reference.

```
import javax.xml.ws.wsaddressing.W3CEndpointReference;
import javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder;

import javax.xml.namespace.QName;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class MyClass {

  // Create the printer
  ...

  //Define the printer resource ID as a static constant as it is required in later steps
    public static final QName PRINTER_SERVICE_QNAME = new QName("example.printer.com", "printer", "...");
    public static final QName PRINTER_ENDPOINT_NAME = new QName("example.printer.com", "PrinterService", "...");

    public W3CEndpointReference createPrinter(java.lang.Object createPrinterRequest)
    {
```

```
    Document document = ...;

    // Create or lookup the stateful resource and derive a resource
    // identifier string.
    String resource_identifier = "...";

    // Associate this resource identifier with the EndpointReference as
    // a reference parameter.
    // The choice of name is arbitrary, but should be unique
    // to the service.
    Element element = document.createElementNS("example.printersample",
            "IBM_WSRF_PRINTERID");
    element.appendChild( document.createTextNode(resource_identifier) );
    ...

    // Create an EndpointReference that targets the appropriate WebService URI and port name.
    // Alternatively, the getEndpointReference() method of the MessageContext could be used.
    W3CEndpointReferenceBuilder builder = new W3CEndpointReferenceBuilder();
    builder.serviceName(PRINTER_SERVICE_QNAME);
    builder.endpointName(PRINTER_ENDPOINT_NAME);
    builder.referenceParameter(element);

    // The endpoint reference now targets the resource rather than the service.
    return builder.build();
   }
}
```

Extending the target service to match incoming messages to Web service resource instances

Because of the Web service implementation described previously, the printer resource instance now has a unique identifier embedded in its endpoint reference. This identifier becomes a reference parameter in the SOAP header of subsequent messages that are targeted at the Web service, and can be used by the Web service to match incoming messages to the appropriate printer.

When a Web service receives a message containing WS-Addressing message addressing properties, the WebSphere Application Server processes these properties before the message is dispatched to the application endpoint, and sets them into the message context on the thread. The Printer Web service application accesses the reference parameters that are associated with the target endpoint from the WebServiceContext object, as illustrated in the following example:

```
@Resource
private WebServiceContext context;
...
List list = (List) context.getMessageContext().get(MessageContext.REFERENCE_PARAMETERS);
```

If your application uses the 2004/08 version of the WS-Addressing specification, use the IBM proprietary API to retrieve the message parameters, as illustrated in the following example.

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
...
 // Initialize the reference parameter name
 QName name = new QName(..);
 // Extract the String value.
 String resource_identifier =
        EndpointReferenceManager.getReferenceParameterFromMessageContext(PRINTER_ID_PARAM_QNAME);
```

The Web service implementation can forward messages based on the printer ID to the appropriate printer instances.

Using endpoint references to send messages to an endpoint

The client uses the endpoint reference returned from the service to create a JAX-WS proxy for the printer, as illustrated in the following example.

```
javax.xml.ws.Service jaxwsServiceObject= ...;
W3CEndpointReference epr = ...;
...
Printer myPrinterProxy = jaxwsServiceObject.getPort(epr, Printer.class, new AddressingFeature());
```

The proxy object now represents the new printer resource instance, and can be used by the client to send messages to the printer through the Printer Web service. When the client invokes the service, WebSphere Application Server adds appropriate message addressing properties to the message header, which in this case is a reference parameter contained within the endpoint reference that identifies the target printer resource.

From the perspective of the client, the endpoint reference is opaque. The client cannot interpret the contents of any endpoint reference parameters and should not try to use them in any way. Clients cannot directly create instances of endpoint references because the reference parameters are private to the service provider; clients must obtain endpoint references from the service provider, for example through a provider factory service, and then use them to direct Web service operations to the endpoint that is represented by the endpoint reference, as shown.

Web Services Addressing APIs

This product provides interfaces at the application programming level to enable application developers, including developers of Web Services Resource Framework applications, to create references to, and to target, Web service resource instances. If you are a system programmer, you can use some these interfaces with the Web Services Addressing (WS-Addressing) system programming interfaces.

This product provides two separate sets of application programming interfaces (APIs):

- Standard Java API for XML-Based Web Services (JAX-WS) 2.1 APIs. Use these APIs with JAX-WS applications.
- IBM proprietary WS-Addressing APIs. Use these APIs with applications that use either JAX-WS or JAX-RPC. For JAX-WS applications, this API provides more functionality than the standard JAX-WS 2.1 API.

These APIs are described in more detail in the WS-Addressing API documentation.

JAX-WS 2.1 APIs

The standard JAX-WS 2.1 APIs in this product are contained in the javax.xml.ws.wsaddressing package. Refer to the JAX-WS 2.1 API documentation for more information about these APIs.

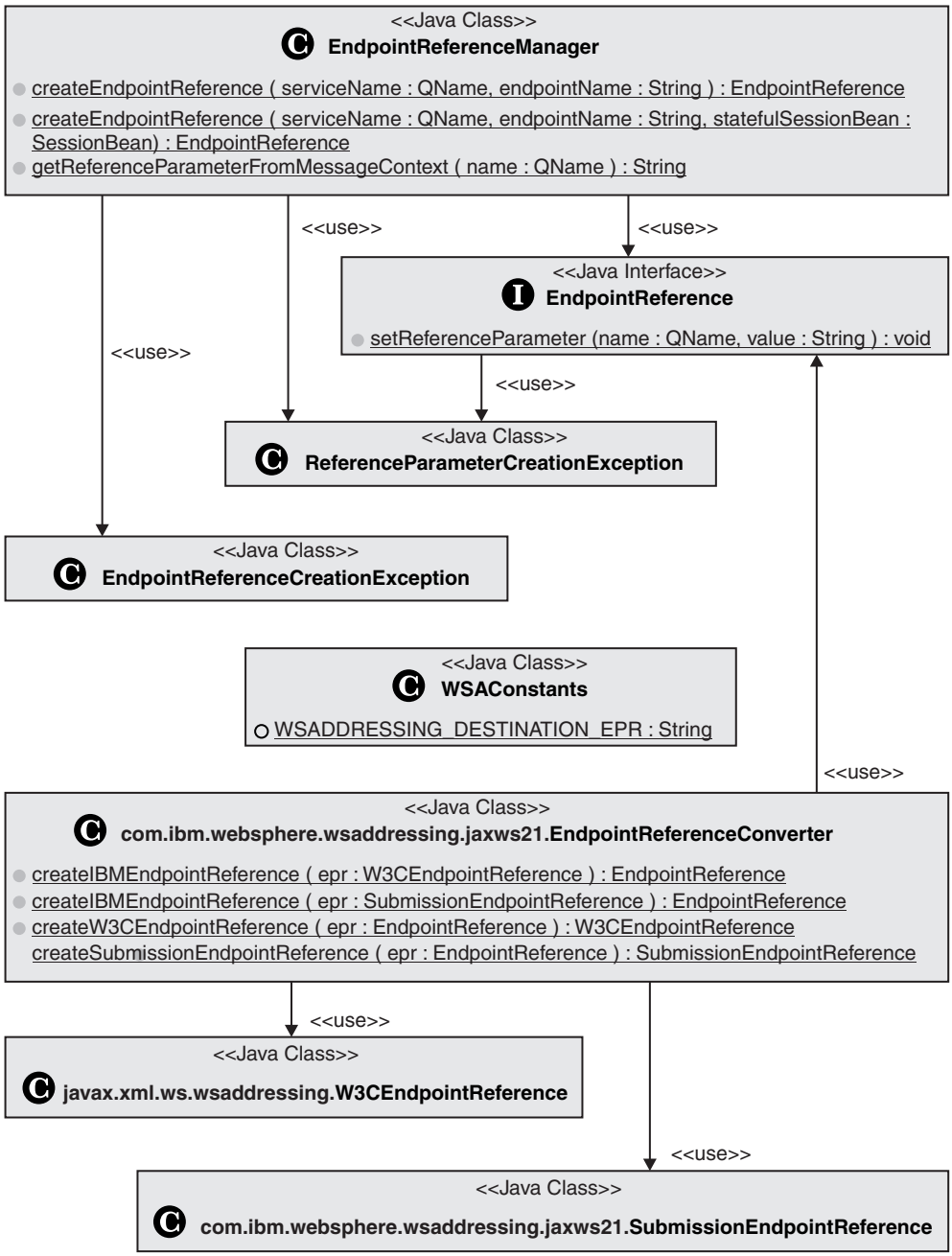
The implementation of the standard JAX-WS 2.1 APIs in this product also contains application programming interfaces, in the com.ibm.websphere.wsaddressing.jaxws21 package. These APIs are described in more detail in the generated API documentation in this information center. These APIs provide the following features:

- A class, com.ibm.websphere.wsaddressing.jaxws21.SubmissionEndpointReference, for representing endpoints that conform to the 2004/08 WS-Addressing specification.
- A class, com.ibm.websphere.wsaddressing.jaxws21.SubmissionEndpointReferenceBuilder, for creating a SubmissionEndpointReference instance to represent 2004/08 endpoints in Web services other than the one generating the endpoint reference.
- A class, com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceCoverter, for converting EndpointReference instances created using the IBM proprietary WS-Addressing API into either W3CEndpointReference or SubmissionEndpointReference instances, or back again.

- A class, com.ibm.websphere.wsaddressing.jaxws21.SubmissionAddressingFeature, for enabling WS-Addressing on clients, and an annotation, @SubmissionAddressing, for enabling WS-Addressing on servers.

IBM proprietary WS-Addressing APIs

These application programming interfaces are contained in the com.ibm.websphere.wsaddressing package and are summarized in the following diagram. The diagram also shows the following classes from the JAX-WS 2.1 API: com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceConverter, javax.xml.ws.wsaddressing.W3CEndpointReference and com.ibm.websphere.wsaddressing.jaxws21.SubmissionEndpointReference.



These interfaces provide the following features:

- A mechanism for creating a com.ibm.websphere.wsaddressing.EndpointReference instance to represent a WS-Addressing endpoint reference using the com.ibm.websphere.wsaddressing.EndpointReferenceManager.createEndpointReference interface.
- A deprecated class, com.ibm.websphere.wsaddressing.EndpointReferenceCoverter, for converting EndpointReference instances into deprecated classes com.ibm.websphere.wsaddressing.W3CEndpointReference or com.ibm.websphere.wsaddressing.SubmissionEndpointReferences, for use in JAX-WS applications.

  Note: These classes are deprecated in favour of the JAX-WS 2.1 classes of the same name (EndpointReferenceConverter, SubmissionEndpointReference, and W3CEndpointReference) contained in the com.ibm.websphere.wsaddressing.jaxws21 and javax.xml.ws.wsaddressing.jaxws21 packages, as shown on the diagram.
- A method, com.ibm.websphere.wsaddressing.EndpointReference.setReferenceParameter, to enable you to associate reference parameters with an EndpointReference instance.
- An interface to enable a client to configure its BindingProvider request context, or Stub or Call object, based on an EndpointReference instance. All invocations on the BindingProvider, Stub or Call object are subsequently targeted at the endpoint that is represented by the EndpointReference instance. To achieve this behavior, set the com.ibm.websphere.wsaddressing.WSAConstants.WSADDRESSING_DESTINATION_EPR property on the BindingProvider request context, or Stub or Call object, to the appropriate EndpointReference instance.
- A mechanism for acquiring individual reference parameters that are associated with the incoming message context, to correlate the message to a specific resource instance through the com.ibm.websphere.EndpointReferenceManager.getReferenceParameterFromMessageContext interface.

Using the IBM proprietary Web Services Addressing SPIs: Performing more advanced Web Service Addressing tasks

This product provides proprietary system programming interfaces for more advanced Web Services Addressing (WS-Addressing) tasks, which involve the WS-Addressing message-addressing properties that are passed in the SOAP header of a Web service message. You can also use the SPIs to choose a WS-Addressing specification level other than the default used by the product.

Before you begin

You cannot use the standard JAX-WS 2.1 API classes with these proprietary SPIs. However, you can convert endpoint references created using the standard JAX-WS 2.1 API classes to instances of the com.ibm.websphere.wsaddressing.EndpointReference class, using the com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceConverter class. You can then use these converted endpoint references with the SPIs.

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Perform this task to specify or acquire WS-Addressing message-addressing properties, or if you have an application that needs to interoperate with a client or endpoint that is not using the default WS-Addressing specification supported by this product.

- To manipulate message-addressing properties, follow the instructions in “Specifying and acquiring message-addressing properties using the IBM proprietary Web Services Addressing SPIs” on page 700
- To interoperate with the pre-W3C specification of WS-Addressing, with the namespace http://schemas.xmlsoap.org/ws/2004/08/addressing, refer to “Interoperating with Web Services Addressing endpoints that do not support the default specification supported by WebSphere Application Server” on page 701.

Specifying and acquiring message-addressing properties using the IBM proprietary Web Services Addressing SPIs

Using the proprietary Web Services Addressing (WS-Addressing) system programming interfaces (SPIs), you can add WS-Addressing message addressing properties (MAPs) to the SOAP headers of an outbound client message, through properties on the JAX-WS BindingProvider request context, or the JAX-RPC Stub or Call object. When the target endpoint receives the message, the SPI enables the endpoint to acquire the MAPs through properties on the message context.

About this task

There are no equivalent SPIs in the JAX-WS 2.1 standard. If you want to set message-addressing properties in a client that uses JAX-WS 2.1 endpoint references, you must convert the endpoint references to the IBM proprietary classes, before using them with these SPIs.

Perform this task if you are a Web service developer using the WS-Addressing support, or a system programmer using the IBM proprietary WS-Addressing SPIs to specify message addressing properties, such as fault or reply endpoint references, on Web services messages.

The properties that you can set or retrieve are described, with the Java type of property instances, in “IBM proprietary Web Services Addressing SPIs” on page 702. Most properties are of type com.ibm.websphere.wsaddressing.EndpointReference, for example destination, reply, or fault endpoint references. The relationship property is a java.util.Set object that contains instances of the com.ibm.wsspi.wsaddressing.Relationship class. Use relationships when you want to specify an association between messages; for example, in a response message you might want to specify the ID of the message to which you are replying. The action property is an AttributedURI object that identifies a specific method or operation within the target endpoint.

Note: The destination endpoint reference and action properties are required for the message to be WS-Addressing compliant.

1. On the client, obtain the endpoint reference from the service and associate it with your BindingProvider object’s request context, or your Stub or Call object, as described in “Creating a JAX-RPC Web service application that uses Web Services Addressing” on page 688.
2. Create instances of the required properties. For example, if you want to specify an endpoint reference for the target service to send replies to, create an instance of the com.ibm.websphere.wsaddressing.EndpointReference class, to use as the WSADDRESSING_REPLYTO_EPR property.
3. Set the required properties by associating them with the BindingProvider object’s request context, or the Stub or Call object. If you are using a Stub or Call object, use the setProperty(String property_name, Object value) method. Note that unlike the endpoint reference required for the first step, these endpoint references do not need to be converted to another type, because they are passed in the header of the SOAP message rather than the body. The following example sets a destination endpoint reference and a reply endpoint reference on a BindingProvider object’s request context:

```
import javax.xml.ws.BindingProvider;
...
 javax.xml.ws.Service jaxwsServiceObject=...;
 Printer myPrinterProxy = jaxwsServiceObject.getPort(portName, Printer.class);

 javax.xml.ws.BindingProvider myBindingProvider = (javax.xml.ws.BindingProvider)myPrinterProxy;

 // Retrieve the request context for the BindingProvider object
 Map myMap = myBindingProvider.getRequestContext();

 // Associate the endpoint reference for the Web service. This property is required for the message
 // to be WS-Addressing compliant.
```

```
myMap.put(WSADDRESSING_DESTINATION_EPR, destinationEpr);

// Associate the endpoint reference that represents the reply to the request context
myMap.put(WSADDRESSING_REPLYTO_EPR, replyToEpr);
```

When an invocation occurs on the BindingProvider, Stub, or Call object, the product adds the appropriate MAPs to the message header.

4. On the server, retrieve the MAPs from the inbound message through the javax.xml.ws.WebServiceContext or javax.xml.rpc.handler.MessageContext object that is currently on the thread. When WebSphere Application Server receives the message, it puts the MAP information into the message context on the thread, making it available to the service. You can retrieve the message context by, for example, using the session context of the endpoint enterprise bean. For more information about message contexts, refer to the JSR-109 standard. The following example retrieves the reply endpoint reference using the Web service context:

```
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.WebServiceContext;
...

  // Obtain the message context from the WebService context
  private WebServiceContext wsContext;
  MessageContext context = wsContext.getMessageContext();

  // Retrieve the reply endpoint reference
  replyToEpr = context.getProperty(WSADDRESSING_INBOUND_REPLYTO_EPR);
```

Interoperating with Web Services Addressing endpoints that do not support the default specification supported by WebSphere Application Server

A target Web service endpoint might not support the same Web Services Addressing (WS-Addressing) namespace as this product. In most cases, you do not need to perform any extra actions to interoperate with such endpoints, however some scenarios require additional steps in the implementation of your Web service.

About this task

WebSphere Application Server supports the default WS-Addressing 2005/08 namespace http://www.w3.org/2005/08/addressing. Perform this task when you want to interoperate with endpoints that support other namespaces. This task specifically describes interoperation with endpoints that are hosted on a node that supports only the 2004/08 namespace: http://schemas.xmlsoap.org/ws/2004/08/addressing.

If you are using the standard JAX-WS 2.1 API, ensure that you use the appropriate feature, annotation or endpoint reference class for the 2004/08 namespace.

If you are sending to or receiving messages from an endpoint that supports only the 2004/08 namespace, you do not have to perform any additional steps for interoperability. This product recognizes and understands incoming WS-Addressing messages that conform to the 2004/08 specification, and outbound messages automatically adhere to the namespace of their destination endpoint reference. If you are sending a request, all WS-Addressing elements, such as [reply endpoint] or [fault endpoint] elements, must use the same namespace as the message. Any discrepancy results in a JAX-WS or JAX-RPC configuration error.

If you are interacting in a different way with an endpoint that supports only the 2004/08 namespace, such as exporting endpoint references in the message header or body, and you are not using the JAX-WS 2.1 standard API, you must perform additional steps as detailed below.

- If you are generating a Web service for use by a client that supports only the 2004/08 specification, update the WS-Addressing namespace in the Web Services Description Language (WSDL) document for your Web service, by changing `http://www.w3.org/2006/05/addressing/wsdl` to `http://schemas.xmlsoap.org/ws/2004/08/addressing`.

Note: Only the WS-Addressing WSDL Action extensibility element is recognized by pre-W3C WS-Addressing clients.

- If you are creating endpoint references at run time for export to an endpoint that supports the 2004/08 namespace only, perform the following steps:
  1. Create the endpoint reference to export.
  2. Associate the appropriate namespace with the endpoint reference, using the setNamespace method. The following example illustrates the association of the 2004/08 namespace with an endpoint reference:

```
import com.ibm.wsspi.wsaddressing.EndpointReference;
import com.ibm.wsspi.wsaddressing.NamespaceNotSupportedException;
import com.ibm.wsspi.wsaddressing.WSAConstants;

:

EndpointReference epr = ...

try
{
    epr.setNamespace(WSAConstants.WSADDRESSING_NAMESPACE_2004_08);
} catch (NamespaceNotSupportedException e)
{
    // Error handling code here
}
```

  When you pass the endpoint reference to the target endpoint, in either the SOAP body or the SOAP header of a message, the endpoint reference is appropriately serialized into SOAP elements according to its namespace.
- To establish the namespace of an inbound request, use the IBM proprietary WS-Addressing system programming interface (SPI) to retrieve the WSADDRESSING_INBOUND_NAMESPACE property from the inbound message context. This property specifies the Core WS-Addressing specification namespace of the incoming message.

  Note: This procedure uses the IBM proprietary WS-Addressing API. There is no equivalent procedure in the JAX-WS 2.1 API.

  You can retrieve the message context by, for example, using the session context of the endpoint enterprise bean. For more information about message contexts, refer to the JSR-109 specification. The following code example shows how you can establish the namespace of an incoming message on the receiving endpoint:

```
import com.ibm.wsspi.wsaddressing.WSAConstants;
import javax.xml.rpc.handler.MessageContext;

:
  // If the endpoint is implemented as an enterprise bean, you can use its session context
  // to obtain the message context
  private SessionContext sessionContext;
  MessageContext context = sessionContext.getMessageContext();

  try
  {
    String namespace = (String)msgContext.getProperty(WSAConstants.WSADDRESSING_INBOUND_NAMESPACE);
  } catch (IllegalArgumentException e)
  {
    // Error handling code here
  }
```

IBM proprietary Web Services Addressing SPIs

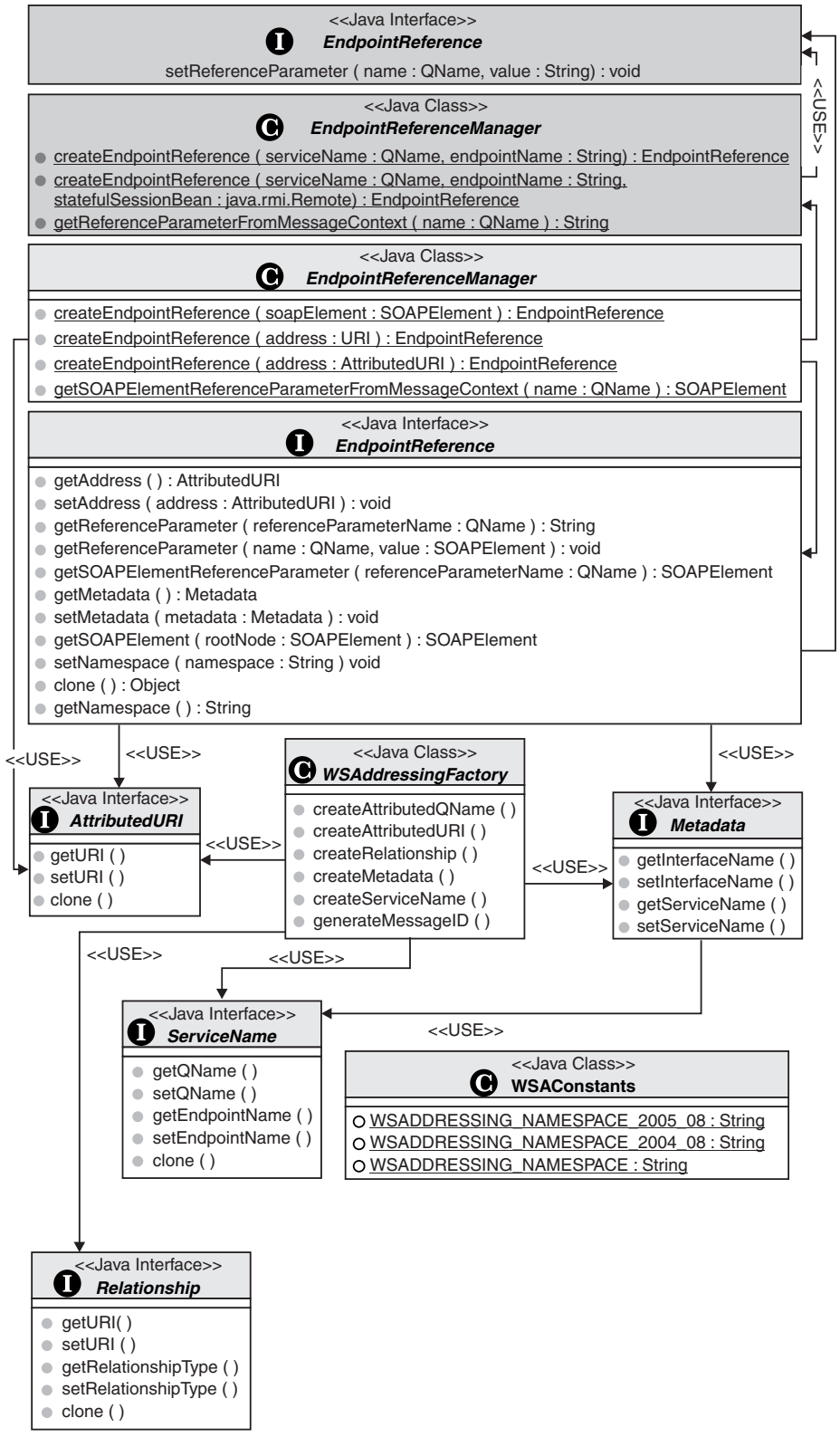
The IBM proprietary Web Services Addressing (WS-Addressing) system programming interfaces (SPIs) extend the IBM proprietary WS-Addressing application programming interfaces (APIs) to enable you to create and reason about the contents of endpoint references and other WS-Addressing artifacts, and to set or retrieve WS-Addressing message-addressing properties (MAPs) on or from Web service messages.

You cannot use the standard JAX-WS 2.1 API classes with these proprietary SPIs. However, you can convert endpoint references created using the standard JAX-WS 2.1 API classes to instances of the proprietary com.ibm.websphere.wsaddressing.EndpointReference class, using the com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceConverter class. You can then use these converted endpoint references with the proprietary SPIs.

The programming interfaces in this topic are described in more detail in the IBM WS-Addressing SPI documentation.

Creating, refining, and reasoning about the contents of endpoint references

The proprietary SPIs for creating, refining, and reasoning about the contents of endpoint references are contained in the com.ibm.wsspi.wsaddressing package and are summarized in the following illustration (the first two interfaces are proprietary API interfaces that are extended by the SPIs):



The SPI extends the proprietary WS-Addressing com.ibm.websphere.wsaddressing.EndpointReference API to provide a number of additional methods through the com.ibm.wsspi.wsaddressing.EndpointReference interface. You can cast instances of com.ibm.websphere.wsaddressing.EndpointReference to com.ibm.wsspi.wsaddressing.EndpointReference to access this additional functionality.

Similarly, the SPI com.ibm.wsspi.wsaddressing.EndpointReferenceManager extends the functionality that is provided in the com.ibm.websphere.wsaddressing.EndpointReferenceManager API.

You can perform the following actions using the additional methods that are provided by the EndpointReference and EndpointReferenceManager SPIs:

Create endpoint references

Create EndpointReference objects by specifying the URI of the endpoint that the EndpointReference object is to represent, using the createEndpointReference(URI) operation, or the EndpointReferenceManager.createEndpointReference(AttributedURI) operation. These methods differ from the createEndpointReference method that is provided at the API level, in that they do not automatically generate the URI for the EndpointReference instance. You might use these methods when you know that the URI of the endpoint is stable, for example in a test environment with no deployment considerations.

Map between XML and Java representations of an endpoint reference

You can serialize instances of the EndpointReference interface to their corresponding SOAP element instances using the EndpointReference.getSOAPElement operation. Conversely, you can deserialize SOAP elements of type EndpointReferenceType into their corresponding EndpointReference Java representation, by using the EndpointReference.createEndpointReference(SOAPElement) operation. You might find these serialization and deserialization interfaces useful if you are creating custom binders for types that contain EndpointReference instances.

Use more complex reference parameter types

The proprietary interfaces that are provided at the API level are restricted to reference parameters of type xsd:string to allow for a simpler programming model. The SPIs extend this support to allow reference parameters of type <xsd:any>. The EndpointReference interface provides mechanisms for getting and setting reference parameters as SOAP elements. Additionally, the EndpointReferenceManager class provides the getSOAPElementReferenceParameterFromMessageContext operation, which enables receiving endpoints to acquire reference parameters that are not of type String from the incoming message.

Note: When invoking a service with an EndpointReference object that contains a reference parameter, you must create the reference parameter using a complete QName object, with all parts present: namespace, localpart, and prefix. If the QName object is not complete, service invocations fail.

Set and reason about endpoint reference contents

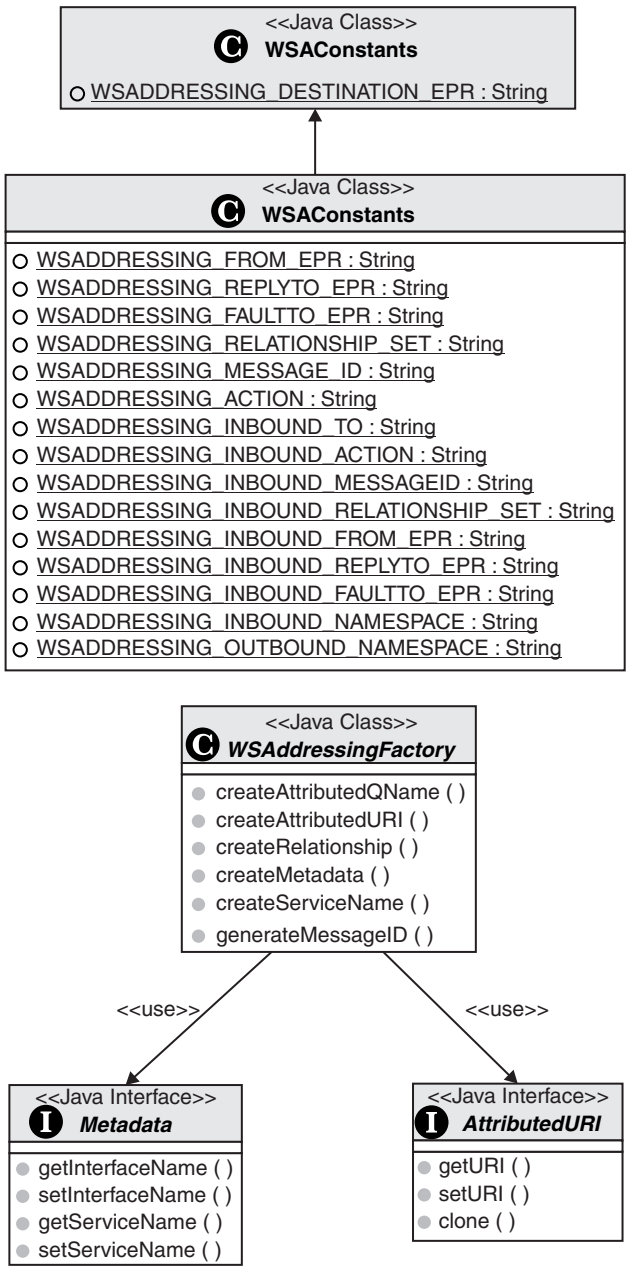
The EndpointReference interface provides operations for you to set and reason about the contents of an EndpointReference instance, such as its WS-Addressing address and metadata properties. Additional interfaces are provided to represent the artifacts making up an endpoint reference: Metadata, AttributedURI, and ServiceName. You create instances of these interfaces using operations that are provided by the proprietary WSAddressingFactory class.

Acquire and change the supported namespace

The WS-Addressing support in this product supports multiple namespaces. The setNamespace and getNamespace operations that are provided on the proprietary EndpointReference interface enable you to change and acquire the namespace that is associated with a particular EndpointReference object. Serialization to SOAP elements is in accordance with the namespace of the EndpointReference object. By default, the namespace of the destination endpoint reference (the endpoint reference set as the com.ibm.websphere.wsaddressing.WSAConstants.WSADDRESSING_DESTINATION_EPR property on the JAX-WS BindingProvider object's request context or the JAX-RPC Stub or Call object), defines the namespace of the message-addressing properties of the message.

Setting and Retrieving WS-Addressing message-addressing properties

The IBM proprietary WS-Addressing SPI provides a number of constants that identify JAX-WS or JAX-RPC properties that you can use to set WS-Addressing MAPs on outbound messages, and message context properties that you can use to retrieve MAPs on inbound messages. These constants are shown in the following diagram in the com.wsspi.wsaddresssing.WSAConstants class. The diagram also shows the interfaces that are required for generating instances of the appropriate property value types AttributedURI and Relationship. The first WSAConstants interface is a proprietary API interface.



Setting WS-Addressing message-addressing properties on outbound messages

You can add WS-Addressing message information headers to outgoing messages by setting the appropriate properties on the JAX-WS BindingProvider object's request context, or the JAX-RPC Stub or

Call object, prior to invoking a message with the BindingProvider, Stub, or Call object. The following table summarizes the relevant properties and their types.

Table 18. Outbound properties that you can set on the BindingProvider object's request context (or the Stub or Call object), their Java types and equivalent abstract WS-Addressing MAP name or names.

Property name (of type String)	**Java type of property value**	**Abstract WS-Addressing MAP name or names**	**Default value**
WSADDRESSING_ DESTINATION_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[destination] URI [reference parameters]* (any)	Not set Note that this property comes from the API.
WSADDRESSING_ FROM_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[source endpoint]	Not set
WSADDRESSING_ REPLYTO_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[reply endpoint]	Either 'none', if the message is a one-way message with no reply, or not set. For two-way asynchronous messages in JAX-WS applications, this property is generated automatically. If, in this situation, you attempt to set this property, a javax.xml.ws.WebServiceException exception is thrown. This exception is also thrown for two-way synchronous messages that do not use the anonymous URI.
WSADDRESSING_ FAULTTO_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[fault endpoint]	Not set If you attempt to set this property for two-way asynchronous messages in JAX-WS applications, a javax.xml.ws.WebServiceException exception is thrown. This exception is also thrown for two-way synchronous messages that do not use the anonymous URI.
WSADDRESSING_ RELATIONSHIP_SET	java.util.Set containing instances of com.ibm.wsspi.wsaddressing. Relationship	[relationship]	Not set
WSADDRESSING_ MESSAGE_ID	com.ibm.wsspi.wsaddressing. AttributedURI	[message id]	Generated and set to a unique value
WSADDRESSING_ ACTION	com.ibm.wsspi.wsaddressing. AttributedURI	[action]	Generated and set, according to the WS-Addressing specification

Table 18. Outbound properties that you can set on the BindingProvider object's request context (or the Stub or Call object), their Java types and equivalent abstract WS-Addressing MAP name or names. (continued)

Property name (of type String)	Java type of property value	Abstract WS-Addressing MAP name or names	Default value
WSADDRESSING_ OUTBOUND_ NAMESPACE	String	none	The WS-Addressing namespace of the WSADDRESSING_ DESTINATION_EPR property, if specified, otherwise the default namespace

Retrieving WS-Addressing message-addressing properties from inbound messages

WS-Addressing message information headers that correspond to the last inbound message are available from the inbound properties that are defined in the WSAConstants class. The following table summarizes the available inbound properties. You acquire reference parameters from the message context using the proprietary EndpointReferenceManager.getReferenceParameter interface.

Table 19. Inbound properties that you can acquire from the message context, their Java types and equivalent abstract WS-Addressing MAP name.

Message context property name (of type String)	Java type of property value	Abstract WS-Addressing MAP name
WSADDRESSING_INBOUND_TO	com.ibm.wsspi.wsaddressing. AttributedURI	[destination]
No specific property. Use the EndpointReferenceManager. getReferenceParameter(QName name) method to obtain the associated MAP.	Any	[reference parameters]*
WSADDRESSING_INBOUND_FROM_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[source endpoint]
WSADDRESSING_INBOUND_REPLYTO_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[reply endpoint]
WSADDRESSING_INBOUND_FAULTTO_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[fault endpoint]
WSADDRESSING_INBOUND_RELATIONSHIP	java.util.Set containing instances of com.ibm.wsspi.wsaddressing. Relationship	[relationship]
WSADDRESSING_INBOUND_MESSAGE_ID	com.ibm.wsspi.wsaddressing. AttributedURI	[message id]
WSADDRESSING_INBOUND_ACTION	com.ibm.wsspi.wsaddressing. AttributedURI	[action]
WSADDRESSING_INBOUND_NAMESPACE	String	The WS-Addressing namespace of the incoming message

Enabling Web Services Addressing support for JAX-WS applications

The Web Services Addressing (WS-Addressing) support provides mechanisms to address Web services and provide addressing information in messages. For JAX-WS applications, you can enable WS-Addressing support in several different ways, such as configuring policy sets or using annotations in code.

About this task

Note: This release introduces a new way to enable WS-Addressing. You can now use JAX-WS 2.1 annotations and feature classes to enable WS-Addressing from either the server or the client. You also have more control over the behavior of WS-Addressing when using policy sets; you can specify whether WS-Addressing is enabled and whether to use synchronous, asynchronous, or both messaging patterns.

Perform this task to enable the WS-Addressing support, either as a service provider or as a client of a service provided by another party. This task also describes how to disable the WS-Addressing support, which can improve performance for those applications that do not use WS-Addressing or any protocol that depends on the WS-Addressing support.

For service providers, WS-Addressing support is enabled by default, so you do not have to perform any actions to enable support. However, you can use the enabling mechanisms to modify other WS-Addressing behavior for the service, such as whether WS-Addressing information is required, and what is included in the generated WSDL document.

- Modify the behavior of the WS-Addressing support after the application is deployed by attaching a policy set to the application. Within the policy set, you can configure the WS-Addressing policy type to specify whether WS-Addressing information is required in incoming messages, and whether to use synchronous or asynchronous messaging. You can communicate the WS-Addressing policy configuration to other servers and clients that support WS-Policy, by enabling policy sharing on the server, and by applying the provider policy on the client. Settings set using this method override those set using other methods.
- Modify the behavior of the WS-Addressing support during development of the service by using the Addressing or SubmissionAddressing annotations in the service code. Within each annotation you can specify whether the server requires WS-Addressing information in incoming messages. The presence of the Addressing annotation in the code also adds a UsingAddressing element to any WSDL document that is generated for the service. If you provide your own WSDL document instead of relying on the JAX-WS runtime environment to generate one, you must include the UsingAddressing element yourself, if required.

For service clients, WS-Addressing support is disabled by default. Use one of the following methods to enable WS-Addressing support:

- Specify the UsingAddressing element in the WSDL document for the service. If the service uses the Addressing annotation and the WSDL document is generated from the code, the UsingAddressing element will already exist.
- Use addressing features in the client code. Settings set using this method override those set in the WSDL document for the service.
- Set the com.ibm.websphere.webservices.use.async.mep property on the client request context.
- You can also use any option available to JAX-RPC applications, such as manually adding the UsingAddressing element to the WSDL document, or using the IBM proprietary WS-Addressing SPI to add message-addressing properties to the message request context.

The behavior of the WS-Addressing support is summarized in the following paragraphs.

WebSphere Application Server clients add WS-Addressing headers to messages in the following situations:

- A policy set containing a WS-Addressing policy is attached to a client artifact, and one or both of the following statements are true:
  - The WS-Addressing policy for the client specifies that WS-Addressing is mandatory.
  - The client artifact has provider and client policies applied, policy sharing is enabled on the server, and the policy configuration for the server requires WS-Addressing.

- The client application code uses a WS-Addressing feature class to specify that WS-Addressing is enabled.
- The WSDL document for the service contains the UsingAddressing element, for example if the document was generated from service application code that contains the Addressing annotation. The value of the `required` parameter is irrelevant.

  Note: This statement does not apply to Dispatch clients, because these clients do not use the WSDL document.
- Message-addressing properties are set on the client.
- The com.ibm.websphere.webservices.use.async.mep property is set on the client request context.

If a client does not include WS-Addressing headers in messages, the server generates a fault message in the following situations:

- A policy set containing a WS-Addressing policy is attached to a server artifact, and the WS-Addressing policy specifies that WS-Addressing is mandatory.
- The server application code uses the Addressing or SubmissionAddressing annotation to specify that WS-Addressing is required.
- To modify the behavior of the WS-Addressing support using policy sets, perform the following steps:
  1. Ensure that you have a policy set that contains the WS-Adressing policy type. If you need to create a new policy set, or add the WS-Addressing policy to an existing policy set, refer to Managing policy sets using the administrative console for instructions.
  2. Configure the WS-Addressing policy type according to the instructions in Configuring the WS-Addressing policy. You can specify whether WS-Addressing is mandatory, and whether to use a synchronous or asynchronous message exchange pattern. The default settings are that WS-Addressing is not mandatory, and both synchronous and asynchronous messaging patterns are used.
  3. Attach the policy set to a Web service provider or client artifact, according to the instructions in Attaching a policy set to a service artifact.
  4. Optional: If you want to communicate the WS-Addressing policy settings to other servers and clients, configure policy sharing as described in Configuring a service provider to share its policy configuration or Configuring the client policy using a service provider policy. If policy sharing is enabled and the server and client cannot agree a policy, normal WS-Policy behavior applies (a policy error is produced).
- To modify the behavior of the WS-Addressing support programmatically in the service application, use one of the following addressing annotations with up to two optional parameters, which specify whether WS-Addressing is enabled and whether WS-Addressing is required. The default settings are that WS-Addressing is enabled but not required.
  - @Addressing: use this annotation if you want to use the 2005/08 WS-Addressing specification
  - @SubmissionAddressing: - use this annotation if you want to use the 2004/08 WS-Addressing specification.

  In the following example, the Addressing annotation is used with no parameters, so the default settings apply.

```
import javax.xml.ws.soap.Addressing;

@Addressing
@WebService(endpointInterface =
   "org.apache.axis2.jaxws.calculator.Calculator",
   serviceName = "CalculatorService",
   portName = "CalculatorServicePort",
   targetNamespace = "http://calculator.jaxws.axis2.apache.org")
```

  In the following example, the SubmissionAddressing annotation is used with parameters that specify that WS-Addressing is enabled and required.

```
import javax.xml.ws.soap.Addressing;

@SubmissionAddressing(enabled="true", required="true")
@WebService(endpointInterface =
   "org.apache.axis2.jaxws.calculator.Calculator",
   serviceName = "CalculatorService",
   portName = "CalculatorServicePort",
   targetNamespace = "http://calculator.jaxws.axis2.apache.org")
```

The server processes any WS-Addressing headers that conform to this relevant specification in inbound SOAP messages. If you specify that WS-Addressing is required, and an inbound SOAP message does not include any WS-Addressing headers, or includes WS-Addressing headers that do not conform to the specification indicated by the annotation, the server returns a fault message. For example, if a client sends a message that includes 2004/08 WS-Addressing headers, and the server requires 2005/08 headers, the server returns a fault message.

If you use the Addressing annotation and generate a WSDL document from the code, a UsingAddressing element, with parameters that match those of the annotation, is created in the WSDL document. Clients using this WSDL document will include WS-Addressing information in their messages. The SubmissionAddressing annotation is not understood by current WSDL generation tools. However, the WSDL document does not distinguish between the 2005/08 specification and the 2004/08 specification, so if you want to generate a WSDL document from code that contains a SubmissionAddressing annotation, you can do so by using both the Addressing and SubmissionAddressing annotations together.

Note:

- You can use the Addressing annotation only with a SOAP (1.1 or 1.2) over HTTP binding. If you use the class with another binding, such as XML over HTTP, an exception is thrown on clients, and on servers the Web service fails to deploy.
- Annotation settings override settings in the WSDL document. Annotation settings might differ from WSDL settings if you create the WSDL document manually rather than generating it from the code.
- If you generate a WSDL document from this code, the UsingAddressing element generated currently has a namespace prefix of wsaw, rather than wsam as specified by the WS-Addressing Metadata specification. This result is because the existing WSDL generation tools are not aware of this new specification. The product supports both prefixes, but if you require the wsam prefix, edit the WSDL document manually.

- To enable WS-Addressing support programmatically on the client by using features, create an instance of one of the following addressing feature classes, with up to two optional parameters, which specify whether WS-Addressing is enabled and whether WS-Addressing is required. The default settings are that WS-Addressing is enabled but not required, however the required property is not used by the client.
  - AddressingFeature: use this class if you want to send messages that include WS-Addressing headers that conform to the 2005/08 WS-Addressing specification
  - SubmissionAddressingFeature: use this class if you want to send messages that include WS-Addressing headers that conform to the 2004/08 WS-Addressing specification

  For example, to specify that WS-Addressing is enabled and required, and that the 2005/08 specification should be used, use the following code:

  ```
  AddressingFeature feat = new AddressingFeature(true, required);
  ```

  To specify that WS-Addressing is disabled for the 2004/08 specification, use the following code:

  ```
  SubmissionAddressingFeature feat = new SubmissionAddressingFeature(false);
  ```

  If you specify that WS-Addressing is enabled, the client includes WS-Addressing headers in SOAP messages. The required property is ignored, but could be used by the application to check that the client has created the correct headers. The headers conform to the WS-Addressing specification indicated by the type of feature class used. If the server does not use annotations, or uses policy sets to enable WS-Addressing, the server accepts both the 2005/08 and 2004/08 specifications.

Note:

- You can use the addressing feature classes only with a SOAP (1.1 or 1.2) over HTTP binding. If you use the class with another binding, such as XML over HTTP, an exception is thrown on clients, and on servers the Web service fails to deploy.
- If you use both feature classes, the specification that is used depends on the type of endpoint reference that you also specify. For example, if you specify a W3CEndpointReference object, the specification that is used is the 2005/08 specification. If you do not specify an endpoint reference, the default specification is the 2005/08 specification. If you specify an endpoint reference whose type conflicts with that indicated by the feature class, for example a W3CEndpointReference object with a SubmissionAddressingFeature instance, an error is thrown.

- On WebSphere Application Server clients, you can also enable WS-Addressing support by setting the com.ibm.websphere.webservices.use.async.mep property on the client request context. For more information, see “Invoking JAX-WS Web services asynchronously” on page 540.

Results

WS-Addressing properties are now included in the SOAP message header, and are processed by the server on receipt of the message.

Enabling Web Services Addressing support for JAX-RPC applications

The Web Services Addressing (WS-Addressing) support provides mechanisms to address Web services and provide addressing information in messages. To enable the WS-Addressing support for JAX-RPC applications, either configure the Web Services Description Language (WSDL) file for a service that runs on WebSphere Application Server, or use the WS-Addressing application programming interface (API) or system programming interface (SPI) to add WS-Addressing properties in a WebSphere Application Server client.

About this task

Perform this task to enable the WS-Addressing support, either as a service provider or as a client of a service provided by another party. This task also describes how to disable the WS-Addressing support, which can improve performance for those applications that do not use WS-Addressing or any protocol that depends on the WS-Addressing support.

If you are creating a Web service, you can enable the WS-Addressing support during development of the service, by including the UsingAddressing extensibility element in the WSDL binding element for the service. This element contains a `required` attribute that has a value of either `false`, which specifies that WS-Addressing information is accepted but not required in incoming messages, or `true`, which specifies that WS-Addressing information is required in incoming messages. The default value is `false`. Messages from WebSphere Application Server Version 7.0 clients always include WS-Addressing information if your service WSDL file includes the UsingAddressing element, regardless of the value of the `required` attribute.

If you are creating a client application to use a service from another provider, you might not have access to the WSDL file for the service, or the service might use a version of WSDL that does not support the UsingAddressing element (if the service is not running on a current version of this product). However, you can still enable WS-Addressing support, during run time, by setting WS-Addressing properties on the JAX-RPC Stub or Call object that you use to communicate with the service.

The following table summarizes the behavior of the WS-Addressing support in each of the scenarios mentioned previously.

Table 20. The behavior of the WS-Addressing support in the product

	The WSDL for the service specifies UsingAddressing required = ″false″	**The WSDL for the service specifies UsingAddressing required = ″true″**	**The WSDL for the service does not specify UsingAddressing**
A client sends a message that contains WS-Addressing information	The WS-Addressing information is processed by the product.	The WS-Addressing information is processed by the product.	The WS-Addressing information is processed by the product.
A non-WebSphere Application Server client sends a message that does not contain WS-Addressing information	The message is accepted.	The service returns a fault.	The message is accepted.
A WebSphere Application Server client sends a message, without specifying addressing properties	The message automatically contains the mandatory WS-Addressing information, as defined in the WS-Addressing specification. The information is processed by the product.	The message automatically contains the mandatory WS-Addressing information, as defined in the WS-Addressing specification. The information is processed by the product.	WS-Addressing information is not added. The message is accepted.

- To enable WS-Addressing support from the server by configuring the WSDL file, perform the following steps:
  1. Ensure that the WSDL file for the service contains the UsingAddressing extensibility element on the binding element. If you generated the WSDL file using the Java2WSDL tool, this element is automatically added for you. If you created the WSDL file yourself, for use with the WSDL2Java tool, you must add the extensibility element. The UsingAddressing element has a `required` attribute with a default value of `false`. For example:

     ```
     <wsdl:binding name="TestServiceSoapBinding" type="intf:TestService">
          <wsaw:UsingAddressing wsdl:required="false"
                                xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"/>

          <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

          <wsdl:operation name="invokeInstance">
            ...
          </wsdl:operation>

       </wsdl:binding>
     ```

     This code indicates that the endpoint will process WS-Addressing information, but that this information is not required.
  2. Optional: To specify that WS-Addressing information is required, change the value of the `required` attribute to `true`. If the endpoint receives a message that does not contain the mandatory WS-Addressing elements within the message header, the endpoint returns a fault message, as defined in the WS-Addressing specification.

     WebSphere Application Server clients and Proxy Server for IBM WebSphere Application Server always send WS-Addressing conformant messages to endpoints with bindings that specify the UsingAddressing element.
- To enable WS-Addressing support from a WebSphere Application Server client, use the IBM proprietary WS-Addressing API or SPI to associate one or more WS-Addressing properties with the JAX-RPC Stub or Call object that is used to send messages to the endpoint.

These properties become message-addressing properties (MAPs) in the SOAP message header. If the node that receives the message is a WebSphere Application Server node, it processes the incoming MAPs in accordance with the WS-Addressing specification, even if the service does not have a UsingAddressing element in its WSDL file.

Use this method when communicating with endpoints that use earlier versions of the WS-Addressing specification (for example: http://schemas.xmlsoap.org/ws/2004/08/addressing) that do not support the UsingAddressing element, or when the WSDL file for the target endpoint is not available to the client.

Results

WS-Addressing properties are now included in the SOAP message header, and are processed by the server on receipt of the message.

Disabling Web Services Addressing support

The Web Services Addressing (WS-Addressing) support provides mechanisms to address Web services and provide addressing information in messages. WS-Addressing support is disabled by default on clients. The method for disabling WS-Addressing support on servers depends on whether your application is based on JAX-RPC or JAX-WS.

About this task

You do not need to disable WS-Addressing support even if your application does not require it, because in most cases WS-Addressing support does not have a negative impact on the running of applications. For JAX-RPC applications, disabling WS-Addressing support can be risky as this action also disables support for other specifications such as Web Services Atomic Transactions.

- To disable WS-Addressing support for JAX-WS service providers, use both the Addressing and SubmissionAddressing annotations in the service code, with the `enabled` parameter set to `false`. For example:

  ```
  import javax.xml.ws.soap.Addressing;

  @Addressing(enabled="false")
  @SubmissionAddressing(enabled="false")
  @WebService(...)
  ```

- You do not have to take any action to disable WS-Addressing support for JAX-WS clients, because WS-Addressing support is disabled by default. However, you can programmatically specify that WS-Addressing is disabled by using both the AddressingFeature and SubmissionAddressingFeature classes in the client code, with the `enabled` parameter set to `false`. For example:

  ```
  AddressingFeature feat = new AddressingFeature(false);
  SubmissionAddressingFeature feat = new AddressingFeature(false);
  ```

- To disable WS-Addressing support for JAX-RPC service providers or clients, set the `com.ibm.ws.wsaddressingAndDependentsDisabled` system property to `true`. For example:

  ```
  java -Dcom.ibm.ws.wsaddressingAndDependentsDisabled=true ... application_name
  ```

  Note: Use this property with care because applications might require WS-Addressing message addressing properties to function correctly. Setting this property also disables support for the following specifications, which depend on the WS-Addressing support: Web Services Atomic Transactions, Web Services Business Agreement, Web Services Notification and Web Services Reliable Messaging.

Results

By completing this task, you disabled the WS-Addressing support. Disabling WS-Addressing on clients prevents WebSphere Application Server sending WS-Addressing message addressing properties in the SOAP header of outbound Web service messages. Disabling WS-Addressing on servers additionally prevents WebSphere Application Server processing WS-Addressing MAPs in incoming SOAP headers.

Creating stateful Web services using the Web Services Resource Framework

You can implement a stateful Web service as a WS-Resource, and reference it using a WS-Addressing endpoint reference. You develop WS-Resources in the same way as ordinary Web services using the same tools, however, you must perform some additional tasks, as described in this topic.

About this task

Perform this task when you want to create a WS-Resource, which is a combination of a stateful resource and a Web service through which the resource is accessed. To complete this task you must have knowledge of standard Web services development tasks, and the Web Services Resource Framework (WSRF) specifications. For an introduction to the WSRF specifications, read the OASIS WSRF Primer document.

1. Identify or create the resource component for which the WS-Resource provides access. This resource component can either be an existing system or entity, or a new component. You have no constraints on how you implement the resource; it can be a simple Java class, a stateless session enterprise bean, an entity bean backed by a relational database, a Service Data Object (SDO), a Java connector, or any other component.
2. Identify or create a resource properties schema document for the WS-Resource. Use IBM Rational Application Developer for WebSphere, or any XML schema authoring tool, to create an XML schema. The schema defines the XML complexType element for the root element of the resource properties document.
3. Create or generate a WSDL document for the Web service component of the WS-Resource. See “Developing a WSDL file for JAX-RPC applications” on page 576 for information about creating WSDL files.
4. Edit the WSDL file to add a ResourceProperties attribute to the portType element. This attribute identifies the root element of the resource properties document that you created earlier. For example, if a Printer service has a resource properties document with a root element <printer_properties> in the namespace http://example.org/printer, then the wsdl:portType element might look as follows:

   ```
   <wsdl:portType xmlns:pr="http://example.org/printer"
                  xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"
                  name="Printer" wsrf-rp:ResourceProperties="pr:printer_properties">
   ```

5. Provide a means to obtain an EndpointReference that points to the WS-Resource. You might define a wsdl:operation element called Create that returns a wsdl:output message of type EndpointReferenceType. See “Example: Creating a Web service that uses the Web Services Addressing API to access a Web Services Resource (WS-Resource) instance” on page 725 for an example of a CreatePrinter operation that returns an EndpointReference object for a Printer WS-Resource.
6. Define each WSRF-defined operation that the WS-Resource supports as a child element of the wsdl:portType element. For each WSRF-defined operation that is supported by the port type, specify the WS-Addressing action attribute on each wsdl:message element. For example, the GetResourceProperty operation is defined in the WSDL as follows:

   ```
   <wsdl:operation name="GetResourceProperty"
                   xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                   xmlns:wsrf-rpw="http://docs.oasis-open.org/wsrf/rpw-2">
     <wsdl:input name="GetResourcePropertyRequest" message="wsrf-rpw:GetResourcePropertyRequest"
       wsaw:Action="http://docs.oasis-open.org/wsrf/rpw-2/GetResourceProperty/GetResourcePropertyRequest"/>
     <wsdl:output name="GetResourcePropertyResponse" message="wsrf-rpw:GetResourcePropertyResponse"
       wsaw:Action="http://docs.oasis-open.org/wsrf/rpw-2/GetResourceProperty/GetResourcePropertyResponse"/>
     ...
   </wsdl:operation>
   ```

   The wsaw:Action attribute ensures that the WSRF-defined wsaw:Action URIs are used for the WSRF-defined messages, rather than default URI values.

Note: The WS-ResourceProperties specification requires the presence of the GetResourceProperty operation if the ResourceProperties attribute is present on the PortType element.

7. Follow the instructions from step 2 in “Creating a JAX-RPC Web service application that uses Web Services Addressing” on page 688 to create the implementation of the WS-Resource, enable the client to access the WS-Resource using an endpoint reference, and deploy the application.

What to do next

Review “Example: Creating a Web service that uses the Web Services Addressing API to access a Web Services Resource (WS-Resource) instance” on page 725 for sample WS-Resource code.

Web Services Resource Framework support

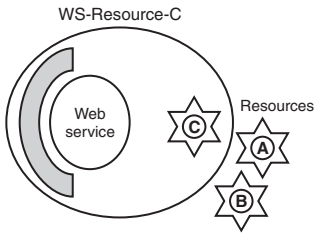
The Web Services Resource Framework (WSRF) support in WebSphere Application Server provides the environment for Web service applications that follow the OASIS WSRF specifications.

WSRF overview

Web service interfaces often need to provide stateful interactions with the clients of the service. For example, a Web service interface such as a shopping cart, where the result of one operation influences the carrying out of the succeeding operations. The OASIS Web Services Resource Framework (WSRF) defines a generic framework for modelling and accessing stateful resources using Web services, so that the definition and implementation of a service and the integration and management of multiple services is easier.

WSRF introduces the concept of an XML document description, called the *resource properties document schema*, which is referenced by the WSDL description of a Web service and which explicitly describes a view of the state of the resource with which the client interacts. A service described in this way is called a *WS-Resource*.

A WS-Resource is defined as the combination of a resource and a Web service through which the resource is accessed. The following figure illustrates a Web service, at http://www.example.com/service, and three resources, A, B, and C, which are accessed through the Web service. Three WS-Resources are therefore illustrated in the figure:



A WS-Resource is referenced by a WS-Addressing endpoint reference that uniquely identifies the WS-Resource, typically by containing an identifier of the resource component of the WS-Resource inside the EndpointReference ReferenceParameter element. In the previous example, WS-Resource-C is the combination of the Web service and the resource that is identified by C, and a reference to WS-Resource-C might be as follows:

```
<wsa:EndpointReference>
    <wsa:Address>
        http://www.example.com/service
    </wsa:Address>
    <wsa:ReferenceParameters>
```

```
        <tns:SomeDisambiguatorElement>C</tns:SomeDisambiguatorElement>
    </wsa:ReferenceParameters>
    ...
</wsa:EndpointReference>
```

Each such WS-Resource has a resource property document (an XML instance document) that describes a view of the state of the resource. The WSDL for a WS-Resource identifies the XML schema that describes the type of the resource property document through a ResourceProperties attribute of the wsdl:PortType element. By specifying this standard WSDL extension for the resource properties document schema, WSRF enables the definition of simple, generic messages that interact with the WS-Resource.

For example, consider a Printer WS-Resource that has the following resource properties document schema:

```
<?xml version="1.0"?>
<xsd:schema ...
   xmlns:pr="http://example.org/printer.xsd"
   targetNamespace="http://example.org/printer.xsd" >
 <xsd:element name="printer_properties">
   <xsd:complexType>
     <xsd:sequence>
        <xsd:element ref="pr:printer_name" />
        <xsd:element ref="pr:queued_job_count" />
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 ...
</schema>
```

The WSDL PortType element for such a WS-Resource declares the Resource Properties Document type as follows:

```
<wsdl:portType xmlns:pr="http://example.org/printer.xsd"
               xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"
               name="Printer" wsrf-rp:ResourceProperties="pr:printer_properties">
```

Each WS-Resource has a unique, logical resource properties document instance that is a view of the state of the resource. The WS-ResourceProperties specification describes the interoperable protocol messages that a WS-Resource can implement to get, set, or query the state of the resource by operating on the resource properties document. Some of these operations affect the resource properties document as a whole, and some of them operate on one or more elements within the document (the individual resource properties, for example pr:printer_name). Each WS-Resource can have a finite lifecycle and can be created and destroyed; the WS-ResourceLifetime specification describes the interoperable protocol messages that a WS-Resource can implement to destroy itself or to alter its termination time.

For more information about WSRF, refer to the WSRF Primer document published by the OASIS Technical Committee.

WSRF Programming Model

The WSRF specifications define only the protocol messages and the semantic behavior that is expected of a WS-Resource when it processes these messages; the specifications do not prescribe the means to implement WS-Resource objects. WSRF is primarily an application-level protocol and the tools for implementing WS-Resources are the same tools that are used for implementing any other type of Web service. WSRF uses WS-Addressing endpoint references and the programming model for WS-Resources is similar to the model for any Web service that uses WS-Addressing; this model is described in “Web Services Addressing application programming model” on page 674.

WSRF extends the WebSphere Application Server WS-Addressing programming model in two ways, which differentiate a WS-Resource from a generic resource that is accessed through a Web service using WS-Addressing:

- WSRF requires the ResourceProperties attribute on the wsdlPortType element. This attribute declares that the portType element is implemented by a WS-Resource rather than a generic Web service. The WS-Resource must declare which WSRF operations it supports by copying those operations into the portType element of its WSDL definition. The WS-Resource is free to choose any implementation strategy to represent the stateful resource and to process the WSRF messages; you can implement a resource using a simple Java class, a stateless session enterprise bean, an entity bean backed by a relational database, a Service Data Object (SDO), and so on.
- WSRF defines a hierarchy of Java BaseFault types.

Web Services Resource Framework base faults

The Web Services Resource Framework (WSRF) provides a recommended basic fault message element type from which you can derive all service-specific faults. The advantage of a common basic type is that all faults can, by default, contain common information. This behavior is useful in complex systems where faults might be systematically logged, or forwarded through several layers of software before being analyzed.

The common information includes the following items:

- A mandatory timestamp.
- An element that can be used to indicate the originator of the fault.
- Other elements that can describe and classify the fault.

The following two standard faults are defined for use with every WSRF operation:

ResourceUnkownFault
: This fault is used to indicate that the WS-Resource is not known by the service that receives the message.

ResourceUnavailableFault
: This fault is used to indicate that the Web service is active, but temporarily unable to provide access to the resource.

The following XML fragment shows an example of a base fault element:

```
<wsrf-bf:BaseFault>
   <wsrf-bf:Timestamp>2005-05-31T12:00:00.000Z</wsrf-bf:Timestamp>
   <wsrf-bf:Originator>
     <wsa:Address>
       http://www.example.org/Printer
     </wsa:Address>
     <wsa:ReferenceParameters>
       <pr:pr-id>P1</pr:pr-id>
     </wsa:ReferenceParameters>
   </wsrf-bf:Originator>
   <wsrf-bf:Description>Offline for service maintenance</wsrf-bf:Description>
   <wsrf-bf:FaultCause>OFFLINE</wsrf-bf:FaultCause>
</wsrf-bf:BaseFault>
```

Note: The elements and classes that are discussed in the rest of this topic apply to JAX-RPC applications only. If your application uses JAX-WS, use the artifacts that are generated, for example by the wsimport tool, from the application WSDL document and XML schema that define and use the specific BaseFault type.

The BaseFault class

For JAX-RPC applications, WebSphere Application Server provides Java code mappings for all the base fault element types that are defined by the WSRF specifications, forming an exception hierarchy where each Java exception extends the com.ibm.websphere.wsrf.BaseFault class. Each fault class follows a similar pattern.

For example, the BaseFault class defines the following two constructors:

```
package com.ibm.websphere.wsrf;
public class BaseFault extends Exception
{
    public BaseFault()
    {
       ...
    }
    public BaseFault(EndpointReference originator,
                     ErrorCode errorCode,
                     FaultDescription[] descriptions,
                     IOSerializableSOAPElement faultCause,
                     IOSerializableSOAPElement[] extensibilityElements,
                     Attribute[] attributes)
    {
       ...
    }
    ...
}
```

The IOSerializableSOAPElement class

Because the BaseFault class extends the java.lang.Exception class, the BaseFault class must implement the java.io.Serializable interface. To meet this requirement, all properties of a BaseFault instance must be serializable. Because the javax.xml.soap.SOAPElement class is not serializable, WebSphere Application Server provides an IOSerializableSOAPElement class, which you can use to wrap a javax.xml.soap.SOAPElement instance to provide a serializable form of that instance.

Create an IOSerializableSOAPElement instance by using the IOSerializableSOAPElementFactory class, as follows:

```
// Get an instance of the IOSerializableSOAPElementFactory class
IOSerializableSOAPElementFactory factory = IOSerializableSOAPElementFactory.newInstance();

// Create an IOSerializableSOAPElement from a javax.xml.soap.SOAPElement
IOSerializableSOAPElement serializableSOAPElement = factory.createElement(soapElement);

// You can retrieve the wrapped SOAPElement from the IOSerializableSOAPElement
SOAPElement soapElement = serializableSOAPElement.getSOAPElement();
```

Any application-specific BaseFault instances must also adhere to this serializable requirement.

Application-specific faults

Applications can define their own extensions to the BaseFault element. Use XML type extensions to define a new XML type for the application fault that extends the BaseFaultType element. For example, the following XML fragment creates a new PrinterFaultType element:

```
 <xsd:complexType name="PrinterFaultType">
   <xsd:complexContent>
     <xsd:extension base="wsrf-bf:BaseFaultType"/>
   </xsd:complexContent>
 </xsd:complexType>
```

The following example shows how a Web service application, whose WSDL definition might define a print operation that declares two wsdl:fault messages, constructs a PrinterFault object:

```
import com.ibm.websphere.wsrf.BaseFault;
import com.ibm.websphere.wsrf.*;
import javax.xml.soap.SOAPFactory;
...
    public void print(PrintRequest req) throws PrinterFault, ResourceUnknownFault
    {
        // Determine the identity of the target printer instance.
```

```
        PrinterState state = PrinterState.getState ();
        if (state.OFFLINE)
        {
            try
            {
                // Get an instance of the SOAPFactory
                SOAPFactory soapFactory = SOAPFactory.newInstance();

                // Create the fault cause SOAPElement
                SOAPElement faultCause = soapFactory.createElement("FaultCause");
                faultCase.addTextNode("OFFLINE");

                // Get an instance of the IOSerializableSOAPElementFactory
                IOSerializableSOAPElementFactory factory = IOSerializableSOAPElementFactory.newInstance();

                // Create an IOSerializableSOAPElement from the faultCause SOAPElement
                IOSerializableSOAPElement serializableFaultCause = factory.createElement(faultCause);

                FaultDescription[] faultDescription = new FaultDescription[1];
                faultDescription[0] = new FaultDescription("Offline for service maintenance");
                throw new PrinterFault(
                         state.getPrinterEndpointReference(),
                         null,
                         faultDescription,
                         serializableFaultCause,
                         null,
                         null);
            }
            catch (SOAPException e)
            {
               ...
            }
        }
        ...
```

The following code shows how base fault hierarchies are handled as Java exception hierarchies:

```
import com.ibm.websphere.wsrf.BaseFault;
import com.ibm.websphere.wsrf.*;
...
try
{
   printer1.print(job1);
}
catch (ResourceUnknownFault exc)
{
   System.out.println("Operation threw the ResourceUnknownFault");
}
catch (PrinterFault exc)
{
   System.out.println("Operation threw PrinterFault");
}
catch (BaseFault exc)
{
   System.out.println("Exception is another BaseFault");
}
catch (Exception exc)
{
   System.out.println("Exception is not a BaseFault");
}
```

Custom binders

When you define a new application-level base fault, for example the PrinterFault fault with the PrinterFaultType type shown previously, you must provide a custom binder to define how the Web services run time serializes the Java class into an appropriate XML message, and conversely how to deserialize an XML message into an instance of the Java class.

The custom binder must implement the com.ibm.wsspi.webservices.binding.CustomBinder interface. Package the binder in a Java archive (JAR) file along with a declarative metadata file, `CustomBindingProvider.xml`, in the `/META-INF/services` directory of the JAR file. This metadata file defines the relationship between the custom binder, the Java BaseFault implementation and the BaseFault type. For example, you might define a custom binder called PrinterFaultTypeBinder, to map between the XML PrinterFaultType element and its Java implementation, PrinterFault, as follows:

```
<customdatabinding:provider
   xmlns:customdatabinding="http://www.ibm.com/webservices/customdatabinding/2004/06"
   xmlns:pr="http://example.org/printer.xsd"
   xmlns="http://www.ibm.com/webservices/customdatabinding/2004/06">
 <mapping>
  <xmlQName>pr:PrinterFaultType</xmlQName>
  <javaName>PrinterFault</javaName>
  <qnameScope>complexType</qnameScope>
  <binder>PrinterFaultTypeBinder</binder>
 </mapping>
</customdatabinding:provider>
```

The BaseFaultBinderHelper class

WebSphere Application Server provides a BaseFaultBinderHelper class, which provides support for serializing and deserializing the data that is specific to a root BaseFault class, which all specialized BaseFault classes must extend. If a custom binder uses the BaseFaultBinderHelper class, the custom binder then needs to provide only the additional logic for serializing and deserializing the extended BaseFault data.

The following code shows how you can implement a custom binder for the PrinterFaultType element to take advantage of the BaseFaultBinderHelper class support:

```
import com.ibm.wsspi.wsrf.BaseFaultBinderHelper;
import com.ibm.wsspi.wsrf.BaseFaultBinderHelperFactory;
import com.ibm.wsspi.webservices.binding.CustomBinder;
import com.ibm.wsspi.webservices.binding.CustomBindingContext;
...

public PrinterFaultTypeBinder implements CustomBinder
{
   // Get an instance of the BaseFaultBinderHelper
   private BaseFaultBinderHelper baseFaultBinderHelper = BaseFaultBinderHelperFactory.getBaseFaultBinderHelper();

   public SOAPElement serialize(Object data, SOAPElement rootNode, CustomBindingContext context) throws SOAPException
   {
     // Serialize the BaseFault specific data
     baseFaultBinderHelper.serialize(rootNode, (BaseFault)data);

     // Serialize any PrinterFault specific data
     ...

     // Return the serialized PrinterFault
     return rootNode;
   }

   public Object deserialize(SOAPElement rootNode, CustomBindingContext context) throws SOAPException
   {
     // Create an instance of a PrinterFault
     PrinterFault printerFault = new PrinterFault();

     // Deserialize the BaseFault specific data - any additional data which
     // forms the PrinterFault extension will be returned as a SOAPElement[].
     SOAPElement[] printerFaultElements = baseFaultBinderHelper.deserialize(printerFault, rootNode);

     // Deserialize the PrinterFault specific data contained within the printerFaultElements SOAPElement[]
```

```
    ...

    // Return the deserialized PrinterFault
    return printerFault;
  }

 ...
}
```

Web Services Resource Framework resource property and lifecycle operations

The Web Services Resource Framework (WSRF) contains specifications that describe the operations that a Web Services Resource (WS-Resource) can implement to get, set, or query the state of the resource by operating on the resource properties document.

For a complete description of all the standard property and lifetime operations that are defined by the Web Services Resource Framework (WSRF), see the WS-ResourceProperties and WS-ResourceLifetime specifications. The principle WSRF operations that a Web Services Resource (WS-Resource) can support are described in the following table.

Table 21. Principle WSRF operations that are supported by WS-Resources

Operation	Description

GetResourcePropertyDocument	Returns the entire resource properties document for the WS-Resource.
	Message format
	`<wsrf-rp:GetResourcePropertyDocument/>`
	Response format
	`<wsrf-rp:GetResourcePropertyDocumentResponse>`
	`  {any}`
	`</wsrf-rp:GetResourcePropertyDocumentResponse>`
	where *{any}* is the content of the resource properties document.

PutResourcePropertyDocument	Replaces the entire resource properties document for the WS-Resource with the document specified.
	Message format
	`<wsrf-rp:PutResourcePropertyDocument>`
	`  {any}`
	`</wsrf-rp:PutResourcePropertyDocument>`
	where *{any}* is the content of the new resource properties document.
	Response format
	`<wsrf-rp:PutResourcePropertyDocumentResponse>`
	`  {any} ?`
	`</wsrf-rp:PutResourcePropertyDocumentResponse>`
	where *{any}* is the content of the new resource properties document. If the content is the same as the requested content, then the *{any}* element must not be present.

Table 21. Principle WSRF operations that are supported by WS-Resources (continued)

Operation	Description

GetResourceProperty	Returns the value or values of the specified resource property found within the resource properties document for the WS-Resource.
	Message format
	`<wsrf-rp:GetResourceProperty>`
	`  QName`
	`</wsrf-rp:GetResourceProperty>`
	Response format
	`<wsrf-rp:GetResourcePropertyResponse>`
	`  {any}*`
	`</wsrf-rp:GetResourcePropertyResponse>`
	where *{any}** is a sequence of elements that match the *QName* specified in the request.

GetMultipleResourceProperties	Returns the value or values of the specified resource properties found within the resource properties document for the WS-Resource.
	Message format
	`<wsrf-rp:GetMultipleResourceProperties>`
	`  <wsrf-rp:ResourceProperty>QName<wsrf-rp:ResourceProperty>+`
	`</wsrf-rp:GetMultipleResourceProperties>`
	Response format
	`<wsrf-rp:GetMultipleResourcePropertiesResponse>`
	`  {any}*`
	`</wsrf-rp:GetMultipleResourcePropertiesResponse>`
	where *{any}** is a sequence of elements that match the *QNames* specified in the request.

InsertResourceProperties	Inserts the resource property elements specified into the resource properties document for the WS-Resource.
	Message format
	`<wsrf-rp:InsertResourceProperties>`
	`  <wsrf-rp:Insert>`
	`  {any}*`
	`  </wsrf-rp:Insert>`
	`</wsrf-rp:InsertResourceProperties>`
	where `{any}*` is a sequence of elements with the same QName.
	Response format
	`<wsrf-rp:InsertResourcePropertiesResponse/>`

Table 21. Principle WSRF operations that are supported by WS-Resources (continued)

Operation	Description

UpdateResourceProperties	Updates the resource property elements specified into the resource properties document for the WS-Resource.
	Message format
	`<wsrf-rp:UpateResourceProperties>`
	`  <wsrf-rp:Upate>`
	`  {any}*`
	`  </wsrf-rp:Upate>`
	`</wsrf-rp:UpateResourceProperties>`
	where *{any}** is a sequence of elements with the same QName.
	Response format
	`<wsrf-rp:UpateResourcePropertiesResponse/>`

DeleteResourceProperties	Deletes the resource property elements specified from the resource properties document for the WS-Resource.
	Message format
	`<wsrf-rp:DeleteResourceProperties>`
	`  <wsrf-rp:Delete ResourceProperty="QName"/>`
	`</wsrf-rp:DeleteResourceProperties>`
	where *QName* is the QName of the resource property to delete.
	Response format
	`<wsrf-rp:DeleteResourcePropertiesResponse/>`

QueryResourceProperties	Query the resource properties document using a query expression, such as XPath.
	Message format
	`<wsrf-rp:QueryResourceProperties>`
	`  <wsrf-rp:QueryExpression`
	`  Dialect="http://www.w3.org/TR/1999/REC-xpath-19991116">`
	`  xsd:any`
	`  </wsrf-rp:QueryExpression>`
	`</wsrf-rp:QueryResourceProperties>`
	where *xsd:any* is the XPath query expression to apply to the resource properties document.
	Response format
	`<wsrf-rp:QueryResourcePropertiesResponse>`
	`  {any}`
	`</wsrf-rp:QueryResourcePropertiesResponse>`
	where *{any}* is the result of evaluating the query expression against the resource properties document.

Destroy	Destroys the WS-Resource.
	Message format
	`<wsrf-rl:Destroy/>`
	Response format
	`<wsrf-rl:DestroyResponse/>`
	This response indicates successful destruction of the WS-Resource.

Table 21. Principle WSRF operations that are supported by WS-Resources (continued)

Operation	Description

SetTerminationTime	WS-Resources that support scheduled termination can implement this operation to allow a requester to change the time at which the WS-Resource destroys itself.
	
	Message format
	`<wsrf-rl:SetTerminationTime>`
	`  [<wsrf-rl:RequestedTerminationTime>`
	`  xsd:dateTime`
	`  </wsrf-rl:RequestedTerminationTime>]`
	`  |`
	`  [<wsrf-rl:RequestedLifetimeDuration>`
	`  xsd:duration`
	`  </wsrf-rl:RequestedLifetimeDuration>]`
	`</wsrf-rl:SetTerminationTime>`
	
	where the termination time is either an absolute time or a relative duration.
	
	Response format
	`<wsrf-rl:SetTerminationTimeResponse>`
	`  <wsrf-rl:NewTerminationTime>`
	`  xsd:dateTime`
	`  </wsrf-rl:NewTerminationTime>`
	`  <wsrf-rl:CurrentTime>`
	`  xsd:dateTime`
	`  </wsrf-rl:CurrentTime>`
	`<wsrf-rl:SetTerminationTimeResponse>`
	
	This response contains the time, from the perspective of the WS-Resource, when the WS-Resource destroys itself. The response also contains the WS-Resource's value of the current time.
	
	A variety of ways exist in which a WS-Resource can implement scheduled destruction. For example, a WS-Resource that is implemented as an enterprise bean might use the enterprise bean container timer service by implementing the ejbTimeout callback method of the javax.ejb.TimedObject interface, and by creating a Timer object that expires at the scheduled destruction time and drives this callback method. EJB timer service Timer objects are persistent and survive server restarts, and are therefore a simple means to manage the lifecycle of WS-Resources that have a finite lifecycle and require a time-based destruction mechanism.

Example: Creating a Web service that uses the Web Services Addressing API to access a Web Services Resource (WS-Resource) instance

This example extends the example ″Creating a Web service that uses the Web Services Addressing API to access a generic Web service resource instance″, to use a WS-Resource instance. A WS-Resource, by definition, is a combination of a resource and a Web service through which the resource is accessed.

Creating a resource properties schema document for the WS-Resource

As described in the WS-Resource specification, which is part of the Web Services Resource Framework (WSRF) specification, a WS-Resource is accessed through a WS-Addressing endpoint reference, and a view on the state of its resource is maintained in a *resources properties* XML document. Use of a WS-Resource, for representing stateful resources, provides an interoperable means to interact with the state representation of resources using standardized web service messages.

A WS-Resource must have a resource properties XML document, described by XML schema, which describes a particular view of the state of the WS-Resource. The printer WS-Resource schema document is illustrated in the following example.

```
<?xml version="1.0"?>
<xsd:schema ...
   xmlns:pr="http://example.org/printer.xsd"
   targetNamespace="http://example.org/printer.xsd" >
 <xsd:element name="printer_properties">
   <xsd:complexType>
     <xsd:sequence>
        <xsd:element ref="pr:printer_reference" />
        <xsd:element ref="pr:printer_name" />
        <xsd:element ref="pr:printer_state" />
        <xsd:element ref="pr:printer_accepting_jobs" />
        <xsd:element ref="pr:queued_job_count" />
        <xsd:element ref="pr:operations_supported" />
        <xsd:element ref="pr:document_format_supported" />
        <xsd:element ref="pr:job_hold_until_default"
                     minOccurs="0" />
        <xsd:element ref="pr:job_hold_until_supported"
                     minOccurs="0"
                     maxOccurs="unbounded" />
        <xsd:element ref="wsrf-rp:QueryExpressionDialect"
                     maxOccurs="unbounded" />
        <xsd:element ref="pr:job_properties" minOccurs="0"
                     maxOccurs="unbounded" />
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 ...
</schema>
```

Creating and editing the WSDL definition for the Web service component of the WS-Resource

The WSDL definition for the Printer WS-Resource server is the same as in “Example: Creating a Web service that uses the IBM proprietary Web Services Addressing API to access a generic Web service resource instance” on page 692, with the addition of a ResourceProperties attribute on the wsdlPortType element. This attribute declares that the port type is implemented by a WS-Resource rather than a generic Web service. Because the interface contains a resource properties document type declaration, the interface must also contain the WSRF-defined GetResourceProperty operation; this operation is required by the WS-ResourceProperties specification.

```
<wsdl:definitions targetNamespace="http://example.org/printer" ...
             xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"
             xmlns:wsrf-rpw="http://docs.oasis-open.org/wsrf/rpw-2"
             xmlns:wsa="http://www.w3.org/2005/08/addressing"
             xmlns:pr="http://example.org/printer">
  <wsdl:types>
    ...
    <xsd:schema...>
      <xsd:element name="CreatePrinterRequest"/>
      <xsd:element name="CreatePrinterResponse"
                   type="wsa:EndpointReferenceType"/>
    <xsd:import namespace="http://www.w3.org/2005/08/addressing"
                schemaLocation="http://www.w3.org/2005/08/addressing/ws-addr.xsd"/>
    <xsd:import namespace=http://docs.oasis-open.org/wsrf/rp-2
                schemaLocation="http://docs.oasis-open.org/wsrf/rp-2.xsd"/>
   </xsd:schema>

   <!-- Import WSDL definitions for GetResourceProperties -->
  <wsdl:import namespace="http://docs.oasis-open.org/wsrf/rpw-2"
               location="http://docs.oasis-open.org/wsrf/rpw-2.wsdl" />

  </wsdl:types>
```

```
  <wsdl:message name="CreatePrinterRequest">
    <wsdl:part name="CreatePrinterRequest"
               element="pr:CreatePrinterRequest" />
  </wsdl:message>
  <wsdl:message name="CreatePrinterResponse">
    <wsdl:part name="CreatePrinterResponse"
               element="pr:CreatePrinterResponse" />
  </wsdl:message>

  <!-- The port type has a ResourceProperties attribute that references the resource
  properties document -->
  <wsdl:portType name="Printer" wsrf-rp:ResourceProperties="pr:printer_properties">
    <wsdl:operation name="createPrinter">
      <wsdl:input name="CreatePrinterRequest"
                  message="pr:CreatePrinterRequest" />
      <wsdl:output name="CreatePrinterResponse"
                   message="pr:CreatePrinterResponse" />
    </wsdl:operation>

    <!-- The GetResourceProperty operation is required by the WS-ResourceProperties specification -->
    <wsdl:operation name="GetResourceProperty"
        <wsdl:input name="GetResourcePropertyRequest"
                    message="wsrf-rpw:GetResourcePropertyRequest"
                    wsa:Action="http://docs.oasis-open.org/wsrf/rpw-2/GetResourceProperty/
                                                                    GetResourcePropertyRequest"/>
        <wsdl:output name="GetResourcePropertyResponse"
                     message="wsrf-rpw:GetResourcePropertyResponse"
                     wsa:Action="http://docs.oasis-open.org/wsrf/rpw-2/GetResourceProperty/
                                                                   GetResourcePropertyResponse"/>
        <wsdl:fault name="ResourceUnknownFault"
                    message="wsrf-rw:ResourceUnknownFault"/>
        <wsdl:fault name="InvalidResourcePropertyQNameFault"
                    message="wsrf-rpw:InvalidResourcePropertyQNameFault" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Implementing the Web service component of the WS-Resource

You implement the Web service in the same way as a normal Web service, as described in “Example: Creating a Web service that uses the IBM proprietary Web Services Addressing API to access a generic Web service resource instance” on page 692. This example discusses the use of endpoint references that refer to generic Web service resource instances. A WS-Resource instance is a specific type of such a resource instance, that supports the standardized message exchanges defined in the WSRF specification.

Assembling Web services applications

You can assemble Java-based Web services applications using assembly tools.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with the application server.

About this task

After you develop your Web service application, you are now ready to assemble the application. Assembling a Web service application consists of creating the Java Platform, Enterprise Edition (Java EE) modules that you can deploy onto application servers. The modules are created from code artifacts such as Web application archives (WAR) files for JavaBeans applications or enterprise beans Java archive (JAR) files for enterprise beans applications. This packaging and configuring of code artifacts into

enterprise application modules (EAR files) or standalone Web modules is necessary for deploying the modules onto an application server.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. Assemble your Web services enabled bean into the appropriate module.
   - For JavaBeans enabled as Web services:
     a. “Assembling a WAR file that is enabled for Web services from Java code” on page 730.
     b. “Assembling a Web services-enabled WAR file from a WSDL file” on page 731.
   - For enterprise beans enabled as Web services:
     a. “Assembling a JAR file that is enabled for Web services from an enterprise bean.”
     b. “Assembling a Web services-enabled enterprise bean JAR file from a WSDL file” on page 729.
3. Assemble the Web services enabled module into an enterprise archive (EAR) file.
   - “Assembling a Web services-enabled WAR into an EAR file” on page 733.
   - “Assembling an enterprise bean JAR file into an EAR file” on page 732.
4. Enable the EAR file for EJB modules that contain Web services. When the EAR file contains Enterprise JavaBeans (EJB) modules that contain Web services, you must run the endptEnabler command-line tool or an assembly tool before deployment to produce a Web services endpoint WAR file. This tool is also used to specify whether the Web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.
5. Assemble a Web services-enabled WAR file into an EAR file.

Results

You have a Web services-enabled EAR file that you can deploy onto the application server.

What to do next

Now you need to deploy the Web services-enabled EAR file onto your application server.

Assembling a JAR file that is enabled for Web services from an enterprise bean

You can assemble a Web service-enabled enterprise bean Java archive (JAR) file with an assembly tool using artifacts generated from tooling.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

You need the following artifacts that are generated from the **WSDL2Java** command-line tool to complete this task:

- An assembled enterprise bean JAR file that is not enabled for Web services
- A compiled Java class for the service endpoint interface
- A Web Services Description Language (WSDL) file
- The complete `webservices.xml`,`ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptor, and Java API for XML-based remote procedure call (JAX-RPC) mapping file.

About this task

Assemble a Web services-enabled enterprise bean JAR file from Java code by following the actions in the steps for this task section.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.

Results

You have the artifacts required to Web service-enable an Enterprise JavaBeans (EJB) module for Web services. The artifacts are added to the JAR file. Now you need to configure the deployment descriptors so that you can deploy the Web service into the application server run time environment.

Example

The `AddressBook.jar` JAR file contains the following files after assembly. The files added in this task are in bold. These files include the WSDL file, the deployment descriptors, and the JAX-RPC mapping file.

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookBean.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

What to do next

Assemble the EAR file so that you can deploy the EAR file into WebSphere Application Server.

Assembling a Web services-enabled enterprise bean JAR file from a WSDL file

You can assemble a Web services-enabled enterprise bean Java archive (JAR) file from a Web Services Description Language (WSDL) file with an assembly tool.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

You need the following artifacts to complete this task:
- An assembled enterprise bean JAR file that contains the Enterprise JavaBeans (EJB) implementation and all classes that generate from the **WSDL2Java** command-line tool when the role argument is `develop-server` and the container argument is `EJB`.
- A WSDL file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file.

About this task

Assemble a Web services-enabled enterprise bean JAR file from a WSDL file by following the actions in the steps for this task section.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.

Results

You have the artifacts required to Web service-enable an EJB module for Web services. The artifacts are added to the JAR file. Now you need to configure the deployment descriptors so that you can deploy the Web service into the application server runtime environment.

Example

After assembling the `AddressBook.jar` JAR file contains the following files after assembly. The files added in this task are in bold. These files include the WSDL file, the deployment descriptors, and the JAX-RPC mapping file.

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookSoapBindingImpl.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

What to do next

Configure the `webservices.xml` deployment descriptor . You need to configure the deployment descriptors for the Web service so that WebSphere Application Server can process the incoming Web services requests.

Assembling a WAR file that is enabled for Web services from Java code

You can assemble a Web archive (WAR) file that is enabled for Web services from Java code with an assembly tool.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

For Java API for XML-Based Web Services (JAX-WS) Web service applications, you need the portable artifacts that are generated by the wsgen command-line tool when starting from a service endpoint implementation to complete this task. The wsgen tool processes a compiled service endpoint implementation class as input and generates the following portable artifacts:

- any additional Java Architecture for XML Binding (JAXB) classes that are required to marshal and unmarshal the message contents. The additional classes include classes that are represented by the @RequestWrapper annotation and the @ResponseWrapper annotation for a wrapped method.
- a WSDL file if the optional *-wsdl* argument is specified. The wsgen command does not automatically generate the WSDL file. The WSDL file is automatically generated when you deploy the service endpoint.

For Java API for XML-based RPC (JAX-RPC) Web service applications, you need the following artifacts that are generated by the WSDL2Java command-line tool to complete this task:

- An assembled WAR file that contains the `web.xml` file, but is not enabled for Web services.
- The Java class for the service endpoint interface
- A Web Services Description Language (WSDL) file
- The complete `webservices.xml`,`ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file classes that are generated by the WSDL2Java command.

About this task

Assemble a Web services-enabled WAR file from Java code by following the actions in the steps for this task section.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Import the JavaBeans implementation and the artifacts generated by the command-line tooling into the assembly tool.
4. Migrate WAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.

Results

The artifacts required to enable the Web module for Web services are added to the WAR file.

What to do next

Now you can assemble the WAR file that is enabled for Web services into an EAR file.

Assembling a Web services-enabled WAR file from a WSDL file

You can assemble a Web archive (WAR) file from a Web Services Description Language (WSDL) file that is enabled for Web services.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

For Java API for XML-Based Web Services (JAX-WS) Web service applications, you need the portable artifacts that are generated by the wsimport command-line tool when starting from a WSDL file to complete this task. The wsimport tool processes a WSDL file as input and generates the following portable artifacts:

- Service Endpoint Interface (SEI)
- Service class
- Exception classes that are mapped from the `wsdl:fault` class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

You can package the generated artifacts in a Web archive (WAR) file with the WSDL file and schema documents along with the endpoint implementation that you plan to deploy.

For Java API for XML-based RPC (JAX-RPC) Web service applications, you need the following artifacts that are generated by the WSDL2Java command-line tool to complete this task:

- An assembled WAR file that contains the Enterprise JavaBeans (EJB) implementation, all the classes that generate from the WSDL2Java command-line tool and the `web.xml` deployment descriptor file.
- A WSDL file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file.

About this task

Assemble a Web services-enabled WAR file from a WSDL file by following the actions in the steps for this task section.

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Import the JavaBeans implementation and the artifacts generated by the command-line tooling into the assembly tool.
4. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.

Results

The artifacts required to enable the Web module for Web services is added to the WAR file.

What to do next

Now you can assemble the WAR file that is enabled for Web services into an EAR file.

Assembling an enterprise bean JAR file into an EAR file

You can assemble an enterprise bean Java archive (JAR) file into an enterprise archive (EAR) file with an assembly tool. Assembling the JAR file, and now the EAR file, are required tasks to enable Java code for Web services.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

Before assembling a Web services-enabled EAR file you must assemble an enterprise bean JAR file that you want to enable for Web services. To learn more about the artifacts that are needed for the assembly of the enterprise bean JAR file see Assemble an enterprise bean JAR file from Java code that is enabled for Web services.

About this task

To assemble a Web services-enabled EAR file:

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Assemble the Web services-enabled JAR file into an EAR file. The EAR file can contain an enterprise bean or application client JAR files, WAR files, Web applications, and metadata describing the applications or `application.xml` files.

Results

A Web services-enabled EAR file.

Example

In the following example, there is an `application.xml` deployment descriptor packaged with a Web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.jar
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
 <application id="Application_ID">
  <display-name>AddressBookJ2WEE</display-name>
  <description>AddressBook EJB Example from Java</description>
  <module id="EjbModule_1">
   <ejb>AddressBook.jar</ejb>
  </module>
 </application>
```

What to do next

Enable the EAR file. Then, deploy the EAR file into WebSphere Application Server.

Assembling a Web services-enabled WAR into an EAR file

You can assemble a Web services-enabled Web archive (WAR) file into an enterprise archive (EAR) file with an assembly tool.

Before you begin

You can assemble Java-based Web services modules with assembly tools provided with WebSphere Application Server.

About this task

Assemble a Web services-enabled WAR file into an EAR file using the steps provided in this task section.

1. Start an assembly tool. Read about starting the assembly tool in theRational Application Developer documentation.
2. Assemble the Web services-enabled WAR file into an EAR file. Now assemble the EAR file that contains the JAR or WAR files. The EAR file can contain an enterprise bean or application client JAR files; Web applications or WAR files; and metadata describing the applications or `application.xml` files.

Results

A Web services-enabled EAR file.

Example

In the following example, there is an `application.xml` deployment descriptor packaged with a Web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.war
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
 <application id="Application_ID">
  <display-name>AddressBook</display-name>
  <description>AddressBook Example from Java bean</description>
  <module id="WebModule_1">
   <web>
    <web-uri>AddressBook.war</web-uri>
    <context-root>/AddressBook</context-root>
   </web>
  </module>
 </application>
```

What to do next

Deploy Web services.

Enabling an EAR file for EJB modules that contain Web services

When your enterprise archive (EAR) file contains enterprise JavaBeans (EJB) modules that contain Web services, you must run the endptEnabler command-line tool or an assembly tool before deployment to produce a Web services endpoint Web archive (WAR) fiile.

Before you begin

Assemble an enterprise Java archive (JAR) file that is enabled for Web services from an enterprise bean. The enterprise JAR file is an artifact that is required to build the EAR file.

About this task

You can add router modules to your application that is enabled for Web services with either the **endptEnabler** command-line tool or with assembly tools provided with WebSphere Application Server. The

tool that you choose to use for this task depends on your preference to work with a command-line tool or a graphical user interface. See the assembly tools documentation to learn how to use assembly tool to accomplish this task.

These tools add one or more router modules to the EAR file for each Web service-enabled enterprise JavaBeans (EJB) module contained in the EAR file. A router module provides an endpoint for the Web service in a particular EJB module.

You should not modify the contents of the EJB module or the Web module that was generated using the **endptEnabler**command-line tool. If you do, an error occurs during run time. The following is an example of the error that displays:

```
"Error]- WSWS3142E: Error: Could not find Web services engine.]: javax.servlet.ServletException: WSWS3142E:
Error: Could not find Web services engine."
```

Each router module supports a specific transport such as HTTP or Java Message Service (JMS). If no enterprise bean JAR modules are located in the EAR file, it is not necessary to use these tools.

Enable an EAR file with the **endptEnabler** command-line tool. In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application.

What to do next

Deploy the EAR file into WebSphere Application Server. An assembled EAR file that is enabled for Web services is required for deployment.

Enabling an EAR file for Web services with the endptEnabler command

Use the endptEnabler command-line tool to enable an enterprise archive (EAR) file for Enterprise JavaBeans (EJB) modules that contain Web services and to specify whether the Web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.

Before you begin

Before doing this task, you need to assemble a Web services-enabled enterprise Java archive (JAR) into an EAR file.

About this task

The endptEnabler command-line tool adds one or more router modules to the EAR file for each Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) based Web service-enabled enterprise bean Java archive (JAR) module within the EAR file. A router module provides an endpoint for the Web services in a particular enterprise bean JAR module.

Each router module supports a specific transport such as HTTP or JMS. An HTTP router module is a Web Archive (WAR) module that provides an HTTP endpoint for each of the Web services contained within a particular enterprise bean JAR module. Likewise, a JMS router module is an enterprise bean JAR module that contains a Message Driven Bean (MDB) that serves as the message listener for requests intended for the Web service endpoints.

If no enterprise bean JAR modules exist in the EAR file, it is not necessary to use this tool.

1. Invoke the endptEnabler command from the *install_root*\bin directory. For operating systems such as Linux, HP-UX, Solaris, AIX or z/OS, invoke the command from the *install_root*/bin directory.
2. Enter the name of the EAR file, when prompted.

3. Enter various input values as requested by the endptEnabler command. You are prompted for various input values for each enterprise bean JAR module that is enabled for Web services in the EAR file. Typically, you accept the defaults for each prompt. To learn about the properties of this command, see the endptEnabler command documentation.
   a. Specify an HTTP router module to transport your EJB-based Web service. Use the `-transport http` option to indicate the Web service is available using HTTP. One router module is created for each enterprise bean JAR file that contains either JAX-WS or JAX-RPC Web services.
   b. Specify an JMS router module to transport your EJB-based Web service. Use the `-transport jms` option to indicate the Web service is available using JMS. One router module is created for each enterprise bean JAR file that contains either JAX-WS or JAX-RPC Web services.

Results

An HTTP or JMS router module is added to the EAR file for each enterprise bean JAR module within the EAR file that contains Web services endpoints. For HTTP, a context-root is configured for the application so that the Web service can be invoked through a Web address. The Web address used to invoke the Web service is:

`http://`*`host[:port]`*`/`*`<context-root>`*`/services/`*`<port-component-name>`*

Ensure that you install the HTTP or Java Message Service (JMS) router module that you generated with the endptEnabler command onto the same target as your Web services enterprise bean JAR files. These HTTP or JMS router modules are included in your Web services application and they need to use the runtime libraries of the application server.

What to do next

Deploy the EAR file onto your application server. An assembled EAR file that is enabled for Web services is required for deployment.

If you are using JMS as a transport for your Web service requests, define the various JMS objects such as queues, topics, or connection factories, that will be used by your application prior to installing the application.

endptEnabler command:

The **endptEnabler** command is used to enable a set of Web services within an enterprise archive (EAR) file. The **endptEnabler** command must run on EAR files containing Enterprise JavaBeans (EJB) modules that are enabled for Web services.

Each router module provides a Web service endpoint for a particular transport. For example, you can add a HTTP router module so that the Web service can receive requests over the HTTP transport. Or, you can add a Java Message Service (JMS) router module so that the Web service can receive requests from a JMS queue or topic.

In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application. The **endptEnabler** command makes a backup copy of your original EAR file in the event that you need to remove or add services at a later time. If your EAR file contains an enterprise bean Java archive (JAR) file that is enabled for Web services, you must run the **endptEnabler** command before the EAR file is deployed. Otherwise, you do not need to run the command.

endptEnabler usage syntax

Invoke the **endptEnabler** command from the WebSphere Application Server `bin` directory. The command syntax is presented in the following example:

```
endptEnabler
  [-verbose|-v]
  [-quiet|-q]
  [-help|-h|-?]
  [-properties|-p properties-filename]
  [-transport|-t default-transports]
  [-enableHttpRouterSecurity]
  [ear-filename]
```

All parameters are optional and described in the following list:

- **-verbose, -v**

  This parameter details and displays progress messages as the endptEnabler tool processes the EAR file. This command-line option is mapped to the verbose global property.
- **-quiet, -q**

  This parameter makes sure that there are no displays of per-module progress messages as the endptEnabler tool processes the EAR file. This command-line option is mapped to the quiet global property.
- **-help, -h, -?**

  This parameter displays a brief help message that explains the various options.
- **-properties, -p <properties-filename>**

  This parameter reads properties from the *properties-filename* properties and controls the behavior of the endptEnabler tool.
- **-transport, -t <default-transports>**

  This parameter specifies the default list of transports for which router modules are created for each enterprise bean JAR file contained in the EAR file. This command-line option is mapped to the defaultTransports global property. The following are examples of this parameter:

  ```
  -transport http (the default)
  -transport jms
  -t http,jms
  ```
- **-enableHttpRouterSecurity**

  This parameter enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB modules are secured in the enterprise bean JAR file. This command-line option is mapped to the http.enableRouterSecurity global property.
- **<ear-filename>**

  This parameter specifies the name of the EAR file to be processed.

  If the *ear-filename* parameter is not entered on the command line, the interactive mode is used. In the interactive mode, you are prompted for the EAR file name, the router module names and other important values as the processing occurs. The following dialog is an example of the endptEnabler interactive mode.

  In this dialog, the user input is in fixed width font and the endptEnabler output is in bold.

  ```
  endptEnabler<enter>
  WSWS2004I: IBM WebSphere Application Server Release 5
  WSWS2005I: Web Services Enterprise Archive Endpoint Enabler Tool.
  WSWS2007I: (C) COPYRIGHT International Business Machines Corp. 1997, 2003
  WSWS2006I: Please enter the name of your EAR file: AddressBook.ear<enter>

  WSWS2003I: Backing up EAR file to: AddressBook.ear~

  WSWS2016I: Loading EAR file: AddressBook.ear
  WSWS2017I: Found EJB Module: AddressBookEJB.jar

  WSWS2029I: Enter http router name for EJB Module AddressBookEJB
  [AddressBookEJB_HTTPRouter.war]:<enter>
  WSWS2030I: Enter http context root for EJB Module AddressBookEJB
  [/AddressBookEJB]:<enter>
  WSWS2024I: Adding http router for EJB Module AddressBookEJB.jar.
  ```

```
WSWS2036I: Saving EAR file AddressBook.ear...
WSWS2037I: Finished saving the EAR file.
WSWS2018I: Finished processing EAR file AddressBook.ear.
```

If the *ear-filename* parameter is entered on the command-line, the non-interactive mode is used. In the non-interactive mode, the router module names and other important values are determined from the user-specified properties or default values.

endptEnabler properties

With the **endptEnabler** command you can control its run-time behavior by specifying a set of properties with the -properties command-line option. These properties are organized in one of two ways: global and per-module. Global properties affect the overall behavior of the tool as it processes multiple enterprise bean JAR modules within the EAR file. Per-module properties affect the processing of a particular enterprise bean JAR module.

Global properties

The following table describes the global properties supported by the **endptEnabler** command:

Property name	Description	Default value
verbose	Displays detailed progress messages.	False
quiet	Displays only brief progress messages.	False
http.enableRouter Security	Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB modules are secured in the enterprise bean JAR file.	False
http.router ModuleNameSuffix	Specifies the suffix used to construct default HTTP router module names. The `.war` extension is added by the **endptEnabler** command.	_HTTPRouter
jms.routerModule NameSuffix	Specifies the suffix used to construct default JMS router module names. The `.jar` extension is added by the **endptEnabler** command.	_JMSRouter
jms.default DestinationType	Specifies the default destination type to use for all JMS router modules that are added to the EAR file. This type is either queue or topic.	queue
defaultTransports	Specifies the default list of transports for which router modules are created. The list can contain the values `http` and `jms`. Multiple values are separated by a comma. Examples are: `http, jms` and `http,jms`.	http

Per-module properties

The following table describes the per-module properties supported by the **endptEnabler** command. The *ejbJarName* variable refers to the name of an enterprise bean JAR module within the EAR file, without the `.jar` extension.

Property name	Description	Default value

<ejbJarName> .transports	Lists the transports for which router modules are created for a particular enterprise bean JAR file. The list can contain the values `http` and`jms`. Multiple values are separated by a comma. Examples are: `http, jms` and `http,jms`.	http
<ejbJarName>.http.skip	Specifies the flag which bypasses the addition of an HTTP router module, even if it otherwise is added based on other properties. Valid values are `true` and `false`.	false
<ejbJarName> .http.routerModuleName	Specifies the name of the HTTP router module for a particular enterprise bean JAR file.	*ejbJarName*_HTTPRouter
<ejbJarName> .http.contextRoot	Specifies the context root associated with the HTTP router module for a particular enterprise bean JAR file.	/*ejbJarName*
<ejbJarName>.jms.skip	Specifies the flag that bypasses the addition of an JMS router module even if it otherwise is added based on other properties. Valid values are `true` and `false`.	false
<ejbJarName>.jms. routerModuleName	Specifies the name of the JMS router module for a particular enterprise bean JAR file.	*ejbJarName*_JMSRouter
<ejbJarName>.jms.activationSpecJndiName	Specifies the Java Naming and Directory Interface (JNDI) name of the activation specification that is configured for the Message Driven Bean (MDB) within the JMS router module.	null
<ejbJarName>.jms.listenerInputPortName	Specifies the name of the listener port to configure for the MDB within the JMS router module. The listener port is configured only if an activationSpecJndiName property is not specified.	null
<ejbJarName>.jms. destinationType	Specifies the JMS destination type associated with the MDB within the JMS router. Valid values are `queue` and `topic`.	queue
<ejbJarName>.<port_local_name>.http.urlPattern	Specifies the URL pattern for ports. If you have EJB module with the indicated name that has a port with the indicated local name, you can specify the HTTP URL pattern with this property. This property only applies to HTTP router modules. It has no affect on JMS router modules.	null

Properties example

Suppose an EAR file contains an enterprise bean JAR file named, `StockQuoteEJB.jar` that contains Web services. The following set of properties can be used to control the **endptEnabler** command run-time behavior as it processes the EAR file:

```
StockQuoteEJB.transports=http,jms

StockQuoteEJB.http.routerModuleName=StockQuoteEJB_HTTP

StockQuoteEJB.http.contextRoot=/StockQuote

StockQuoteEJB.jms.routerModuleName=StockQuoteEJB_JMS

StockQuoteEJB.jms.destinationType=queue
```

endptEnabler examples

The following commands are examples of how the **endptEnabler** command can be used:

```
endptEnabler MyApp.ear

endptEnabler -t jms,http MyApp.ear

endptEnabler -v -properties MyApp.props MyApp.ear

endptEnabler -q -t jms MyApp.ear

endptEnabler -v -t http,jms
```

Deploying Web services applications onto application servers

After assembling the artifacts required to enable the Web module for Web services into an enterprise archive (EAR) file, you can deploy the EAR file into the application server.

Before you begin

To deploy Java-based Web services, you need an enterprise application, also known as an EAR file that is configured and enabled for Web services.

A Java API for XML-Based Web Services (JAX-WS) application does not require additional bindings and deployment descriptors for deployment whereas a Java API for XML-based RPC (JAX-RPC) Web services application requires you to add additional bindings and deployment descriptors for application deployment. JAX-WS is much more dynamic, and does not require any of the static data generated by the deployment step required for deploying JAX-RPC applications.

For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

Note: In a mixed node cell, you can only target a JAX-WS enabled enterprise beans module to a server using WebSphere Application Server Version 7.0. However, you can target a JAX-WS enabled Web application archives (WAR) module to a server using either WebSphere Application Server Version 7.0 or WebSphere Application Server Version 6.1 Feature Pack for Web Services

You can use the wsdeploy command with JAX-RPC applications to add WebSphere product-specific deployment classes to a Web services-compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file.

To install or deploy a JAX-WS application, you only need to install the JAX-WS enabled EAR file. If your Web services application contains only JAX-WS endpoints, you do not need to run the wsdeploy command, as this command is used to process only JAX-RPC endpoints.

Ensure that you have installed the HTTP or Java Message Service (JMS) router module that was generated with the endptEnabler command onto the same target as your Web services enterprise bean JAR files. These HTTP or JMS router modules are included in your Web services application and they need to use the runtime libraries of the application server.

About this task

This task is one of the steps in developing and implementing Web services.

You can use either the administrative console or the **wsadmin** scripting tool to deploy an EAR file. If you are installing an containing Web services by using the **wsadmin** command, specify the **-deployws** option for JAX-RPC applications. If you are installing an application containing Web services by using the administrative console, select **Deploy WebServices** in the Install New Application wizard. For more information about installing applications using the administrative console see Installing a new application.

If your JAX-RPC Web services application was previously deployed with the **wsdeploy** command, it is not necessary to specify Web services deployment during installation.

The following actions deploy the EAR file with the **wsadmin** command:

1. Start *install_root*/bin/wsadmin from a command prompt.
2. Deploy the EAR file.
   - For JAX-WS Web service applications, enter the **$AdminApp install *EARfile* "-usedefaultbindings"** command at the **wsadmin** prompt.
   - For JAX-RPC Web service applications, enter the **$AdminApp install *EARfile* "-usedefaultbindings -deployws"** command at the **wsadmin** prompt.

Results

You have a Web service installed onto your application server.

Note: While installing Web services applications that contain a large number of enterprise beans onto the application server, you might receive out of memory errors. If you receive out of memory errors, increase the heap size of your Java Virtual Machine (JVM). If you are installing the application server in a network deployment environment, you might need to increase the heap size of the JVM in the application servers in which you are installing the application, and in the deployment manager profile, dmgr. Read about tuning the IBM virtual machine for Java documentation to learn more about tuning the application server environment.

What to do next

You can confirm that the Web services application was deployed by entering the Web service endpoint URL in a browser, then viewing an informative page. The information page contains the following information:

```
{http://webservice.pli.tc.wssvt.ibm.com}RetireWebServices
Hello! This is an Axis2 Web service!
```

The first line of this information is variable, depending on your Web service. The URI in the brackets is the namespace and the string that follows, in this example RetireWebServices, is the name of the port used to access the Web service.

The next step you might want to consider is to apply security to your Web service.

Provide options to perform the Web services deployment settings

Use this panel to specify options for Web services deployment.

This administrative console panel is a step in the application installation and update wizards.

To view this panel, you must select **Deploy Web services** on the **Select installation options** panel.

To view this administrative console page, complete the following steps:

1. Click **Applications > New application >** ***application_path*** .
2. Select the option to **Show all installation options and parameters** .
3. Click **Next** to get to the **Step: Select installation options** panel.
4. Select **Deploy Web service**.
5. Click **Next** to get to the **Step: Provide options to perform the Web services deployment** panel.

You can specify the Web services deployment options on this panel only when installing or updating an application that uses Web services.

The wsdeploy command is supported by Java API for XML-based RPC (JAX-RPC) applications. The Java API for XML-Based Web Services (JAX-WS) programming model that is implemented by the application server does not support the wsdeploy command. If your Web services application contains only JAX-WS endpoints, you do not need to run the wsdeploy command, as this command is used to process only JAX-RPC endpoints.

The options that you specify set parameter values for the **wsdeploy** command. The **wsdeploy** command adds product-specific deployment classes to a Web services-compatible enterprise archive (EAR) file or an application client Java archive (JAR) file. These classes include:

- Stubs
- Serializers and deserializers
- Implementations of service interfaces

The **wsdeploy** command is run during installation after you click **Finish** on the **Summary** panel of the wizard.

Related tasks

Installing enterprise application files with the console
Installing Java Platform, Enterprise Edition (Java EE) application files consists of placing assembled enterprise application, Web, enterprise bean (EJB), or other installable modules on a server or cluster configured to hold the files. Installed files that start and run properly are considered *deployed*.

Related reference

“wsdeploy command” on page 743
This topic explains how to use the **wsdeploy** command-line tool with Web services. The **wsdeploy** command adds WebSphere product-specific deployment classes to a Web services-compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file.

Enterprise application settings
Use this page to configure an enterprise application.

Deploy Web services option - Classpath

Specifies entries to add to the CLASSPATH when the generated classes are compiled.

To specify the class paths of multiple entries, you need to separate the entries with a semicolon on Windows platforms and on Linux, Unix, and z/OS platforms, you need to use a colon to separate the entries. This is the same separator that is used with the CLASSPATH environment variable.

This option is the same as the **wsdeploy** command parameter `-cp class_path`.

Data type	String
Default	null

Deploy Web services option - Extension Directories

Specifies a directory that contains zipped or Java archive (JAR) files. All zipped and JAR files in this directory are added to the CLASSPATH used to compile the generated files.

This option is the same as the **wsdeploy** command parameter `-jardir directory`.

Data type	String
Default	null

wsdeploy command

This topic explains how to use the **wsdeploy** command-line tool with Web services. The **wsdeploy** command adds WebSphere product-specific deployment classes to a Web services-compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file.

The wsdeploy command is supported by Java API for XML-based RPC (JAX-RPC) applications. The Java API for XML-Based Web Services (JAX-WS) programming model that is implemented by the application server does not support the wsdeploy command. If your Web services application contains only JAX-WS endpoints, you do not need to run the wsdeploy command, as this command is used to process only JAX-RPC endpoints.

The deployment classes that are added by the wsdeploy tool to a Web services-compatible EAR file or a JAR file include:
- Stubs
- Serializers and deserializers
- Implementations of service interfaces

This deployment step must be performed at least once, and can be performed more often. Deployment can be performed separately using the **wsdeploy** command, assembly tools, or when the application is installed. When using the **wsadmin** command for installation, specify the **-deployws** option.

The **wsdeploy** command operates as noted in the following list:
- Each module in the enterprise application or JAR file is examined.
- If the module contains Web services implementations, indicated by the presence of the `webservices.xml` deployment descriptor, the associated Web Services Description Language (WSDL) files are located and the **WSDL2Java** command is run with the role deploy-server option.
- If the module contains Web services clients, indicated by the presence of the client deployment descriptor, the associated WSDL files are located and the **WSDL2Java** command is run with the role deploy-client option.
- The files generated by the **WSDL2Java** command are compiled and repackaged.

See **WSDL2Java** command for more information about the files that are generated for deployment.

When the generated files are compiled, they can reference application-specific classes outside the EAR or JAR file, if the EAR or JAR file is not self-contained. In this case, use either the -jardir or -cp option to specify additional JAR or zip files to be added to CLASSPATH variable when the generated files are compiled.

wsdeploy command syntax

The command syntax is noted in the following example:

```
wsdeploy Input_filename Output_filename [options]
```

Required options:

- ***Input_filename***

  Specifies the path to the EAR or JAR file to deploy.
- ***Output_filename***

  Specifies the path of the deployed EAR or JAR file. If *output_filename* already exists, it is silently overwritten. The *output_filename* can be the same as the *input_filename*.

Other options:

- **-jardir** *directory*

  Specifies a directory that contains JAR or zip files. All JAR and zip files in this directory are added to the CLASSPATH used to compile the generated files. This option can be specified zero or more times.
- -**cp** *entries*

  Specifies entries to add to the CLASSPATH when the generated classes are compiled. Multiple entries are separated the same as they are in the CLASSPATH environment variable.
- **-codegen**

  Specifies to generate but not compile deployment code. This option implicitly specifies the -keep option.
- **-debug**

  Includes debugging information when compiling, that is, use javac -g to compile.
- **-help**

  Displays a help message and exit.
- **-ignoreerrors**

  Do not stop deployment if validation or compilation errors are encountered.
- **-keep**

  Do not delete working directories containing generated classes. A message is displayed indicating the name of the working directory that is retained.
- **-novalidate**

  Do not validate the Web services deployment descriptors in the input file.
- **-trace**

  Displays processing information, including the names of the generated files.

Example The following example illustrates how the options are used with the **wsdeploy** command:

```
wsdeploy x.ear x_deployed.ear -trace -keep
Processing web service module x_client.jar.
Keeping directory: f:\temp\Base53383.tmp for module: x_client.jar.
Parsing XML file:f:\temp\Base53383.tmp\WarDeploy.wsdl
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java.
Done processing module x_client.jar.
```

Messages

- Flag *-f* is not valid.

  Option *f* was not recognized as a valid option.
- Flag *-c* is ambiguous.

  Options can be abbreviated, but the abbreviation must be unique. In this case, the **wsdeploy** command cannot determine which option was intended.
- Flag *-c* is missing parameter *-p*.

  A required parameter for an option is omitted.
- Missing *p* parameter.

A required option is omitted.

JAX-WS application deployment model

The administration function of the product is enhanced to support installing and deploying Java Application Programming Interface (API) for XML Web Services (JAX-WS) applications like any other WebSphere Application Server applications.

A JAX-WS application is packaged as a Web archive (WAR) file or a WAR module within an Enterprise Archive (EAR) file. The JAX-WS application deployment model is similar to the Java API for XML Remote Protocol Call (JAX-RPC) Web services application model. The main difference between them is that JAX-RPC Web services application requires you to add additional bindings and deployment descriptors for application deployment. A JAX-WS application does not require additional bindings and deployment descriptors for deployment. You can deploy your JAX-WS applications as you would deploy any other WebSphere Application Server application.

JAX-WS Web services is a rewrite of JAX-RPC Web services. The table compares the Web services stack for both JAX-WS and JAX-RPC Web services.

JAX-RPC Web services	**JAX-WS Web services**
Bindings are proprietary	Bindings are based on the open source Java API for XML Bindings (JAXB)
Parsing is proprietary	Parsing is based on the open source Java Specification Request (JSR) 173
No Java annotations support	Support for Java annotations such as `@WebService`, `@WebMethod`, `@WebParam`, `@WebResult`, and `@SOAPBinding`
During deployment, some deployment descriptor files are created in a JAX-RPC based service and client. The following files are created on the services side, when it is an EJB based Web service and EJB based module: • `webservices.xml` • <name_of_service>`_mapping.xml` • `ibm-webservices-bnd.xmi` • `ibm-webservices-ext.xmi` When the service is a JavaBeans-based or Web module-based service, the following files and deployment descriptors are required: • webservices.xml • <name_of_service>_mapping.xml • In the web.xml, there is no additional content • ibm-webservices-bnd.xmi • ibm-webservices-ext.xmi The web.xml exists in both EJB and JavaBeans based services. However, there is no additional content added to the file during deployment of a Web service application or module.	For JAX-WS Web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

Note: In WebSphere Application Server Version 7.0, the default annotation support behavior has changed. In the Version 6.1 Feature Pack for Web services, the default behavior is to scan pre-Java EE 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default behavior is to not scan pre-Java EE 5 modules for annotations during application installation

or server startup. You can preserve compatibility with feature packs from previous releases by either setting the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a Web archive (WAR) file or EJB module or by defining the Java virtual machine custom property, com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations documentation.

Testing Web services-enabled clients

Once you have developed, assembled, deployed and configured your Web service, you can test to confirm your Web service runs in the application server environment.

Before you begin

Before testing your Web services Java client to confirm your Web service runs in the WebSphere Application Server environment, verify that the server endpoint specified in the client Web Services Description Language (WSDL) file is running and available.

About this task

Tests are run differently depending on whether the client module is in a Java EE container or if the client is running in the Thin Client for Java API for XML-based RPC (JAX-RPC) with WebSphere Application Server application environment or the Thin Client for Java API for XML-Based Web Services (JAX-WS) with WebSphere Application Server application environment.

1. Setup the application server environment.
   - If you are running the application server, run the **setupCmdLine** script to set the application server environment values globally.

     Run the **setupCmdLine** script from the *profile_root*/<*application_server*>/bin directory.
   - If you are running the application client, run the **setupClient** script to set the application client environment values globally.

     app_client_root/AppClient/bin/setupClient.sh
2. Test an unmanaged client JAR file.
   a. Run your application with the **java** command.

      For JAX-WS applications:

      ```
      "$JAVA_HOME/bin/java"
      -classpath
      "$WAS_HOME/runtimes/com.ibm.jaxws.thinclient_7.0.0.jar:
      <list_of_your_application_jars_and_classes>"
      <fully_qualified_class_name_to_run> <your_application_parameters>
      ```

      For JAX-RPC applications:

      ```
      "$JAVA_HOME/bin/java"
      -classpath
      "$WAS_HOME/runtimes/com.ibm.ws.webservices.thinclient_7.0.0.jar:
      <list_of_your_application_jars_and_classes>"
      <fully_qualified_class_name_to_run> <your_application_parameters>
      ```

   The unmanaged client application runs.
3. Test a managed JAX-RPC client EAR file or an unmanaged JAX-WS client EAR file.
   a. Run your client application with the **launchClient** command. The following example illustrates the use of this command:

      ```
      launchClient clientEar
      ```

Results

You have a Web services-enabled client that is tested. Now you can add security measures to the Web service. Security measures are optional.

Using the UDDI registry

The Universal Description, Discovery, and Integration (UDDI) registry is a directory for Web services that is implemented using the UDDI specification. It is a component of WebSphere Application Server.

About this task

Throughout this information, the term *UDDI registry* refers to the UDDI registry component that is supplied as part of WebSphere Application Server.

This section describes the UDDI registry and how to use it. An administrator can install, configure, and manage the UDDI registry. A user can use the UDDI registry user interface and the UDDI Application Programming Interfaces (APIs).

1. To learn about the UDDI registry, see the following topics:
   - "Overview of the Version 3 UDDI registry"
   - "UDDI registry terminology" on page 750
2. To use the UDDI registry, see the following topics:
   - Getting started with the UDDI registry
   - "Using the UDDI registry user interface" on page 833
   - "UDDI registry client programming" on page 819
   - "Java API for XML Registries (JAXR) provider for UDDI" on page 786
3. To install, configure, or manage a UDDI registry, see the following topics:
   - Getting started with the UDDI registry
   - "Setting up and deploying a new UDDI registry" on page 792
   - Configuring UDDI registry security
   - Managing the UDDI registry
   - "UDDI registry management interfaces" on page 753
   - Configuring SOAP API and GUI services for the UDDI registry
   - Applying an upgrade to the UDDI registry
   - Removing and reinstalling the UDDI registry
   - Migrating the UDDI registry
4. To resolve any problems with the UDDI registry, see the following topic:
   - UDDI registry troubleshooting

Overview of the Version 3 UDDI registry

The Universal Description, Discovery, and Integration (UDDI) specification defines a way to publish and discover information about Web services. The term *Web service* describes specific business functionality that is exposed by a company, usually through an Internet connection, to allow another company, its subsidiaries, or software program, to use the service.

You can find the UDDI specification on the OASIS UDDI Web page.

The UDDI specification defines a standard for the visibility, reusability, and manageability that are essential for a service-oriented architecture (SOA) registry service.

The UDDI registry is a directory for Web services that is implemented using the UDDI specification. It is a component of WebSphere Application Server.

A critical component of IBM's on-demand service-oriented architecture, the UDDI registry solves the problem of discovery of technical components for an enterprise and its partners by:

- Providing control, flexibility, and confidentiality so that an enterprise can protect its e-business investments
- Increasing efficiency by making it easier to identify technical assets
- Leveraging existing infrastructures

The following example shows how the UDDI registry can be used in a larger enterprise.

A company has a legacy application that provides telephone numbers and human resources (HR) information about employees. This application is turned into a Web service and published to the registry. A developer in the same company wants to write an application for a procurement function that also needs to provide HR information to the supplier. The application needs to give the supplier access to the employee account codes after the employee provides a name or serial number. Before Web services, the developer might be in one of the following situations:

- The developer does not know about the similar application
- The developer knows about the application, but cannot reuse it because of technical barriers
- The developer knows about the application and reuses it, but only after significant time and negotiation

With UDDI, the developer can search for the Web service and reuse the existing technical component in their new application for the supplier in minutes. The developer saves time and gets the application running sooner, thereby increasing efficiency and saving the company time and money. The UDDI registry was the first version 2 standard-compliant UDDI registry for private enterprise work. The UDDI registry in this version has the following characteristics:

- It supports the UDDI Version 3.0 specification, in addition to the Version 1.0 and Version 2.0 standard APIs.
- It leverages the proven, reliable WebSphere Application Server technology.
- It uses a relational database, such as DB2, for its persistent store.

What is new in UDDI Version 3

The main aspects of the UDDI Version 3 specification that are provided with this version of WebSphere Application Server are as follows:

Improved recognition of the importance of private UDDI registries
Private UDDI registries are registries that are installed, owned, managed, and controlled by a separate body such as a department within a company, a company, an industry consortium, or an e-marketplace.

Publisher-assigned keys
The publisher of a UDDI entity can specify its key, rather than the registry automatically assigning a unique key. This means that more human-friendly, URI-based keys can be used, and it makes it easier to manage multiple registries.

UDDI information model improvements
The UDDI data structures are extended, which improves the ability of UDDI to represent businesses and services through metadata.

Security enhancements
Digital signatures provide additional security. Each of the main UDDI entities can be digitally signed, which improves the integrity and trustworthiness of UDDI data.

Ownership transfer APIs
These APIs support the transfer of the ownership of a UDDI entity from one publisher to another.

UDDI policy
You can set policy to define the behavior of a UDDI registry and therefore recognize the different environments in which a UDDI registry is used.

HTTP GET support for UDDI entities
You can use HTTP GET to access XML representations of each of the UDDI data structures. This extends the HTTP GET service beyond the scope for discovery URLs in the UDDI Version 2 specification.

Additional UDDI registry capabilities

The Version 3 UDDI registry in this version of WebSphere Application Server provides the following capabilities that are additional to support for the UDDI Version 3 specification:

Version 2 UDDI inquiry and publish SOAP API compatibility
There is backward compatibility for the Version 1 and Version 2 SOAP inquiry and publish APIs.

UDDI administrative console extension
The WebSphere Application Server administrative console includes a section that administrators can use to manage UDDI-specific aspects of their WebSphere environment. This management includes the ability to set defaults for initialization of the UDDI node, such as its node ID, and to set the UDDI Version 3 policy values.

UDDI registry administrative interface
The Java Management Extensions (JMX) administrative interface enables administrators to manage UDDI-specific aspects of the WebSphere environment programmatically.

Multidatabase support
The UDDI data is persisted to a registry database. The following database products that are supported by WebSphere Application Server are also supported for use as the persistence store for the UDDI registry. For specific details on supported levels, see Detailed system requirements page.

- Apache Derby Version 10.2
- DB2 for z/OS Versions 7 and 8

User-defined value set support
You can create your own categorization schemes or value sets. These are in addition to the standard schemes, such as North American Industry Classification System (NAICS), that are provided with the UDDI registry.

UDDI utility tools
You can use UDDI utility tools to import or export entities that use the UDDI Version 2 API.

UDDI user interface
The UDDI user console supports the UDDI Version 3 inquiry and publish APIs.

UDDI Version 3 client
The Java client for UDDI Version 3 handles the construction of raw SOAP requests for the client application. It is a JAX-RPC client and uses Version 3 data types, which are generated from the UDDI Version 3 Web Services Description Language (WSDL) and schema. These data types are serialized or deserialized to the XML, which constitutes the raw UDDI requests.

UDDI Version 2 clients
The following clients for UDDI Version 2 requests are provided:

- UDDI4J. A Java class library for issuing UDDI requests. This client is provided in WebSphere Application Server Version 5 for both UDDI Version 1 requests (uddi4j.jar) and Version 2 requests (uddi4jv2.jar). These class libraries are still supported, as part of the com.ibm.uddi.jar file, but are now both deprecated.

- JAXR. The Java API for XML Registries (JAXR) is a Java client API for accessing UDDI and ebXML registries. WebSphere Application Server provides a JAXR provider for accessing the UDDI registry that conforms to the JAXR 1.0 specification.
- EJB. An Enterprise JavaBeans (EJB) interface for issuing UDDI Version 2 requests. This client is still supported, but is now deprecated.

UDDI registry terminology

Some terms specific to the UDDI registry are explained. Also, the relationship between the versions of the UDDI registry, the Organization for the Advancement of Structured Information (OASIS) specification, and the WebSphere Application Server level are shown.

Throughout the UDDI information in this information center, the directory location of WebSphere Application Server is referred to as *app_server_root*.

UDDI Definitions

bindingTemplate
AbindingTemplate is technical information about a service entry point and construction specifications.

businessEntity
A businessEntity is information about the party who publishes information about a family of services.

businessService
A businessService is descriptive information about a particular service.

customized UDDI node
A customized UDDI node is a UDDI node that is initialized with customized settings for the UDDI properties and UDDI policies. In particular, this type kind of node does not have default values for those properties that are read-only after initialization.

Use a customized UDDI node for anything other than simple testing purposes (for which a default UDDI node is enough). To set up a customized UDDI node, see Setting up a customized UDDI node.

When you first start a customized UDDI node, you must set values for certain properties, and then initialize the node (using the administrative console or UDDI administrative interface), before the node is ready to accept UDDI requests. The properties that you must set control characteristics of the UDDI node that cannot be changed after initialization.

An advantage of using a customized UDDI node is that can set these properties to values that are suitable for your environment and usage of UDDI.

After a customized UDDI node is initialized, it is the same as a default UDDI node except that it uses customized UDDI property and policy values.

default UDDI node
A default UDDI node is a UDDI node that is initialized with default settings for the UDDI properties and UDDI policies, including the properties that are read-only after initialization. A default UDDI node is intended for use for testing and to provide a simple way to become familiar with the behavior of the UDDI registry.

You can set up a default UDDI node in two ways. The first is to run the uddiDeploy.jacl script, specifying the 'default' option, in which case the UDDI database will be an Apache Derby database that is created for you automatically.

The second is to make sure the PDS member INSERT is included in the JCL used to create the database, in which case the UDDI database can be Apache Derby or DB2.

After a default UDDI node is initialized, it is the same as a customized UDDI node except that it uses default UDDI property and policy values.

policy profile
A policy profile is set of UDDI policies. The default policy profile is the profile created when the default UDDI node is created. In the default policy profile, the nodeID and root key generator are set to read-only and cannot be changed after installation.

publisherAssertion
A publisherAssertion is information about a relationship between two parties, asserted by one or both.

tModel
A tModel (short for technical model) is a data structure representing a reusable concept, such as a Web service type, a protocol used by Web services, or a category system.

tModel keys in a service description are a technical ″fingerprint″ that you can use to trace the compatibility origins of a given service. They provide a common point of reference so that you can identify compatible services.

tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels that represent value sets such as category, identifier, and relationship systems are used to provide additional data to the UDDI core entities to facilitate discovery along a number of dimensions. This additional data is captured in keyedReferences that reside in categoryBags, identifierBags, or publisherAssertions. The tModelKey attributes in these keyedReferences refer to the value set that relates to the concept or namespace being represented. The keyValues contain the actual values from that value set. In some cases, keyNames are significant, for example, to describe relationships and when using the general keywords value set. In all other cases, keyNames provide a version of the keyValue that people can read.

UDDI application
The UDDI application is the UDDI registry enterprise application.

UDDI entitlement
A UDDI entitlement is an entitlement that a UDDI user or publisher has in a UDDI registry, such as the capability to publish keyGenerators, or the tier to which the publisher is assigned (in other words, the number of entities that the publisher is entitled to publish). Each UDDI publisher has a range of settings for the various UDDI entitlements. A UDDI entitlement is sometimes referred to as a 'user entitlement', or as the UDDI publisher's set of 'user entitlements'.

UDDI node
A UDDI node is a set of Web Services that support at least one of the UDDI API sets, which supports interaction with UDDI data through the UDDI APIs. There is no direct mapping between a UDDI node and a WebSphere Application Server node. A UDDI node consists of an instance of the UDDI application running in an application server (or a cluster of UDDI application instances running in a cluster of application servers), together with an instance of the UDDI database containing UDDI data.

UDDI node initialization
UDDI node initialization is the process that sets up values in the UDDI database, and establishes the ″personality″ of the UDDI node. A UDDI node cannot accept UDDI API requests until it is initialized.

UDDI node state
The UDDI node state describes the current state of the UDDI node, as opposed to the state of the UDDI application (which is either stopped or started). A UDDI node can be in one of the following states:

- not initialized
- initialization pending
- initialization in progress

- migration pending
- migration in progress
- value set creation pending
- value set creation in progress
- activated
- deactivated

UDDI NodeId
The UDDI NodeId is a unique identifier of a UDDI node.

UDDI policy
A UDDI policy is a statement of required and expected behavior of a UDDI registry, specified using policy values for the various policies that are defined in the UDDI Version specification.

UDDI property
A UDDI property is a value for a property that controls the personality or behavior of a UDDI node.

UDDI publisher
A UDDI publisher is a WebSphere user who is entitled to publish UDDI entities to a specified UDDI registry. A UDDI publisher is sometimes referred to as a 'UDDI user', or simply as a 'publisher' when used in a UDDI context.

UDDI registry
A UDDI registry comprises one or more UDDI nodes. The UDDI registry in this version of WebSphere Application Server supports single-node UDDI registries only.

UDDI tier
A UDDI tier determines the number of UDDI entities of each type (business, services per business, bindings per service, tModel, publisher assertion) that a UDDI publisher is entitled to publish. Each UDDI publisher is assigned (either by default or explicitly by a UDDI administrator) to a particular tier, and cannot publish more entities than are allowed for that tier. There are some predefined tiers supplied with the UDDI registry, and a UDDI administrator can create additional tiers. A UDDI tier is often referred to simply as a 'tier' when used in a UDDI context.

Version 2 UDDI registry
A Version 2 UDDI registry is a UDDI registry implementation that supports Version 2 of the UDDI specification and also Version 1. A Version 2 UDDI registry is included in WebSphere Application Server Network Deployment Version 5.x.

Version 3 UDDI registry
A Version 3 UDDI registry is a UDDI registry implementation that supports Version 3 of the UDDI specification, and also Versions 1 and 2. A Version 3 UDDI registry is included in WebSphere Application Server. Note that Version 3 UDDI registry does not indicate a UDDI registry implementation that supports only UDDI Version 3 requests.

The following table shows how the various versions of the UDDI registry relate to the relevant OASIS specification and WebSphere Application Server level:

UDDI registry Version	OASIS UDDI specification levels supported	Supported on WebSphere Application Server version
1.1	• UDDI Version 1 • UDDI Version 2	4.0.2
1.1.1	• UDDI Version 1 • UDDI Version 2	4.0.3 and later
2.0.x	• UDDI Version 1 • UDDI Version 2	5.0.x

2.1.x	• UDDI Version 1 • UDDI Version 2	5.1.x
3.0.2	• UDDI Version 1 • UDDI Version 2.0.4 (APIs), Version 2.0.3 (data structures) • UDDI Version 3.0.2	6.0.1 and later

UDDI registry management interfaces

This topic explains interfaces and tools that you can use to manage UDDI nodes programmatically.

UDDI registry Administrative (JMX) Interface

The UDDI registry Administrative (JMX) Interface provides a Java API that allows you to manage runtime configuration settings to control UDDI registry runtime behavior, such as setting the maximum number of results that UDDI users can receive for inquiry requests, or creating publish limits for UDDI publishers. Sample client code is provided for you to build on.

User Defined Value Set Support in the UDDI registry

User Defined Value Set Support in the UDDI registry explains the tooling provided to manage your own categorization value sets, including loading value set data into a UDDI registry node.

UDDI Utility Tools

UDDI Utility Tools explains the tooling and Java API for promoting version 2 entities from one UDDI registry to another while retaining entity keys. This is particularly useful for publishing canonical tModels with a predefined key.

UDDI registry Administrative (JMX) Interface

You can use the UDDI registry Administrative Interface to inspect and manage the runtime configuration of a UDDI application. You can manage the information and the activation state about a UDDI node, update properties and policies, set publish tier limits, register UDDI publishers, and control value set support.

The operations of the UDDI registry Administrative Interface can be read and invoked using standard JMX (Java Management Extensions) interfaces. The use of JMX is explained in Using administrative programs (JMX).

Each WebSphere UDDI registry application registers an MBean with an MBean identifier of ‘UddiNode’. This MBean may be used by client applications to inspect and manage the runtime configuration of a UDDI application. This includes managing the activation state of and information about a UDDI node, updating properties and policies, setting publish tier limits, registration of UDDI publishers, and controlling value set support.

You can read and invoke the UddiNode attributes and operations using standard JMX interfaces. A client utility class UddiNodeProxy.java provides a ready-made application to connect to a UddiNode MBean and perform all the available operations. Example classes are also provided to drive UddiNodeProxy and demonstrate how to use the various UDDI management data types.

When WebSphere Application Server security is enabled, you can only invoke the operations of the UddiNode MBean if you are a user in an administrative role. The operations which make updates require the Administrator or Operator role, while get operations can be performed by Administrator, Operator, Configurator and Monitor roles.

UddiNodeProxy Usage

The following .jar files are required for compilation:

- *app_server_root*/plugins/com.ibm.uddi.jar
- *app_server_root*/runtimes/com.ibm.ws.admin.client.jar

The UddiNodeProxy class provides a utility method to programmatically interrogate the UddiNode MBean and output all the available attributes, operations and notifications to System.out. For each operation, the return type, operation name and parameter types are output as well as the impact property which indicates how the operation changes the state of the UddiNode MBean (and the UDDI node). As for all MBeans, the value for the impact property can be one of:

ACTION:
state of MBean will be changed

INFO: of the MBean remains unchanged and will return information

ACTION_INFO:
state of the MBean will change and return some information

UNKNOWN:
the impact of invoking the operation is not known

1. Invoke outputMBeanInterface:`uddiNode.outputMBeanInterface();`

   Expected output:

```
java.lang.String getNodeID() [INFO]
(getter for attribute nodeID)
java.lang.String getNodeState() [INFO]
(getter for attribute nodeState)
java.lang.String getNodeDescription() [INFO]
(getter for attribute nodeDescription)
java.lang.String getNodeApplicationName() [INFO]
(getter for attribute nodeApplicationName)
void activateNode() [ACTION]
(activates UDDI node)
void deactivateNode() [ACTION]
(deactivates UDDI node)
void initNode() [ACTION]
(initializes Uddi node)
com.ibm.uddi.v3.management.Property getProperty(java.lang.String propertyId) [INFO]
(returns UDDI Property)
com.ibm.uddi.v3.management.PolicyGroup getPolicyGroup(java.lang.String policyGroupId) [INFO]
(returns UDDI PolicyGroup)
com.ibm.uddi.v3.management.Policy getPolicy(java.lang.String policyId) [INFO]
(returns UDDI Policy)
void updatePolicy(com.ibm.uddi.v3.management.Policy policy) [ACTION]
(updates UDDI Policy)
void updateProperty(com.ibm.uddi.v3.management.ConfigurationProperty property) [ACTION]
(updates UDDI Property)
void updateProperties(java.util.List properties) [ACTION]
(updates collection of UDDI properties)
void updatePolicies(java.util.List policies) [ACTION]
(updates collection of UDDI policies)
java.util.List getProperties() [INFO]
(returns the collection of UDDI properties)
java.util.List getPolicyGroups() [INFO]
(returns collection of policy groups (note that the policies are not populated))
java.util.List getValueSets() [INFO]
(returns collection of value set status objects)
com.ibm.uddi.v3.management.ValueSetStatus getValueSetDetail(java.lang.String tModelKey) [INFO]
(returns status for a value set)
com.ibm.uddi.v3.management.ValueSetProperty getValueSetProperty(java.lang.String tModelKey,java.lang.
String valueSetPropertyId) [INFO]
(returns a property of a value set)
```

```
void updateValueSet(com.ibm.uddi.v3.management.ValueSetStatus valueSet) [ACTION]
(updates value set status)
void updateValueSets(java.util.List valueSets) [ACTION]
(updates multiple value sets)
void loadValueSet(java.lang.String filePath,java.lang.String tModelKey) [ACTION]
(loads values for a value set from a UDDI registry V3/V2 taxonomy data file.)
void loadValueSet(com.ibm.uddi.v3.management.ValueSetData valueSetData) [ACTION]
(loads values for a value set with the given tModel key.)
void changeValueSetTModelKey(java.lang.String oldTModelKey,java.lang.String newTModelKey) [ACTION]
(replaces all occurrences of values belonging to original tModelKey to new tModelKey.)
void unloadValueSet(java.lang.String tModelKey) [ACTION]
(unloads values for a value set with the given tModel key.)
java.lang.Boolean isExistingValueSet(java.lang.String tModelKey) [INFO]
(Determine if Value Set data exists for the given tModel key.)
java.util.List getTierInfos() [INFO]
(returns the collection of UDDI tier descriptions.)
java.util.List getLimitInfos() [INFO]
(returns the collection of UDDI limit descriptions.)
java.util.List getEntitlementInfos() [INFO]
(returns the collection of UDDI entitlements.)
com.ibm.uddi.v3.management.Tier getTierDetail(java.lang.String tierId) [INFO]
(returns UDDI Tier detail, specifying limits to the number of entities that can be published.)
com.ibm.uddi.v3.management.Tier createTier(com.ibm.uddi.v3.management.Tier tier) [ACTION]
(creates a UDDI Tier, specifying limits to the number of entities that can be published. Returns the
new tier ID.)
com.ibm.uddi.v3.management.Tier updateTier(com.ibm.uddi.v3.management.Tier tier) [ACTION]
(updates UDDI Tier details. Returns the updated Tier.)
void deleteTier(java.lang.String tierId) [ACTION]
(deletes the UDDI Tier, if it not in use.)
void setDefaultTier(java.lang.String tierId) [ACTION]
(Specifies the tier that auto registered UDDI publishers are assigned to.)
java.lang.Integer getUserCount(java.lang.String tierId) [INFO]
(returns the number of UDDI publisher within the specified tier.)
com.ibm.uddi.v3.management.TierInfo getUserTier(java.lang.String userId) [INFO]
(returns UDDI Tier information, specifying the tier this user belongs to.)
com.ibm.uddi.v3.management.UddiUser getUddiUser(java.lang.String userId) [INFO]
(returns UDDI user details, including tier and entitlements details.)
java.util.List getUserInfos() [INFO]
(returns the collection of UDDI user names and the tier they belong to.)
void createUddiUser(com.ibm.uddi.v3.management.UddiUser user) [ACTION]
(creates a new UDDI user.)
void createUddiUsers(java.util.List users) [ACTION]
(creates the collection of new UDDI users.)
void updateUddiUser(com.ibm.uddi.v3.management.UddiUser user) [ACTION]
(updates UDDI user details.)
void deleteUddiUser(java.lang.String userId) [ACTION]
(deletes UDDI publisher.)
void assignTier(java.util.List userIds,java.lang.String tierId) [ACTION]
(sets the tier for a List of users.)
notificationInfo: description=default UDDI event,descriptorType=notification,severity=(6),name=
uddi.node.event
notificationInfo: description=null,descriptorType=notification,severity=(6),name=jmx.attribute.
changed
```

See ManageNodeInfoSample class for sample code that demonstrates the attributes and operations described in this section.

Managing UDDI Node States and Attributes

UDDI nodes can be in one of several states, depending on the way the UDDI application was installed (as a default configuration or one where the administrator controls when initialization occurs). The UddiNode MBean provides four read only attributes: nodeID, nodeState, nodeDescription and nodeApplicationName. In addition the following MBean operations change UDDI node state: activateNode, deactivateNode and initNode.

nodeID

The node ID is the unique identifier for a UDDI node. If the UDDI application is installed as a default configuration the node ID is automatically generated. If the UDDI application is set up manually, the node ID is set by the administrator. It must be a valid UDDI key.

```
String nodeID = uddiNode.getNode();

System.out.println("node ID: " + nodeId);
```

nodeState

The nodeState attribute can have one of the following values:

nodeState value	English text associated with state
node.state.uninitialized	Not initialized
node.state.initialized	Initialized
node.state.initPending	Initialization pending
node.state.initInProgress	Initialization in progress
node.state.initMigrationPending	Migration pending
node.state.initMigration	Migration in progress
node.state.initValueSetCreationPending	Value set creation pending
node.state.initValueSetCreation	Value set creation in progress
node.state.activated	Activated
node.state.deactivated	Deactivated
node.state.unknown	Unknown

After installing a UDDI application using the default configuration, the UDDI node will be in activated state, that is, ready to receive and process UDDI API requests. The node ID and root key generator and some other properties are generated and cannot be changed. For a manually installed UDDI application where you want to specify the UDDI node ID and root key generator values, starting the UDDI application will put the UDDI node into initPending state. In this state, you can update all writable values up until the point you invoke the initNode operation. The initNode operation loads base tModels and value set data and writes all the configuration data to the UDDI node's database. During initialization the state is initInProgress. When initialization completes, the state changes momentarily to initialized and settles at activated. At this point the state can only be switched between activated and deactivated using deactivateNode and activateNode MBean operations.

Each node state value is in fact a message key which can be looked up in the messages.properties resource bundle. The attribute value can be retrieved using the getNodeState method of UddiNodeProxy:

1. Invoke getNodeState:

```
String nodeStateKey = uddiNode.getNodeState();
```

2. Look up translated text from ResourceBundle and output:

```
String messages = "com.ibm.uddi.v3.management.messages";

ResourceBundle bundle = ResourceBundle.getBundle(messages,
                                                 Locale.ENGLISH);

String nodeStateText =  bundle.getString(nodeStateKey);

System.out.println("node state: " + nodeStateText);
```

nodeDescription

You can get the administrator assigned description for the UDDI node using the getNodeDescription method of UddiNodeProxy:

1. Invoke getNodeDescription and output:

```
String nodeDescription = uddiNode.getNodeDescription();
System.out.println("node description: " + nodeDescription);
```

nodeApplicationName

The nodeApplicationName attribute is useful for discovering where the UDDI application that corresponds to the UDDI node is installed. The value will be a concatenation of the cell, node and server names, separated by colons. Retrieve the application location using the getApplicationId method of UddiNodeProxy:

1. Invoke getApplicationId and output:

```
String nodeApplicationId = uddiNode.getApplicationId();

System.out.println("node application location: " +
                                        nodeApplicationId);
```

activateNode

Changes the state of the UDDI node to activated, if the UDDI node was previously deactivated.

1. . Invoke activateNode:

```
uddiNode.activateNode();
```

deactivateNode

Changes the state of the UDDI node to deactivated, if the UDDI node was previously activated.

1. Invoke deactivateNode:

```
uddiNode.deactivateNode();
```

initNode

Causes UDDI node initialization, and when this completes the state of the UDDI node is ‘activated’.

1. Invoke initNode:

```
uddiNode.initNode();
```

Managing Configuration Properties

UDDI node runtime behavior is affected by the setting of several configuration properties. The UddiNode MBean provides operations to inspect and update their values, as follows: getProperties, getProperty, updateProperty and updateProperties.

See ManagePropertiesSample class for sample code that demonstrates the operations described in this section.

getProperties

Returns collection of all configuration properties as ConfigurationProperty objects.

1. Invoke getProperties:

```
List properties = uddiNode.getProperties();
```

2. Cast each collection member to ConfigurationProperty:

```
if (properties != null) {
  for (Iterator iter = properties.iterator(); iter.hasNext();) {
       ConfigurationProperty property =
```

```
                        (ConfigurationProperty) iter.next();
                System.out.println(property);
            }
        }
```

Once you have the ConfigurationProperty objects you can inspect attributes like the ID, value, type, whether the property is read only, required for initialization, and get name and description message keys. For example, invoking the toString method returns results similar to:

```
ConfigurationProperty
id: operatorNodeIDValue
nameKey: property.name.operatorNodeIDValue
descriptionKey: property.desc.operatorNodeIDValue
type: java.lang.String
value: uddi:capnscarlet:capnscarlet:server1:default
unitsKey:
readOnly: true
required: true
usingMessageKeys: false
validValues: none
```

You can use the nameKey and descriptionKey values to look up the translated name and description for a given locale, using the messages.properties resource in the sample package.

getProperty

Returns ConfigurationProperty object with the specified ID. Available property IDs are specified in PropertyConstants together with descriptions of the purpose of the corresponding properties.

1. Invoke getProperty:

```
    ConfigurationProperty property =
uddiNode.getProperty(PropertyConstants.DATABASE_MAX_RESULT_COUNT);
```

2. To retrieve the value of the property you could use the getValue method which returns an Object, but in this case, the property is of type integer, so it's easier to retrieve the value using the convenience method getIntegerValue:

```
int maxResults = property.getIntegerValue();
```

updateProperty

Updates the value of the ConfigurationProperty object with the specified ID. Available property IDs are specified in PropertyConstants together with descriptions of the purpose of the corresponding properties. Although you can invoke the setter methods in a ConfigurationProperty object, the only value that is updated in the UDDI node is the value. So to update a property, the steps are typically:

1. Create a ConfigurationProperty object and set its ID:

```
    ConfigurationProperty defaultLanguage = new ConfigurationProperty();
defaultLanguage.setId(PropertyConstants.DEFAULT_LANGUAGE);
```

2. Set the value:

```
defaultLanguage.setStringValue("ja");
```

3. Invoke updateProperty:

```
uddiNode.updateProperty(defaultLanguage);
```

updateProperties

Updates several ConfigurationProperty objects in a single request. Set up the ConfigurationProperty objects as for the updateProperty operation.

1. Add updated properties to a List:

```
List updatedProperties = new ArrayList();

updatedProperties.add(updatedProperty1);
updatedProperties.add(updatedProperty2);
```

2. Invoke updateProperties:

```
uddiNode.updateProperties(updatedProperties);
```

Managing Policies

Policies affecting behavior of the UDDI API are managed using the following UddiNode operations: getPolicyGroups, getPolicyGroup, getPolicy, updatePolicy and updatePolicies.

See ManagePoliciesSample class for sample code that demonstrates the attributes and operations described in this section.

getPolicyGroups

Returns collection of all policy groups as PolicyGroup objects.

1. Invoke getPolicyGroups:

```
List policyGroups = uddiNode.getPolicyGroups();
```

2. Cast each collection member to PolicyGroup:

```
if (policyGroups != null) {
for (Iterator iter = policyGroups.iterator(); iter.hasNext();) {
  PolicyGroup policyGroup = (PolicyGroup) iter.next();
  System.out.println(policyGroup);
}
}
```

Each policy group has an ID, name and description key, which you can look up in the messages.properties resource in the sample package. Although the PolicyGroup class does have a getPolicies method, PolicyGroup objects that are returned by the getPolicyGroups operation do not contain any Policy objects. Because of this behavior, clients can determine the known policy groups, and their IDs, without retrieving the entire set of policies in one request. To retrieve the policies within a policy group, use the getPolicyGroup operation.

getPolicyGroup

Returns the PolicyGroup object with the supplied ID.

1. Convert policy group ID to a String:

```
String groupId = Integer.toString(PolicyConstants.REG_APIS_GROUP);
```

2. Invoke getPolicyGroup:

```
PolicyGroup policyGroup = uddiNode.getPolicyGroup(groupId);
```

getPolicy

Returns the Policy object for the specified ID. Like a ConfigurationProperty, a Policy object has an ID, name and description keys, type, value and indicators specifying if the policy is read only or required for node initialization.

1. Convert policy ID to a String:

```
String policyId = Integer.toString(
            PolicyConstants.REG_AUTHORIZATION_FOR_INQUIRY_API);
```

2. Invoke getPolicy:

```
Policy policy = uddiNode.getPolicy(policyId);
```

updatePolicy

Updates the value of the Policy object with the specified ID. Available policy IDs are specified in PolicyConstants together with descriptions of the purpose of the corresponding policies. Although you can invoke the setter methods in a Policy object, the only value that is updated in the UDDI node is the value. So to update a policy, the steps are typically:

1. Create a Policy object and set its ID:

```
Policy updatedPolicy = new Policy();
String policyId =
      Integer.toString(PolicyConstants.REG_SUPPORTS_UUID_KEYS);
updatedPolicy.setId(policyId);
```

2. Set the value:

```
updatedPolicy.setBooleanValue(true);
```

3. Invoke updatePolicy:

```
uddiNode.updatePolicy(updatedPolicy);
```

updatePolicies

Updates several Policy objects in a single request. Set up the Policy objects as for the updatePolicy operation.

1. Add updated policies to a List:

```
List updatedPolicies = new ArrayList();

updatedPolicies.add(updatedPolicy1);
updatedPolicies.add(updatedPolicy2);
```

2. Invoke updatePolicies:

```
uddiNode.updatePolicies(updatedPolicies);
```

Managing Tiers

Tiers control how many of each type of UDDI entities a publisher can save in the UDDI registry. A tier has an ID, an administrator defined name and description, and a set of limits, one for each type of entity. Tiers are managed using the following UddiNode operations: createTier, getTierDetail, getTierInfos, getLimitInfos, setDefaultTier, updateTier, deleteTier and getUserCount.

See ManageTiersSample class for sample code that demonstrates the attributes and operations described in this section.

createTier

Creates a new tier, with specified publish limits for each UDDI entity.

1. Set tier name and description in a TierInfo object.

```
String tierName = "Tier 100";
String tierDescription = "A tier with all limits set to 100.";

TierInfo tierInfo = new TierInfo(null, tierName, tierDescription);
```

2. Define Limit objects for each UDDI entity:

```
 List limits = new ArrayList();

Limit businessLimit = new Limit();
businessLimit.setIntegerValue(100);

businessLimit.setId(LimitConstants.BUSINESS_LIMIT);

Limit serviceLimit = new Limit();
serviceLimit.setIntegerValue(100);
serviceLimit.setId(LimitConstants.SERVICE_LIMIT);
```

```
Limit bindingLimit = new Limit();
bindingLimit.setIntegerValue(100);
bindingLimit.setId(LimitConstants.BINDING_LIMIT);

Limit tModelLimit = new Limit();
tModelLimit.setIntegerValue(100);
tModelLimit.setId(LimitConstants.TMODEL_LIMIT);

Limit assertionLimit = new Limit();
assertionLimit.setIntegerValue(100);

assertionLimit.setId(LimitConstants.ASSERTION_LIMIT);
limits.add(businessLimit);
limits.add(serviceLimit);
limits.add(bindingLimit);
limits.add(tModelLimit);
limits.add(assertionLimit);
```

3. Create Tier object:

```
Tier tier = new Tier(tierInfo, limits);
```

4. Invoke create Tier and retrieve created tier:

```
Tier createdTier = uddiNode.createTier(tier);
```

5. Inspect generated tier ID of created tier:

```
tierId = createdTier.getId();
System.out.println("created tier has ID: " + tierId);
```

getTierDetail

Returns the Tier object for the given tier ID. The Tier class has getter methods for the tier ID, tier name and description (as set by the administrator), and the collection of Limit objects which specify how many of each UDDI entity type may be published by UDDI publishers allocated to the tier. The isDefault method indicates whether the tier is the default tier, that is, the tier that is allocated to UDDI publishers when auto registration is enabled.

1. Invoke getTierDetail:

```
Tier tier = uddiNode.getTierDetail("2");
```

updateTier

Updates tier contents with the supplied Tier object.

1. Update an existing Tier object (which may have been newly instantiated, or returned by the getTierDetail or createTier operations). This example retains the tier name and description, and all the limit values except the limit being updated:

```
modifiedTier.setName(tier.getName());
modifiedTier.setDescription(tier.getDescription());

Limit tModelLimit = new Limit();
tModelLimit.setId(LimitConstants.TMODEL_LIMIT);
tModelLimit.setIntegerValue(50);

List updatedLimits = new ArrayList();
updatedLimits.add(tModelLimit);

modifiedTier.setLimits(updatedLimits);
```

2. Invoke updateTier:

```
uddiNode.updateTier(modifiedTier);
```

getTierInfos

Returns collection of lightweight tier descriptor objects (TierInfo) which contain the tier ID, and tier name and description values, and whether the tier is the default tier.

1. Invoke getTierInfos:

```
List tierInfos = uddiNode.getTierInfos();
```

2. Output content of each TierInfo:

```
if (tierInfos != null) {

  for (Iterator iter = tierInfos.iterator(); iter.hasNext();) {
     TierInfo tierInfo = (TierInfo) iter.next();
     System.out.println(tierInfo);
  }
}
```

setDefaultTier

Specifies the tier with the given tier ID is the default tier. The default tier is the tier that is allocated to UDDI publishers when auto registration is enabled. Typically this would be set to a tier with low publish limits to prevent casual users publishing too many entities.

1. Invoke setDefaultTier:

```
uddiNode.setDefaultTier("4");
```

deleteTier

Removes the tier with the given tier ID. Tiers can only be removed if they have no UDDI publishers assigned to them, and the tier is not the default tier.

1. Invoke deleteTier:

```
uddiNode.deleteTier("4");
```

getUserCount

Returns the number of UDDI publishers assigned to tier specified by the tier ID.

1. Invoke getUserCount:

```
Integer userCount = uddiNode.getUserCount("4");
System.out.println("users in tier 4: " + userCount.intValue());
```

getLimitInfos

Returns collection of Limit objects representing the limit values for each type of UDDI entity. Limits are used in Tier objects.

1. Invoke getLimitInfos:

```
List limits = uddiNode.getLimitInfos();
```

2. Output the ID and limit value for each Limit object:

```
for (Iterator iter = limits.iterator(); iter.hasNext();) {
  Limit limit = (Limit) iter.next();

  System.out.println("limit ID: "
                     + limit.getId()
                     + ", limit value: "
                     + limit.getIntegerValue());
}
```

Managing UDDI Publishers

UDDI publishers are managed using the UddiNode MBean operations createUddiUser, createUddiUsers, updateUddiUser, deleteUddiUser, getUddiUser, getUserInfos, getEntitlementInfos, assignTier, getUserTier. An example is provided for each, making use of the UddiNodeProxy client class.

See ManagePublishersSample class for sample code that demonstrates the attributes and operations described in this section.

createUddiUser

Registers a single UDDI publisher, in a specified tier, with specified entitlements. The UddiUser class represents the UDDI publisher, and this is constructed using a user ID, a TierInfo object which specifies the tier ID to allocate the UDDI publisher to, and a collection of Entitlement objects which specify what the UDDI publisher is permitted to do.

Tip: to allocate the UDDI publisher default entitlements, set the entitlements parameter to null.

1. Create the UddiUser object:

```
UddiUser user = new UddiUser("user1", new TierInfo("3"), null);
```

2. Invoke createUddiUser:

```
uddiNode.createUddiUser(user);
```

createUddiUsers

Registers multiple UDDI publishers. This example shows how to register 7 UDDI publishers in one call, with default entitlements.

1. Create TierInfo objects for tiers that publishers will be allocated to:

```
TierInfo tier1 = new TierInfo("1");
TierInfo tier4 = new TierInfo("4");
```

2. Create UddiUser objects for each UDDI publisher, specifying tier to allocate to:

```
UddiUser publisher1 = new UddiUser("Publisher1", tier4, null);
UddiUser publisher2 = new UddiUser("Publisher2", tier4, null);
UddiUser publisher3 = new UddiUser("Publisher3", tier4, null);
UddiUser publisher4 = new UddiUser("Publisher4", tier1, null);
UddiUser publisher5 = new UddiUser("Publisher5", tier1, null);
UddiUser cts1 = new UddiUser("cts1", tier4, null);
UddiUser cts2 = new UddiUser("cts2", tier4, null);
```

3. Add the UddiUser objects to a List:

```
List uddiUsers = new ArrayList();

uddiUsers.add(publisher1);
uddiUsers.add(publisher2);
uddiUsers.add(publisher3);
uddiUsers.add(publisher4);
uddiUsers.add(publisher5);
uddiUsers.add(cts1);
uddiUsers.add(cts2);
```

4. Invoke createUddiUsers:

```
uddiNode.createUddiUsers(uddiUsers);
```

updateUddiUser

Updates a UDDI publisher with the details in the supplied UddiUser object. This is typically used to change the tier of one UDDI publisher or update their entitlements. Tip: only supply the entitlements you want to update – the remainder of available entitlements will retain their existing value.

1. Create Entitlement objects with appropriate permission. (the entitlement IDs are found in EntitlementConstants:

```
Entitlement publishUuiDKeyGenerator =
    new Entitlement(PUBLISH_UUID_KEY_GENERATOR, true);
Entitlement publishWithUuidKey =
    new Entitlement(PUBLISH_WITH_UUID_KEY, true);
```

2. Add Entitlement objects to a List:

```
List entitlements = new ArrayList();
entitlements.add(publishUuiDKeyGenerator);
entitlements.add(publishWithUuidKey);
```

3. Update a UddiUser object with the updated entitlements:

```
user.setEntitlements(entitlements);
```

4. Invoke updateUddiUser:

```
uddiNode.updateUddiUser(user);
```

getUddiUser

Retrieves details about a UDDI publisher in the form of a UddiUser object. This specifies the UDDI publisher ID, information about the tier they are assigned to and the entitlements they possess.

1. Invoke getUddiUser:

```
UddiUser user1 = uddiNode.getUddiUser("user1");
```

2. Output the contents of UddiUser:

```
System.out.println("retrieved user: " + user1);
```

getUserInfos

Returns a collection of UserInfo objects. Each UserInfo represents a UDDI publisher known to the UDDI node, and the name of the tier they are allocated to. To get details about a specific UDDI publisher, including the tier ID, and entitlements, use the getUddiUser operation.

1. Invoke getUserInfos:

```
List registeredUsers = uddiNode.getUserInfos();
```

2. Output the UserInfo objects:

```
System.out.println("retrieved registered users: ");
System.out.println(registeredUsers);
```

getEntitlementInfos

Returns a collection of Entitlement objects. Each entitlement is a property that controls whether permission is granted to a UDDI publisher to perform a specified action.

1. Invoke getEntitlementInfos:

```
List entitlementInfos = uddiNode.getEntitlementInfos();
```

2. Specify where to find message resources:

```
String messages = "com.ibm.uddi.v3.management.messages";
ResourceBundle bundle = ResourceBundle.getBundle(
                                    messages, Locale.ENGLISH);
```

3. Iterate through the Entitlement objects, displaying the ID, name and description:

```
for (Iterator iter = entitlementInfos.iterator(); iter.hasNext();) {
  Entitlement entitlement = (Entitlement) iter.next();

  StringBuffer entitlementOutput = new StringBuffer();

  String entitlementId = entitlement.getId();
  String entitlementName =
              bundle.getString(entitlement.getNameKey());
```

```
String entitlementDescription =
              bundle.getString(entitlement.getDescriptionKey());

  entitlementOutput.append("Entitlement id: ");
  entitlementOutput.append(entitlementId);
  entitlementOutput.append("\n  name: ");
  entitlementOutput.append(entitlementName);
  entitlementOutput.append("\n  description: ");
  entitlementOutput.append(entitlementDescription);

  System.out.println(entitlementOutput.toString());
}
```

deleteUddiUser

Removes the UDDI publisher with the specified user name (ID) from the UDDI registry.

1. Invoke deleteUddiUser:

```
uddiNode.deleteUddiUser("user1");
```

assignTier

Assigns UDDI publishers with supplied IDs to the specified tier. This is useful when you want to restrict several UDDI publishers, perhaps by assigning them all to a tier that doesn't allow publishing of any entities.

1. Create list of publisher IDs:

```
List uddiUserIds = new ArrayList();

uddiUserIds.add("Publisher1");
uddiUserIds.add("Publisher2");
uddiUserIds.add("Publisher3");
uddiUserIds.add("Publisher4");
uddiUserIds.add("Publisher5");
uddiUserIds.add("cts1");
uddiUserIds.add("cts2");
```

2. Invoke assignTier:

```
uddiNode.assignTier(uddiUserIds, "0");
```

getUserTier

Returns information about the tier a UDDI publisher is assigned to. The returned TierInfo has getters methods for retrieving the tier ID, tier name, tier description, and whether the tier is the default tier.

1. Invoke getUserTier:

```
TierInfo tierInfo = getUserTier("Publisher3");
```

2. Output the contents of the TierInfo object:

```
System.out.println(tierInfo);
```

Managing Value Sets

Value sets are represented in a UDDI registry as value set tModels, with a UDDI types keyedReference with value 'categorization'. Such value sets are backed with a set of valid values and for user defined value sets, this data is loaded into the UDDI registry using UddiNode MBean operations (although it is more convenient to use the User defined value set tool for this purpose). Each value set can be controlled by policy as being supported or not supported. When a value set is supported by policy, it can be referenced within UDDI publish requests. The UddiNode operations available to manage value sets and their data are: getValueSets, getValueSetDetail, getValueSetProperty, updateValueSet, updateValueSets, loadValueSet, changeValueSetTModelKey, unloadValueSet and isExistingValueSet.

See ManageValueSetsSample class for sample code that demonstrates the attributes and operations described in this section.

getValueSets

Returns collection of ValueSetStatus objects.

1. Invoke getValueSets:

```
List valueSets = uddiNode.getValueSets();
```

2. Cast each element to ValueSetStatus and output contents:

```
for (Iterator iter = valueSets.iterator(); iter.hasNext();) {

  ValueSetStatus valueSetStatus = (ValueSetStatus) iter.next();
  System.out.println(valueSetStatus);
}
```

getValueSetDetail

Returns ValueSetStatus object for the given value set tModel key.

1. Invoke getValueSetDetail:

```
uddiNode.getValueSetDetail(
               "uddi:uddi.org:ubr:categorization:naics:2002");
```

2. Retrieve and display details:

```
String name = valueSetStatus.getName();
String displayName = valueSetStatus.getDisplayName();
boolean supported = valueSetStatus.isSupported();

System.out.println("name: " + name);
System.out.println("display name: " + displayName);
System.out.println("supported: " + supported);
```

3. Display value set properties:

```
List properties = valueSetStatus.getProperties();

for (Iterator iter = properties.iterator(); iter.hasNext();) {

  ValueSetProperty property = (ValueSetProperty) iter.next();
  System.out.println(property);
}
```

getValueSetProperty

Returns a property of a value set as a ValueSetProperty object. This is mainly for use by the administrative console to render properties of a value set as a row in a table. For example, one such property is the keyedReference which indicates whether the value set is checked.

1. Invoke getValueSetProperty:

```
uddiNode.getValueSetProperty(
               "uddi:uddi.org:ubr:categorization:naics:2002",
               ValueSetPropertyConstants.VS_CHECKED);
```

2. Read and display boolean value of the property:

```
boolean checked = valueSetProperty.getBooleanValue();

System.out.println("checked: " + checked);
```

updateValueSet

Updates value set status. Only the supported attribute can be updated (all other setter methods are used by the UDDI application).

1. Create a ValueSetStatus object specifying the tModel key and the updated supported value:

```
ValueSetStatus updatedStatus = new ValueSetStatus();
updatedStatus.setTModelKey(
               "uddi:uddi.org:ubr:categorization:naics:2002");
updatedStatus.setSupported(true);
```

2. Invoke updateValueSet:

```
uddiNode.updateValueSet(updatedStatus);
```

updateValueSets

Updates value set status for multiple value sets. As for the updateValueSet operation, only the supported attribute is updated.

1. Populate List with updated ValueSetStatus objects:

```
List valueSets = new ArrayList();

ValueSetStatus valueSetStatus = new ValueSetStatus();
valueSetStatus.setTModelKey(
              "uddi:uddi.org:ubr:categorization:naics:2002");
valueSetStatus.setSupported(false);
valueSets.add(valueSetStatus);

valueSetStatus = new ValueSetStatus();
 valueSetStatus.setTModelKey(
               "uddi:uddi.org:ubr:categorizationgroup:wgs84");
valueSetStatus.setSupported(false);
valueSets.add(valueSetStatus);

valueSetStatus = new ValueSetStatus();
valueSetStatus.setTModelKey(
               "uddi:uddi.org:ubr:identifier:iso6523:icd");
valueSetStatus.setSupported(false);
valueSets.add(valueSetStatus);
```

2. Invoke updateValueSets:

```
uddiNode.updateValueSets(valueSets);
```

loadValueSet

Loads values for a value set from a UDDI registry V3/V2 taxonomy data file on the local file system. Note: there is also a loadValueSet operation that takes a ValueSetData object but this is only for use by the user defined value set tool.

1. Invoke loadValueSet:

```
uddiNode.loadValueSet(
                     "/valuesets/myvalueset.txt",
                     "uddi:cell:node:server:myValueSet");
```

changeValueSetTModelKey

Any value set values that were allocated to one value set tModel are allocated to the new value set tModel.

1. Invoke changeValueSetTModelKey with old and new tModel keys:

```
uddiNode.changeValueSetTModelKey(
             "uddi:cell:node:server:myValueSet",
             "uddi:cell:node:server:myNewValueSet");
```

unloadValueSet

Unloads values for a value set with the given tModel key.

1. Invoke unloadValueSet:

```
uddiNode.unloadValueSet("uddi:myValueSet");
```

isExistingValueSet

Determines if value set data exists for the given tModel key.

1. Invoke isExistingValueSet and display result:

```
boolean exists = uddiNode.isExistingValueSet(
                 "uddi:uddi.org:ubr:categorization:naics:2002");
System.out.println("NAICS 2002 is a value set: " + exists);
```

User-defined value set support in the UDDI registry

The UDDI Version 3 registry provides the structure and modeling tools to find information within a registry effectively. Users can define multiple value sets and add custom value sets. In UDDI Version 2, this functionality was called custom taxonomy support.

Data is worthless if it is lost within a mass of other data and cannot be distinguished or discovered. If a client of UDDI cannot effectively find information within a registry, the purpose of UDDI is considerably compromised. Providing the structure and modeling tools to address this problem is at the heart of UDDI's design. The verification of data within UDDI is core to its mission of description, discovery and integration. It achieves this by several means.

It allows users to define multiple value sets that can be used in UDDI. In such a way, multiple classification schemes can be overlaid on a single UDDI entity. This capability allows organizations to extend the set of such systems UDDI registries support. One is not tied to a single system, but can rather employ several different classification systems simultaneously.

While default value sets are shipped with the product, the UDDI Version 3 registry provides tools enabling 'custom' value sets to be added, potentially enabling UDDI entities to be more specifically categorized when published and further enhancing the capability of client to find specific data.

These value sets can be either checked or unchecked, and this is indicated via a keyedReference in the categoryBag of the tModel that represents a value set (a ″categorization tModel″). These keyedReferences have the tModel key for uddi-org:types and are added to the categoryBag to further describe the behavior of the categorization tModel, as follows:

checked
: Marking a tModel with this classification asserts that it represents a categorization, identifier, or namespace tModel that has a validation service to check that category values are present in a specified value set.

unchecked
: Marking a tModel with this classification asserts that it represents a categorization, identifier, or namespace tModel that does not have a validation service.

The procedure defined below describes how to add additional user-defined value sets, and display their allowed values in the UDDI user console value set tree display. Rational Application Developer has a Web Services Explorer user interface that also allows addition and display of custom checked value sets. The publisher of a value set categorization tModel may specify a 'display name' for use in UDDI user console implementations.

Procedure for adding a user defined value set

To add a user defined value set to the UDDI registry, perform the following tasks:

1. Publish a categorization tModel
2. Load the user defined value set data

3. Set the value set to **supported** status using the Administrative console. To do this you must be a user in an administrative role. This means that user defined value sets cannot be added to the UDDI registry without administrator permission.

The checked value set will only be referenced when the above tasks are complete. Value set data must be provided for validating checked value sets.

Value set data may also be used by user consoles for unchecked value sets, but it is not a requirement and is usually only used for presentation of deprecated value sets, such as unspc-org:unspc and back-level compatibility.

If the value set is checked, any publish requests that have a categoryBag containing keyedReferences with the new categorization tModel will be validated. If there is value set data corresponding to the categorization tModel in the registry database, only valid values will be accepted. If there is no value set data in the database **all** values will be rejected, and the publish request will fail. If the categorization tModel is unchecked, all values will be allowed, regardless of whether there is a corresponding value set present in the UDDI registry database. The value set tModel is not available for use until the administrator enables support for it using the administrative console, or the JMX interface.

Suggested approach

To introduce a new value set:
1. Publish the categorization tModel with a keyedReference of type 'uddi-org:categorization:types' with a key value of **categorization**, a keyedReference of type 'uddi-org:categorization:types' with a Key Name of **'Checked value set'** and a Key Value of **'checked'**, or a Key Name of **'Unchecked value set'** and a Key Value of **'unchecked'** and a keyedReference of type 'uddi-org:categorization:general_keywords' supplying the value set display name (as described below).
2. Load user defined value set data into the UDDI registry database using the UDDIUserDefinedValueSet utility (described below).
3. Use the administrative Console to set the status of the value set to supported (as described in Value set settings). This can also be achieved directly using the JMX interface.

Note: The SOAP and EJB interfaces will be able to make use of categorization tModels as soon as they are published. However, the UDDI registry user console will require a restart of the UDDI application because it currently gathers its list of categorizations for use in the value set tree display when the application starts.

Publishing a Checked Categorization tModel

This section describes how to publish a checked categorization tModel with the **'Checked value set'** Key Name for use by a user defined value set.

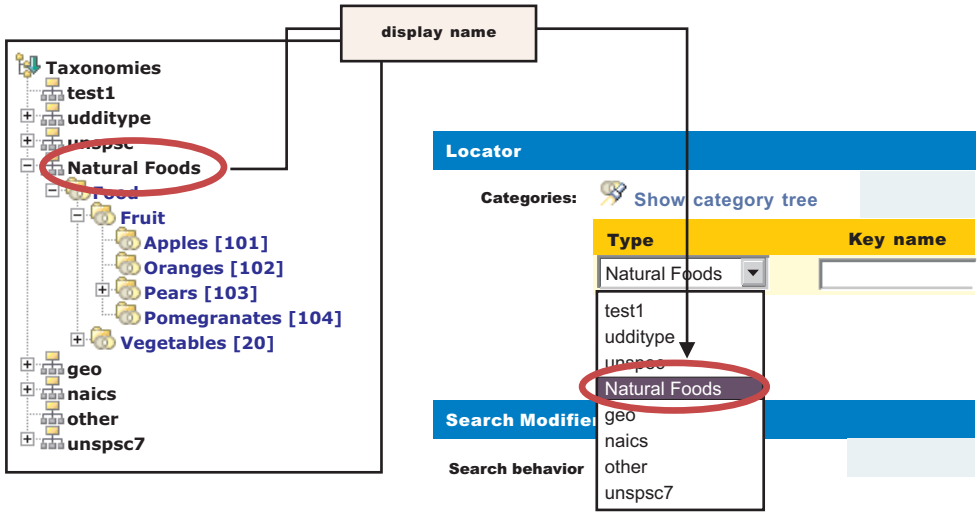
Publish a tModel to the UDDI registry with a categoryBag containing keyedReferences as follows:

Note	tModelKey	KeyName	KeyValue
1	uddi:uddi-org:categorization:types In the UDDI registry user interface this tModelKey can be chosen by selecting the category type of **UDDI Types**	**categorization**	**categorization**
2	uddi:uddi-org:categorization:types In the UDDI registry user interface this tModelKey can be chosen by selecting the category type of **UDDI Types**	**Checked value set**	**checked**

3	uddi:uddi-org:categorization:general_keywords In the UDDI registry user interface this tModelKey can be chosen by selecting the category type of **categorization:general_keywords**	**urn:x-ibm:uddi:customTaxonomy:displayName**	*<User Defined Value Set displayName>*

1. Indicates this tModel is a categorization tModel (required).
2. Indicates use of the tModel will be checked against a list of valid data (required). (Omitting this keyedReference, or explicitly specifying a value of 'unchecked' will indicate this categorization is unchecked).
3. Indicates special use of the general keywords value set, with a proprietary uniform resource name (URN) as the keyName value, defines a name for the user-defined value set that is intended for use in user console implementations where the full tModel name might be too long. The value can be 1-255 characters (inclusive) long.

The displayName is intended to provide a way to label a value set such that, when the UDDI user console displays it in a value set tree or in a pull-down list of available value sets, the meaning is clear to the user without being restricted to 8 characters and without needing to be the same as the published tModelName, which could be as long as 255 characters. An example is shown below:



The urn:x-ibm:customTaxonomy:displayName should be unique if only to avoid confusion when displayed in user interfaces but this is not validated.

To publish a new categorization tModel using SOAP, the message would be:

```
<save_tModel generic="3.0" xmlns="urn:uddi-org:api_v3">
  <authInfo></authInfo>>
  <tModel tModelKey="">
    <name>Natural Foods tModel</name>
    <categoryBag>
      <keyedReference tModelKey="uddi:uddi.org:categorization:types" keyName="categorization"
                      keyValue="categorization"/>
      <keyedReference tModelKey="uddi:uddi.org:categorization:types" keyName="Checked value set"
                      keyValue="checked"/>
      <keyedReference tModelKey="uddi:uddi.org:categorization:general_keywords"
```

```
                    keyName="urn:x-ibm:uddi:customTaxonomy:displayName" keyValue="Natural Foods"/>
    </categoryBag>
  </tModel>
</save_tModel>
```

Note: to specify an unchecked categorization substitute the key name **'Checked value set'** with **'Unchecked value set'** and '**checked'** Key Value with '**unchecked'** or, more simply, omit the keyedReference completely.

Loading User Defined Value Set Data

User Defined Value Set Data File Format

Value set data is identified by a unique code value, an optional description and a parent code that specifies its relationship with other code values. Value set data must adhere to this format:

Column name	Maximum length	Description of use
Code	765	Unique value within the value set used for validation
description	765	Typically used by UDDI user consoles and optionally in the keyedReference as the keyName value
parentcode	765	Indicates which existing **code** is the logical parent of this one, and is used in tree displays

Typically columns are delimited in the value set data file by '#' characters as in this example:

```
00#Food#00
10#Fruit#00
101#Apples#10
102#Oranges#10
103#Pears#10
1031#Anjou#103
1032#Conference#103
1033#Bosc#103
104#Pomegranates#10
20#Vegetables#00
201#Carrots#20
202#Potatoes#20
203#Peas#20
204#Sprouts#20
```

In the example, 'Food' is the description for the root node with child nodes of 'Fruit' and 'Vegetables' (both of these have parentcode values the same as the code value for 'Food').

The value set data in the example file could then be rendered in a tree like this:

```
Food
  Fruit
    Apples
    Oranges
    Pears
      Anjou
      Conference
      Bosc
    Pomegranates
  Vegetables
    Carrots
    Potatoes
    Peas
    Sprouts
```

The file must be saved in UTF-8 format.

Custom Taxonomy files used in UDDI Version 2 are also supported by the utility.

UDDIUserDefinedValueSet

A utility is provided to load value set data into the UDDI registry, assign existing value set data to another tModel and unload existing value set data. This utility uses the UDDI registry's JMX interface and therefore requires a number of connection parameters.

```
Usage: UDDIUserDefinedValueSet.sh '{'function'}' [options]

function:
-load <path> <key>         Load value set data from specified file
-newKey <oldKey> <newKey>  Move value set to a new tModel
-unload <key>              Unload existing value set

options:
-properties <path>         Specify location of configuration file
-host <host name>          Application Server or Deployment Manager host
-port <port>               SOAP Lister port number
-node <node name>          Node running a UDDI server
-server <server name>      Server with UDDI deployed
-columnDelimiter <delim>   Character delimiter to denote field end
-stringDelimiter <delim>   Character delimiter to denote strings


Connector security parameters

-userName <name>
-password <password>
-trustStore <path>
-trustStorePassword <password>
-keyStore <name>
-keyStorePassword <password>
```

Note: Ensure that the command window from which the UDDIUserDefinedValueSet is run is using a suitable codepage and font for displaying the characters contained in the value set name. Use of an incorrect codepage/font may result in unclear messages on a successful load, and create difficulty using the -unload and -newKey options.

The UDDIUserDefinedValueSet script is located in the *app_server_root*/bin directory.

If no connection parameters are supplied a connection is sought on the local host using the default SOAP port number of the deployment manager, or, if there is no deployment manager running, the default Application Server SOAP port number.

Command arguments are case insensitive.

Usage examples

Load a value set data for a tModel on the local UDDI registry using the percent sign as a column marker in the valuesetdata.txt file.

```
UDDIUserDefinedValueSet.sh -load valuesetdata.txt uddi:a708b8a7-35b5-451c-aafc-718ae071fcfe -columnDelimi
ter %
```

Move value set data from one checked tModel to another on a UDDI registry in a network deployment configuration.

```
UDDIUserDefinedValueSet.sh –newKey uddi:a708b8a7-35b5-451c-aafc-718ae071fcfe uddi:b819c9b8-46c6-562d-bb
0d-829bf1820d0f –host depmanagerhost.ibm.com –port 8879 –node uddinode –server uddiserver -override
```

Unload a value set from a tModel from a server with security turned on supplying the connection and security parameters in myproperties.properties file, but supplying the server and password arguments on the command line (which augment or override those contained in the properties file).

```
UDDIUserDefinedValueSet.sh –unload uddi:b819c9b8-46c6-562d-bb0d-829bf1820d0f –server uddiserver
–properties myproperties.properties –password myrealpassword
```

The configuration file, if specified by the optional **-properties** parameter, determines a number of optional parameters. These parameters can be specified on the command line and, if so, override the values in the properties file. These parameters are largely JMX connection parameters and security parameters.

The string.delimiter is typically used where a description value contains the same character as the column delimiter character. For example, if the column.delimiter was set to ',' (a comma), and there was a value set description value of 'Fruits, citrus', you could include this in the value set data file by setting the string.delimiter to ″ (double quote) and enclosing the description in quotes: 'Fruits, citrus'. Note that the quote character is escaped with a backslash ('\') to indicate the literal character is to be used.

If an attempt is made to load a value set to a tModel that has existing value set data, a warning message is given. To override this error provide the **-override** argument. This argument is also required if moving value set data to a new tModel using **-newKey** where the tModel is **checked**, and also unloaded value set data for a **checked** tModel.

Command line arguments and example data	Property and example data	Comments
-columnDelimiter #	column.delimiter=#	The column delimiter that is used in value set data files
-stringDelimiter \″	string.delimiter=\″	The field delimiter (this value must be different to the column.delimiter value)
-host ibm.com®	host=ibm.com	The host name of the system that is running the deployment manager or application server
-port 8880	port=8880	The SOAP port number of the deployment manager or application server
-node ibmNode	node=ibmNode	The name of the node that is running the server with the UDDI registry
-server server1	server=server1	The server that is running the UDDI registry
-userName ibmuser	security.username=ibmuser	The user name. This value is required if WebSphere Application Server security is turned on
-password mypassword	security.password=mypassword	The password
-trustStore /TrustStoreLocation	security.truststore=/TrustStoreLocation	The truststore file location
-keyStore ibmkeystore	security.keystore=ibmkeystore	The keystore name
-trustStorepassword trustpass	security.truststore.password=trustpass	The truststore password
-keyStorePassword keypass	security.keystore.password=keypass	The keystore password

Set the value set to supported

Use the administrative console to set the value set to **supported** by:

- Click *UDDI Nodes* > *<node>* and *Value Sets* (under Additional Properties on the right of the screen)

- Select the Value Set (by checking the box next to it)
- Click *Enable Support* above the list of Value Sets

Validation and Error Handling

The UDDI registry user console performs validation while a save tModel request is being built, that is, before the publish occurs. For example, if the user tries to add two customTaxonomy:displayName keyedReferences the following message is displayed:

```
Advice: Only one 'urn:x-ibm:uddi:customTaxonomy:displayName' key name is allowed for the 'Other' taxonomy.
```

If a keyedReference containing a keyName value that starts with 'urn:x-ibm:uddi:customTaxonomy:' is followed by anything other than 'displayName', the following message is displayed:

```
Advice: Only key name values of 'urn:x-ibm:uddi:customTaxonomy:displayName' are supported.
```

For requests where the save_tModel message may have multiple tModels, if any one of the tModels is a categorization tModel and it fails validation, the request fails with a UDDIInvalidValueException (plus additional information explaining the likely cause), and none of the tModels is published. For example:

```
E_invalidValue (20200) A value that was passed in a keyValue attribute did not pass validation.
This applies to checked categorizations, identifiers and other validated code lists. The error
text will clearly indicate the key and value combination that failed validation. Invalid 'custo
mTaxonomy:dbKey' keyValue [naics] in keyedReference. KeyValue already in use by tModelKey[UUID:
C0B9FE13-179F-413D-8A5B-5004DB8E5BB2]
```

UDDI Utility Tools

The UDDI Utility Tools is a suite of functions that you can use to migrate, move, or copy UDDI Version 2 entities, including child entities and their respective Version 2 entity keys, into a Version 3 UDDI registry.

Note: The UDDI Version 3 publish API supports publisher assigned keys (the Version 2 API did not) and promotion of entities between Version 3 registries can be achieved using normal API functions. UDDI Utility Tools supplied in this release is functionally equivalent to the version supplied in WebSphere Application Server 5.1. However, it is important to know that all UDDI Utility Tools functions in this release are performed using the UDDI Version 2 API. You can export from Version 2 and 3 registries (supplying only the Version 2 representation of the UDDI Entity key) and import into the Version 3 registry, using Version 2 API types. Entities from a Version 3 registry are exported as Version 2 entities and, as such, elements such as digital signatures will not be present. See section Saving Version 3 entities with a supplied key for an example on how to use the Version 3 API to assign your own keys to Version 3 entities.

Other uses of the tool include:

- Search and select entities from a source UDDI registry by specifying Version 2 keys or search criteria
- Publishing canonical tModels in a UDDI registry, including child entities
- Persist UDDI (Version 2) entities in an intermediate XML representation that can be used to customize and copy those entities to multiple target UDDI Registries, by specifying Version 2 Keys
- Update existing entities in a target UDDI registry, including child entities
- Delete selected entities from a target UDDI registry by specifying Version 2 keys

Use the UDDI Utility Tools by running the UDDIUtilityTools.jar file. This file is located in the *app_server_root*/UDDIReg/scripts directory. Alternatively, you can invoke all of the functions of UDDI Utility Tools through the supplied public Java API.

There are five main functions in UDDI Utility Tools:

Export

Given an entity type and key, or a list of entity types and keys, UDDI Utility Tools gets the UDDI entities from the specified registry and writes them to the UDDI Entity Definition File. The entity type for each key can be one of business, service, bindingTemplate or tModel. The Entity

Definition File contains XML that exactly describes each of the specified entities, according to the UDDI Utility Tools schema (which includes the UDDI Version 2 schema). The UDDI Entity Definition File separates entities by type, and automatically detects and records tModels referenced by the specified entities. You can use the 'referenced tModels' section of the file to ensure a target registry includes any referenced tModels before you try to import new entities to that registry.

Import

Given a list of UDDI entities (which can be supplied using the UDDI Entity Definition File generated by the export function, possibly with additional editing, or programmatically in a container object), the import function detects if the entities already exist in the target registry and, if they do not, creates a minimal entity (″stub″) with the specified key. The entities are then published updating the stubs with the supplied data and overwriting, or ignoring, existing entities as specified by the user. Note that the original key is maintained throughout.

Promote

Combines the export and import steps such that the specified entities are extracted (by key) from the source registry and then imported into the target registry in a single logical step. The generation of a UDDI Entity Definition File is optional for this function.

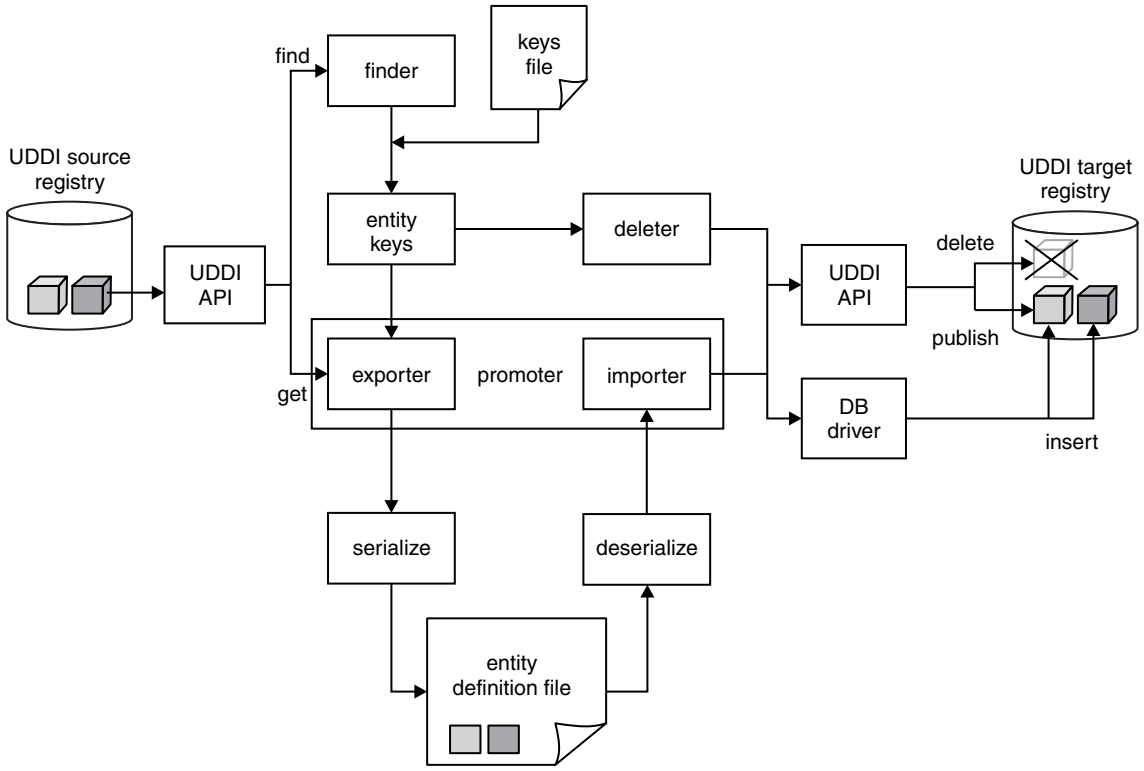
Delete Deletes the specified entities from the target UDDI registry. The entities to delete are specified as an entity type, or a list of entity types, and keys, in the same way as for the export function.

Find Matching Entities

Takes as input search criteria in the form of UDDI Inquiry API objects for each of the various entity types. The set of entities that match the search criteria are used to generate a list of entity keys, and this in turn can be used as input to the export, promote and delete functions.

Note: This function is available only through the programmatic API.

The relationship between the functions, their input and output, and the source and target UDDI Registries is shown in this conceptual overview diagram:



Setting up the configuration file

Configuration data for UDDI Utility Tools resides in a configuration properties file, which describes the runtime environment, UDDI and database locations and access information, logging information, security configuration, entity definition file location, and other flags to control whether referenced entities are to be imported and/or overwritten.

UDDI Utility Tools is distributed with a sample configuration properties file (UDDIUtilityTools.properties) and this is searched for by default in the current directory if no properties path is specified. By default, this file is located in the *app_server_root*/UDDIReg/scripts directory. Copy this sample file to a user writable location. Modify the file, according to the following list, and specify this modified file when running the utility tools.

- Set the classpath, which should include the current directory (.) and the UDDIUtilityTools.jar itself, plus all the dependent jars, which are listed in the Prerequisites section later on in this topic. The classpath must include the database driver jar (for example db2java.zip).

  If you are configuring a JSSE provider, add the .jar file which contains the provider to the classpath. The configuration of a JSSE provider is optional and is performed by setting the `jsse.provider` property. The default value is com.ibm.jsse.IBMJSSEProvider. To specify the FIPS JSSE provider set the value of the `jsse.provider` property to com.ibm.fips.jsse.IBMJSSEFIPSProvider.
- Set other properties, which are commented in the sample UDDIUtilityTools.properties file as shown below.
- Change localhost to the name of your server.
- Change the port number 9080 to your internal HTTP port.

```
############################################
# Runtime environment                      #
# (if invoking via java -jar...)           #
# "X Y" required around paths with spaces.   #
# Replace WAS_HOME with your WAS home path.  #
#                                          #
# db2java.jar is for DB2 - replace this with #
# appropriate database driver file.          #
############################################
classpath=.;WAS_HOME/UDDIReg/scripts/UDDIUtilityTools.jar;WAS_HOME/plugins/com.ibm.ws.runtime.jar;
WAS_HOME/plugins/com.ibm.uddi.jar;WAS_HOME/lib/j2ee.jar;/usr/lpp/db2810/db2810/jcc/classes/
db2java.jar

############################################
# SOAP entry points for source UDDI          #
# Replace localhost:9080 with the required   #
# values.                                  #
############################################
fromInquiryURL=http://localhost:9080/uddisoap/inquiryapi
fromGetURL=http://localhost:9080/uddisoap/get

############################################
# SOAP entry points for target UDDI          #
# Replace localhost:9080 with the required   #
# values.                                  #
############################################
toInquiryURL=http://localhost:9080/uddisoap/inquiryapi
toPublishURL=http://localhost:9080/uddisoap/publishapi

############################################
# UDDI Registry user information             #
#                                          #
# Note: this must match the user information #
# that was used to publish the entities on   #
# the target UDDI registry.                  #
############################################
userID=UNAUTHENTICATED
password=NONE

############################################
```

```
# Configuration for destination UDDI DB      #
# Replace DB2LOCATION with the DB2 location  #
##############################################
dbDriver=com.ibm.db2.jcc.DB2Driver
dbUrl=jdbc:db2:DB2LOCATION
dbUser=db2admin
dbPasswd=db2admin

##############################################
# Security provider configuration            #
##############################################
# Indicates whether security is required on the target registry
secure.connection=true

# The location of the truststore if security is required
trustStore.fileName=TrustFile.jks


# The password for the trust store
trustStore.password=WebAS

# The JSSE Provider class name
jsse.provider=com.ibm.jsse.IBMJSSEProvider

##############################################
# Trace and message logging configuration    #
##############################################
# detail level of message output (all functions)
verbose=true

# detail level of trace output.
# 1: severe
# 2: normal
# 3: detail
traceLevel=3

# path to message log file (relative or absolute)
messageLogFileName=logs/messages.log

# path to trace log file (relative or absolute)
traceLogFileName=logs/trace.log

##############################################
# Miscellaneous Options                      #
##############################################
# indicates if existing entities are overwritten (import/promote)
# Note: tModels in referencedTModels section are never overwritten,
#       regardless of this setting. To overwrite tModels, they must
#       be present in the tModels section.
overwrite=false

# indicates if referenced entities will be imported (import/promote)
importReferencedEntities=true

# location of entity definition file, used for (export/import)
UddiEntityDefinitionFile=definitions/entities01.xml

# namespace prefix to use in definition file (export)
namespacePrefix=promote
```

Prerequisites

To run the UDDI Utility Tools you must use the IBM Development Kit for Java code that is supplied with WebSphere Application Server. This Development Kit is located in *app_server_root*/java/bin.

Ensure that the following .jar files are available to the UDDI Utility Tools. The locations of the .jar files should be specified in the classpath property in the UDDI Utility Tools properties file:

UDDIUtilityTools.jar
This is the tools JAR itself and is located in *app_server_root*/UDDIReg/scripts.

com.ibm.uddi.jar
This file contains the UDDI4J classes and is located in *app_server_root*/plugins.

j2ee.jar
This file contains some required Java platform for enterprise applications classes, and is located in *app_server_root*/lib.

com.ibm.ws.runtime.jar
This is the Apache SOAP implementation and is located in *app_server_root*/plugins.

DbDriver
This is the driver needed to allow the UDDIUtilityTool to connect to your target database. See the table below for the values you need to specify for your chosen database:

	DB2	**Apache Derby**	
DBDriverLocation for classpath on the z/OS platform	*DB2_HOME*/jcc/classes/db2jcc.jar, *DB2_HOME*/jcc/classes/ db2jcc_license_cisuz.jar	*app_server_root*/derby/lib/ derbyclient.jar, *app_server_root*	
Driver on the z/OS platform	com.ibm.db2.jcc.DB2Driver	com.ibm.db2.jcc.DB2Driver	
URL on the z/OS platform	jdbc:db2://host:*database_port*/ *database_location*	jdbc:db2j:net://host:1527/ *database_name* (see note below)	

where

- *app_server_root* is the directory location of WebSphere Application Server.
- *DB2_HOME* is the directory location of DB2, for example c:\Program Files\SQLLIB\java12\
- *database_port* is the port that your DB2 database is listening on.
- *database_name* is the name of the Apache Derby database. Make sure that *database_name* includes the path to the database, for example *profile_root*/databases/com.ibm.uddi/UDDI30

Note:

- If you are using Apache Derby, make the database network enabled so that it can handle multiple connections. For further details, refer to the section about managing the Derby Network Server in the Derby Server and Administration Guide.
- If you are using DB2, add DB2_HOME/icc/lib to your LD_PATH_LIBRARY (or LD_LIBRARY_PATH) and LIBPATH environment variables.

The Security provider configuration section in the above properties file shows the location of the default DummyClientTrustFile.jks file. If you are using your own truststore, ensure that the location is placed here.

The UDDI Utility Tools use UDDI Version 2 SOAP inquiry and publish interfaces. These APIs are protected as described in Access control for UDDI registry interfaces. The UDDI Utility Tools also access the UDDI registry database through the database driver, and access to the database is controlled by the database management system.

The UDDI Entity Definition File

You generate this file by the export and promote functions, or you can choose to create it (either by hand, or by modifying a version of the file output by UDDI Utility Tools specifying the export function). It is the input to the import function.

Note: The extension to the uddi:tModel type to add a 'deleted' attribute is not currently used in UDDI Utility Tools.

The file is validated for well formedness and that it complies with the UDDI Utility Tools schema, shown here.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema id="uddiPromote" attributeFormDefault="unqualified" elementFormDefault="qualified"
 targetNamespace="http://www.ibm.com/xmlns/prod/WebSphere/UDDIUtilityTools" xmlns:xsd="http://www.w3.org
                  /2001/XMLSchema"
xmlns:uddi="urn:uddi-org:api_v2" xmlns="http://www.ibm.com/xmlns/prod/WebSphere/UDDIUtilityTools"
xmlns:promote="http://www.ibm.com/xmlns/prod/WebSphere/UDDIUtilityTools">

 <xsd:import namespace="http://www.w3.org/XML/1998/namespace" schemaLocation="xml.xsd" />
 <xsd:import namespace="urn:uddi-org:api_v2" schemaLocation="uddi_v2.xsd" />

 <!-- define a type to represent state of a tModel -->
 <xsd:simpleType name="tModelDeleted">
  <xsd:restriction base="xsd:NMTOKEN">
   <xsd:enumeration value="true" />
   <xsd:enumeration value="false" />
  </xsd:restriction>
 </xsd:simpleType>

 <!-- extend tModel with additional attribute of type tModelDeleted -->
 <!-- This is restricted to values true or false -->
 <xsd:complexType name="tModel">
  <xsd:complexContent>
   <xsd:extension base="uddi:tModel">
    <xsd:attribute name="deleted" type="promote:tModelDeleted" use="optional" />
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>


 <!-- Top level element definitions -->
 <xsd:element name="uddiEntities" type="promote:uddiEntities" />
 <xsd:complexType name="uddiEntities">
  <xsd:sequence>
   <xsd:element ref="promote:tModels" minOccurs="0" maxOccurs="1" />
   <xsd:element ref="promote:businesses" minOccurs="0" maxOccurs="1" />
   <xsd:element ref="promote:services" minOccurs="0" maxOccurs="1" />
   <xsd:element ref="promote:bindings" minOccurs="0" maxOccurs="1" />
   <xsd:element ref="promote:referencedTModels" minOccurs="0" maxOccurs="1" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="businesses" type="promote:businesses" />
 <xsd:complexType name="businesses">
  <xsd:sequence>
   <xsd:element ref="uddi:businessEntity" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="tModels" type="promote:tModels" />
 <xsd:complexType name="tModels">
  <xsd:sequence>
   <xsd:element ref="uddi:tModel" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="services" type="promote:services" />
 <xsd:complexType name="services">
  <xsd:sequence>
   <xsd:element ref="uddi:businessService" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
 </xsd:complexType>
```

```
 <xsd:element name="bindings" type="promote:bindings" />
 <xsd:complexType name="bindings">
  <xsd:sequence>
   <xsd:element ref="uddi:bindingTemplate" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="referencedTModels" type="promote:referencedTModels" />
 <xsd:complexType name="referencedTModels">
  <xsd:sequence>
   <xsd:element ref="uddi:tModel" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

UDDI Entity Definition File example for canonical tModels

The example Entity Definition File following shows the five main sections for tModels, businesses, services, bindings and referencedTModels:

UDDI Utility Tools can be used to create new UDDI entities in a target UDDI registry. A typical example of this is to introduce a new canonical tModel, which has a publicly known tModel key.

```
<?xml version="1.0" encoding="UTF-8"?>
<promote:uddiEntities xmlns="urn:uddi-org:api_v2" xmlns:promote="http://www.ibm.com/xmlns/prod/WebSphere/
                      UDDIUtilityTools">

  <!-- tModels -->
  <promote:tModels>

    <tModel tModelKey="uuid:ee3966a8-faa5-416e-9772-128554343571" >
      <name>http://schemas.xmlsoap.org/ws/2002/07/policytmodel</name>
      <description>WS-PolicyAttachment policy expression</description>
    </tModel>

    <tModel tModelKey="uuid:ad61de98-4db8-31b2-a299-a2373dc97212" >
      <name>uddi-org:wsdl:address</name>
 <description xml:lang="en">
This tModel is used to specify the URL fact that the address must be obtained from the WSDL deployment
file.
      </description>
      <overviewDoc>
        <overviewURL>
http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm#Address
        </overviewURL>
      </overviewDoc>
    </tModel>

  </promote:tModels>

  <!-- businesses -->
  <promote:businesses>
  </promote:businesses>

  <!-- services -->
  <promote:services>
  </promote:services>

  <!-- bindings -->
  <promote:bindings>
  </promote:bindings>

  <!-- referenced tModels -->
```

```
  <promote:referencedTModels>
  </promote:referencedTModels>

</promote:uddiEntities>
```

Starting UDDI Utility Tools at a command prompt

Ensure that you are using the correct level of Java code by setting the PATH statement to include the Java code that is supplied with WebSphere Application Server. For example, from the command line, type:

```
export PATH=app_server_root/java/bin:$PATH
```

UDDI Utility Tools can be started using:

java - jar UDDIUtilityTools.jar <function> [options]
: using a specified properties file that sets up classpath and other parameters, or it can be called using:

java CommandLineProcessor
: where CommandLineProcessor is the class which processes command line arguments for UDDI Utility Tools, sets up configuration and invokes the appropriate function.

Note: Before you run UDDIUtilityTools.jar from the command line, ensure that you have edited the UDDIUtilityTools.properties file. If you have saved this properties file in a different directory from the directory containing the UDDIUtilityTools.jar file, make sure you specify the location of the properties file as part of the command line arguments. See the Setting up the configuration file section earlier in this topic for more details.

The usage is as follows:

```
Usage: java -jar UDDIUtilityTools.jar {function} [options]

function:
  -promote <entity source>   Promote entities between registries
  -export <entity source>    Extract entities from registry to XML
  -delete <entity source>    Delete entities from registry
  -import                    Create entities from XML to registry

where <entity source> is one of:
  -tmodel|-business|-service|-binding <key> Specify single entity type and key
  -keysFile | -f <filename>  Specify file containing entity types and keys

options:
  -properties <filename>     Specify path to configuration file
  -overwrite | -o            Overwrite an entity if it already exists
  -log | -v                  Output verbose messages
  -definitionFile <filename> Specify path to UDDI entity definition file
  -importReferenced          Import entities referenced by source entities

The following options override property settings in configuration file:
  -overwrite
  -log
  -definitionFile
  -importReferenced

Example: java -jar UDDIUtilityTools.jar -promote -keysFile /uddikeys.txt
```

Below are a set of UDDI Utility Tools command line examples:

To export a single business to the EDF file specified in a properties file in the current directory.

```
java -jar UDDIUtilityTools.jar -export -business 28B8B928-2B2E-4EC9-A647-1E40651E4752
```

As above but this time using a keys file to specify the entities to be exported

```
java -jar UDDIUtilityTools.jar -export -keysFile /myKeyFiles/keyFile01.txt
```

As above but also specifying verbose output to appear on the command line.

```
java -jar UDDIUtilityTools.jar -export -keysFile /myKeyFiles/keyFile02.txt -v
```

To import the contents of the default EDF specified in a UDDIUtilitiyTools.properties file in the current directory.

```
java -jar UDDIUtilityTools.jar -import
```

As above but also specifying that referenced tModels should be imported into the target registry.

```
java -jar UDDIUtilityTools.jar -import -importReferenced
```

To import the entities from an EDF at the specified location.

```
java -jar UDDIUtilityTools.jar -import -definitionFile /myEDFs/entities01.xml
```

To import the entities from the default EDF including referenced tModels. Overwrite specifies that any entities excluding referenced tModels that are found in the target registry should be overwritten.

```
java -jar UDDIUtilityTools.jar -import -overwrite -importReferenced
```

To promote a single service from a source to a target registry using the properties file at a specified location.

```
java -jar UDDIUtilityTools.jar -promote -service 67961D67-330F-4F14-8210-E74A58E710F3
-properties /UUT/myUUTProps.properties
```

To promote a set of entities specified in a keys file.

```
java -jar UDDIUtilityTools.jar -promote -keysFile /myKeyFiles/keyFile03.txt
```

As above but specifying that existing entities in the target registry get overwritten.

```
java -jar UDDIUtilityTools.jar -promote -keysFile /myKeyFiles/keyFile04.txt -overwrite
```

To promote a set of entities specified in a keys file including referenced tModels.

```
java -jar UDDIUtilityTools.jar -promote -keysFile /myKeyFiles/keyFile05.txt -importReferenced
```

To promote a set of entities specified in a keys file but also create an EDF containing the promoted entities.

```
java -jar UDDIUtilityTools.jar -promote -keysFile /myKeyFiles/keyFile06.txt
-definitionFile /myEDFs/entities02.xml
```

To logically delete a single tModel. Note that it is not possible to physically delete tModels.

```
java -jar UDDIUtilityTools.jar -delete -tModel UUID:1E2B9D1E-E53D-4D36-9D46-6CCC176C466A
```

To delete all the entities specified in the keys file. Note that with the exception tModels all other entities will be physically deleted from the target registry.

```
java -jar UDDIUtilityTools.jar -delete -keysFile /myKeyFiles/keyFile04.txt
```

A keys file example

The following example shows the keys that are to be exported, promoted, or deleted from the target registry:

```
#
# Keys of entities to be exported, promoted from source registry or deleted from target registry
#
# Note: keys must be comma separated and on SAME line
# Note: property names are case sensitive. ('tmodels=' will be ignored)
```

```
businesses=97C77097-AC6C-4CA0-A6C4-452F7045C470, 4975E949-581F-4FCA-AD5F-E08280E05F9F
services=BB3864BB-1578-4833-8179-14391F14791F
bindings=
tModels=273F1727-7BFF-4FB5-A1FD-BA5C45BAFD9C
```

Note: If the importReferenced property is set to true, the list of tModels in the referencedTModels section is imported to the target registry. Minimal entities are created if the referencedTModel is new. If the referencedTModel already exists it is never overwritten, regardless of the overwrite property value. This is so that commonly referenced tModels such as categorization tModels do not keep being updated unnecessarily.

Should you need to update a referencedTModel, you must manually move the referencedTModel definition to the tModels section in the entity definition file and set overwrite to true.

Content of the log files

The following examples show the contents of two of the log files that are produced by running the tool. Some comments have been added in square brackets and in italic to highlight important points in the log file. The first is the messages.log, which shows successful and unsuccessful operations for export, import and delete functions:

```
[29/07/04 17:39:57:531 BST] CWUDU0002I: ********** Starting UDDI Utility Tools ********** [timestamp and
 eyecatcher indicate when tool is run]
[29/07/04 17:39:57:531 BST] CWUDU0009I: Exporting entities...
[29/07/04 17:39:57:531 BST] CWUDU0015I: Exported 14 entities.
[29/07/04 17:39:57:531 BST] CWUDU0029I: Serializing...
[29/07/04 17:39:57:531 BST] CWUDU0030I: Serialized entities.
[29/07/04 17:39:57:531 BST] CWUDU0016I: Importing entities...
[29/07/04 17:39:57:531 BST] CWUDU0124I: Created tModel minimal entity with tModelKey [uuid:667e2766-4781-
4151-b3a0-809f7180a096].
[29/07/04 17:39:57:531 BST] CWUDU0121I: Created business minimal entity with businessKey [263f5526-8708-4
834-9f5d-8f8c878f5d6e].
[29/07/04 17:39:57:531 BST] CWUDU0122I: Created service minimal entity with serviceKey [0af2a30a-be70-401
f-a027-331a6c332712].
[29/07/04 17:39:57:531 BST] CWUDU0122I: Created service minimal entity with serviceKey [61012761-d02c-4c7
0-ae98-435ffd4398f9].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [f97af9f9
-7cb7-47bd-8b90-b55e4db590df].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [17e4c017
-d273-43ec-af4a-f9b841f94a30].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [9e2c239e
-3b30-40a9-9c25-ce64edce25b9].
[29/07/04 17:39:57:531 BST] CWUDU0121I: Created business minimal entity with businessKey [49bb6949-4b0e-4
e81-88a7-e26bfbe2a7f1].
[29/07/04 17:39:57:531 BST] CWUDU0122I: Created service minimal entity with serviceKey [003d2b00-f6c0-407
1-8b84-f235a2f28445].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [df1019df
-2d2f-4f32-bf18-4f21274f1835].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [b229aeb2
-f2b1-4115-a06f-536753536f10].
[29/07/04 17:39:57:531 BST] CWUDU0122I: Created service minimal entity with serviceKey [84d8e584-2510-409
9-9b2a-6023f1602a0a].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [62a9a762
-7fff-4f7a-8463-af0c79af63ee].
[29/07/04 17:39:57:531 BST] CWUDU0123I: Created binding template minimal entity with bindingKey [e08654e0
-b212-42c0-bcf3-655e9765f392].
[29/07/04 17:39:57:531 BST] CWUDU0115I: Imported 7 entities and 0 referenced entities. [this kind of
message indicates the operation worked!]
[29/07/04 17:39:57:531 BST] CWUDU0002I: ********** Starting UDDI Utility Tools **********
[29/07/04 17:39:57:531 BST] CWUDU0023I: Deleting entities...
[29/07/04 17:39:57:531 BST] CWUDU0028I: Deleted 7 entities.
```

The second log file shows a typical trace log file entry for an export:

```
[29/07/04 17:39:57:531 BST] ********** Starting UDDI Utility Tools ********** [eyecatcher and timestamp
 indicate when tool is run]
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.PromoterAPI.setUddiEntities()  [the '>' indicates
entry to the constructor of this class]
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.export.KeyFileReader()
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() loaded tModel keys
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() loaded business keys
TransformConfiguration:
  nameSpacePrefix=promote
  uddiEntityDefinitionFile=C:\temp/MigToolFiles/Results/Promote_api_EDF_1.xml

ExportConfiguration:
  fromGetURL=http://yottskry:9080/uddisoap/
  fromInquiryURL=http://yottskry:9080/uddisoap/inquiryAPI

ImportConfiguration:
  overwrite=true
  uddiEntityDefinitionFile=C:\temp/MigToolFiles/Results/Promote_api_EDF_1.xml
  importReferencedEntities=true

PublishConfiguration:
  toInquiryURL=http://davep:9080/uddisoap/inquiryAPI
  toPublishURL=http://yottskry:9080/uddisoap/publishAPI
  userID=Publisher1
  trustStoreFileName=C:\WebSphere600/AppServer/etc/DummyClientTrustFile.jks
  secureConnection=false

DatabaseConfiguration:
  dbDriver=COM.ibm.db2.jcc.DB2Driver
  dbURL=jdbc:db2:LOC1
  dbUser=db2admin

LoggerConfiguration:
  messageStream=null
  messageLogFileName=C:\temp/MigToolFiles/logs/message.log
  traceLogFileName=C:\temp/MigToolFiles/logs/trace.log
  traceLevel=3
  verbose=true


[29/07/04 17:39:57:531 BST] < com.ibm.uddi.promoter.PromoterAPI()
[29/07/04 17:39:57:531 BST] ********** Starting UDDI Utility Tools **********
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.PromoterAPI.setUddiEntities()
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.export.KeyFileReader()
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() loaded tModel keys [ log
entries without a '>' or '<' are status messages only ]
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() loaded business keys
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() loaded service keys
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() loaded binding keys
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.UddiEntityKeys()
[29/07/04 17:39:57:531 BST] < com.ibm.uddi.promoter.UddiEntityKeys() [the '<' indicates exit from the
constructor]
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.export.KeyFileReader() removed duplicate, empty
and null keys
[29/07/04 17:39:57:531 BST] < com.ibm.uddi.promoter.export.KeyFileReader()
[29/07/04 17:39:57:531 BST] < com.ibm.uddi.promoter.PromoterAPI.setUddiEntities()
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.PromoterAPI.deleteEntities()
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.publish.EntityDeleter()
[29/07/04 17:39:57:531 BST] < com.ibm.uddi.promoter.publish.EntityDeleter()
[29/07/04 17:39:57:531 BST] > com.ibm.uddi.promoter.UDDIClient()
[29/07/04 17:39:57:531 BST]   com.ibm.uddi.promoter.UDDIClient() client type: 1
```

Starting UDDI Utility Tools through the API

UDDI Utility Tools provides a public API to functions for exporting, importing, promoting, finding and deleting UDDI entities. All of these functions can be invoked by using the PromoterAPI class. Usage of this class to perform these functions is typically to:

1. Create a Configuration object and populate it from a Properties object or from a configuration properties file.
2. Create a PromoterAPI object passing the Configuration in the constructor.
3. For keys based functions (export, delete and promote), set the keys by supplying a UDDIEntityKeys object, the location of the keys file, or, for one entity, by specifying an entity type and a key value.
4. Invoke the corresponding method for the function required: exportEntities, promoteEntities(boolean), importEntities, deleteEntities or extractKeysFromInquiry(FindTModel, FindBusiness, FindService, FindBinding, FindRelatedBusinesses).

There is some sample code for UDDI Utility Tools, demonstrating usage of the API classes, available from Samples Central.

The ″low-level″ API classes and methods have been deprecated in this release. Refer to the API documentation for details.

Known limitations with UDDI Utility Tools and workarounds

There are some known limitations with UDDI Utility Tools and a workaround for each. See UDDI troubleshooting tips for more information.

Embedded Apache Derby Restriction

The 'export' and 'delete' functions when referencing a source registry with an embedded Apache Derby database are supported. However, the 'import' and 'promote' functions are not supported when referencing a target registry because of a limitation with the UDDI registry when working with an embedded Apache Derby database. To allow the 'promote' and 'import' functions to work, the embedded Apache Derby database needs to be made network enabled. For information about configuring network Apache Derby, refer to the section about managing the Derby Network Server in the Derby Server and Administration Guide.

Saving Version 3 entities with a supplied key

An example of saving a Version 3 business with a defined key is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <save_business xmlns="urn:uddi-org:api_v3">
      <authInfo>a399c4a3-6387-47cd-a1bd-91f7bb91bdd7</authInfo>
      <businessEntity businessKey="uddi:mycompany-p1.com:computers">
        <name xml:lang="en">WithKey</name>
      </businessEntity>
    </save_business>
  </Body>
</Envelope>
```

Known limitations with UDDI Utility Tools and workarounds

There are known limitations with the UDDI Utility Tools and a workaround for each:

- PublisherAssertions are not supported and will not be promoted.

  Workaround: After the user has promoted the businesses that are related, he must recreate the publisherAssertion relationship.
- Referenced businesses in service projections are not added automatically to the EDF in the same manner as referenced tModels.

Workaround: Add the referenced business that will 'own' the projected service to the EDF. If the business is not present in the target registry, it should be placed before the service's owning business in the EDF.

- Cycle detection for service projections are not detected in the same manner as for referenced tModels.

  Workaround: If a circular reference is present between two or more service projections, break the cycle by removing one of the projections temporarily, perform the import and update the changed entity to reestablish the cycle in the target registry.
- tModels that were deleted (in the logical sense) in the source registry are imported and promoted as undeleted in the target registry. This is because, in the UDDI Version 2 specification, the deleted state of tModels is not exposed as API calls.

  Workaround: After importing the tModel, perform a delete. This is done using the UDDI Utility Tools delete function, or any other UDDI registry API access method.
- BindingTemplates referenced by hostingRedirectors are not added automatically to the EDF in the same manner as referenced tModels.

  Workaround: Add the referenced bindingTemplate to the EDF.
- Businesses referenced by an 'owningBusiness' keyedReference are not automatically added to the EDF.

  Workaround: Import the referenced business into the target registry before importing the tModel that references it.
- A few combinations of command line arguments are not validated and prevented, for example, it is possible to specify -import with -keysFile <path to file> in the same command, although the -keysFile is ignored.

Java API for XML Registries (JAXR) provider for UDDI

The Java API for XML Registries (JAXR) is a Java client API for accessing both UDDI (Version 2 only) and ebXML registries. It is part of the Java EE specification.

The JAXR API comprises the Java Platform, Enterprise Edition (Java EE) packages javax.xml.registry and javax.xml.registry.infomodel. There is Java EE API documentation at Web Services Reference.

The preferred UDDI Java client APIs are:

- UDDI4J Version 2, for UDDI Version 2
- UDDI Version 3 Client for Java, for UDDI Version 3

JAXR provider

The current JAXR specification (Version 1.0) defines a JAXR provider as an implementation of the JAXR API. Generally, a JAXR provider can be a JAXR provider for UDDI, a JAXR provider for ebXML, or a pluggable provider that supports both UDDI and ebXML. The JAXR provider for UDDI is a provider for UDDI only.

UDDI versions

A JAXR provider for UDDI accesses a UDDI registry using the UDDI Version 2 SOAP APIs only. The UDDI registry for UDDI Version 3 in this version of WebSphere Application Server supports the UDDI Version 1, 2 and 3 SOAP APIs. Therefore you can use the JAXR provider for UDDI to access this registry. You can also use the JAXR provider to access the UDDI registry for UDDI Version 2 in WebSphere Application Server Version 5.x.

To work with the UDDI Version 3 SOAP APIs, use the UDDI Version 3 Client for Java; you cannot use JAXR.

Capability level

The JAXR specification defines two capability profiles, capability level 0 and capability level 1. The JAXR API documentation categorizes each JAXR method as either level 0 or level 1. Generally, a JAXR provider for UDDI has capability level 0 and supports all level 0 methods, while A JAXR provider for ebXML has capability level 1 and supports all level 0 and level 1 methods. The JAXR provider for UDDI is a capability level 0 provider, and supports only level 0 methods.

JAXR for UDDI - getting started and further information

A sample program demonstrates how to get started with the Java API for XML Registries (JAXR). This topic also discusses class libraries, authentication and security, internal taxonomies, and logging and messages.

A simple sample

The following sample program shows how to obtain the ConnectionFactory instance, create a Connection to the registry and save an Organization in the registry.

```
import java.net.PasswordAuthentication;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Properties;
import java.util.Set;

import javax.xml.registry.BulkResponse;
import javax.xml.registry.BusinessLifeCycleManager;
import javax.xml.registry.Connection;
import javax.xml.registry.ConnectionFactory;
import javax.xml.registry.JAXRException;
import javax.xml.registry.RegistryService;
import javax.xml.registry.infomodel.Key;
import javax.xml.registry.infomodel.Organization;

public class JAXRSample
{
    public static void main(String[] args) throws JAXRException
    {
        //Tell the ConnectionFactory to use the JAXR Provider for UDDI
        System.setProperty("javax.xml.registry.ConnectionFactoryClass",
                                                    "com.ibm.xml.registry.uddi.ConnectionFactoryImpl");
        ConnectionFactory connectionFactory = ConnectionFactory.newInstance();

        //Set the URLs for the UDDI inquiry and publish APIs.
        //These must be the URLs of the UDDI version 2 APIs.
        Properties props = new Properties();
        props.setProperty("javax.xml.registry.queryManagerURL", "http://localhost:9080/uddisoap/inquiryapi");
        props.setProperty("javax.xml.registry.lifeCycleManagerURL", "http://localhost:9080/uddisoap/publishapi");
        connectionFactory.setProperties(props);

        //Create a Connection to the UDDI registry accessible at the above URLs.
        Connection connection = connectionFactory.createConnection();

        //Set the user ID and password used to access the UDDI registry.
        PasswordAuthentication pa = new PasswordAuthentication("Publisher1", new char[] { 'p', 'a', 's',
                                                                                  's', 'w', 'o', 'r', 'd' });
        Set credentials = new HashSet();
        credentials.add(pa);
        connection.setCredentials(credentials);

        //Get the javax.xml.registry.BusinessLifeCycleManager interface, which contains
        //methods corresponding to UDDI publish API calls.
        RegistryService registryService = connection.getRegistryService();
        BusinessLifeCycleManager lifeCycleManager = registryService.getBusinessLifeCycleManager();

        //Create an Organization (UDDI businessEntity) with name "Organization 1".
        Organization org = lifeCycleManager.createOrganization("Organization 1");
```

```
        //Add the Organization to a Collection, ready to be saved in the UDDI registry.
        Collection orgs = new ArrayList();
        orgs.add(org);

        //Save the Organization in the UDDI registry.
        BulkResponse bulkResponse = lifeCycleManager.saveOrganizations(orgs);

        //Obtain the Organization's Key (the UDDI businessEntity's businessKey) from the response.
        if (bulkResponse.getExceptions() == null)
        {
            //1 Organization was saved, so 1 key will be returned in the response collection
            Collection responses = bulkResponse.getCollection();
            Key organizationKey = (Key)responses.iterator().next();
            System.out.println("\nOrganization Key = " + organizationKey.getId());
        }
    }
}
```

Classpath

The class libraries of the JAXR Provider for UDDI are contained within the com.ibm.uddi_1.0.0.jar file, located in the *app_server_root*/plugins directory. When using the JAXR API from within a J2EE application running under WebSphere Application Server, all required classes will automatically be on the classpath. When using the JAXR API from outside this environment, the following jars must be on the Java classpath:

- *app_server_root*/lib/bootstrap.jar
- *app_server_root*/lib/j2ee.jar
- *app_server_root*/plugins/com.ibm.uddi_1.0.0.jar
- *app_server_root*/plugins/com.ibm.ws.runtime_6.1.0

javax.xml.registry.ConnectionFactory

To use the JAXR Provider for UDDI, the name of the ConnectionFactory implementation class must first be specified by setting the System Property “javax.xml.registry.ConnectionFactoryClass” to “com.ibm.xml.registry.uddi.ConnectionFactoryImpl”. Failure to specify this will result in the value defaulting to “com.sun.xml.registry,common.ConnectionFactoryImpl”, which will not be found. This will result in a JAXRException when the ConnectionFactory.newInstance() method is called. The JAXR Provider for UDDI does not support lookup of the ConnectionFactory via JNDI.

javax.xml.registry.Connection Properties

Connection specific properties must be specified by setting a java.util.Properties object on the JAXR ConnectionFactory before obtaining a Connection. The JAXR specification defines the full list of these properties. The table below lists the three most important properties, and what values they should take in order to use the JAXR Provider for UDDI to access the UDDI registry. The only required Connection property is “javax.xml.registry.queryManagerURL”, however it is recommended that “javax.xml.registry.lifeCycleManagerURL” is also set, and that the default value of “javax.xml.registry.security.authenticationMethod” is understood. The rest of the Connection properties defined in the JAXR specification are optional, and their values are not specific to the UDDI registry. The JAXR Provider for UDDI does not define any additional provider-specific properties.

Property	Description
javax.xml.registry.queryManagerURL	The URL of the UDDI registry’s inquiry API for UDDI Version 2. Typically this will be of the form: “http://<hostname>:<port>/uddisoap/inquiryapi”. This property is required.

javax.xml.registry.lifeCycleManagerURL	The URL of the UDDI registry's publish API for UDDI v2. Typically this will be of the form: "http://<hostname>:<port>/uddisoap/publishapi". If this property is not specified, it defaults to the value of the javax.xml.registry.queryManagerURL property, however the UDDI registry will typically have different URLs for the inquiry and publish APIs, and it is recommended to specifiy both properties.
javax.xml.registry.authenticationMethod	The method of authentication to use when authenticating with the registry. This may take one of two values, "UDDI_GET_AUTHTOKEN" and "HTTP_BASIC". The default value is "UDDI_GET_AUTHTOKEN" if none is specified. See section Authentication and Security below for more information.

Authentication and security

Authentication

The javax.xml.registry.authenticationMethod Connection property tells the JAXR Provider which method to use when authenticating with the UDDI registry. The two supported values of this property are "UDDI_GET_AUTHTOKEN" and "HTTP_BASIC". The JAXR Provider for UDDI does not support the "CLIENT_CERTIFICATE" or "MS_PASSPORT" methods of authentication. If this property is not set, the default authentication method is "UDDI_GET_AUTHTOKEN".

UDDI_GET_AUTHTOKEN

The JAXR Provider uses the UDDI V2 get_authToken API to authenticate with the registry. The get_authToken call is made automatically by the JAXR Provider when the Connection credentials are set, and the UDDI V2 authToken returned by the call is saved by the JAXR Provider for use on subsequent UDDI publish API calls.

HTTP_BASIC

The JAXR Provider uses HTTP basic authentication to authenticate with the registry. This is supported by WebSphere Application Server when security is on. No UDDI V2 get_authToken API call is made, instead the username and password are sent in the HTTP headers using HTTP basic authentication every time a UDDI API call is made (both inquiry and publish). If the UDDI registry does not require HTTP basic authentication, the credentials are ignored.

The JAXR Provider uses UDDI Version 2 SOAP inquiry and publish APIs. These APIs are protected as described in Access control for UDDI registry interfaces.

USING SSL (Secure Sockets Layer)

SSL can be used to encrypt HTTP traffic between the JAXR Provider for UDDI and the UDDI registry. To use SSL, the JAXR client program should do the following:

1. When setting the "javax.xml.registry.queryManagerURL" and "javax.xml.registry.lifeCycleManagerURL" Connection properties, specify a URL with the protocol "https" and the correct port for using SSL to access the UDDI registry. The UDDI registry's default port for HTTPS is 9443. Often only the lifeCycleManager URL (the UDDI Publish API URL) will require SSL.
2. Add a new Security Provider to the java.security.Security object, according to the JSSE (Java Secure Sockets Extension) implementation being used. If running under the JVM provided in WebSphere Application Server, the JSSE provided by IBM will automatically be on the classpath. Add the IBM Security Provider as follows:

   ```
   java.security.Security.addProvider(new com.ibm.jsse.JSSEProvider());
   ```

3. Set the System property “javax.net.ssl.trustStore” to be the file name of the client trust store file. The client trust store file is a Java key store (.jks) file and must contain the server certificate of the UDDI registry. Key store files can be managed using the ikeyman tool
4. Set the System property “javax.net.ssl.trustStorePassword”. This is the password used to open the client trust store file.
5. If using an IBM version of JVM that is older than the level provided in this version of WebSphere Application Server, it may be necessary to set the System property “java.protocol.handler.pkgs” to “com.ibm.net.ssl.internal.www.protocol”. For more information on SSL and the ikeyman tool refer to SSL and IKEYMAN within this Information Center.

Internal taxonomies

The JAXR Provider for UDDI supplies the following internal taxonomies:

Taxonomy	ClassificationScheme name (UDDI tModel name)	ClassificationScheme id (UDDI Version 2 tModelKey)
NAICS 1997	ntis-gov:naics:1997	UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2
NAICS 2002	ntis-gov:naics:2002	UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2
UNSPSC 3.1	unspsc-org:unspsc:3-1	UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384
UNSPSC 7	unspsc-org:unspsc	UUID:CD153257-086A-4237-B336-6BDCBDCC6634
ISO3166 2003	ubr-uddi-org:iso-ch:3166-2003	UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88

The tModels corresponding to all of these taxonomies are available in the UDDI Version 3 registry. If using the JAXR Provider to access a UDDI Version 2 registry, only the tModels corresponding to NAICS 1997, UNSPSC 3.1 and ISO3166 are available.

Custom internal taxonomies

A user may supply their own custom internal taxonomies. To create a new custom internal taxonomy and make it available to the JAXR provider, follow these steps:

1. Create a text file containing the taxonomy element data. As an example, look at the file iso3166-2003-data.txt in plugins/com.ibm.uddi_1.0.0. This is the taxonomy data file for the supplied ISO 3166 taxonomy. The first few lines are:

```
iso3166#--#World#--
iso3166#AD#Andorra#--
iso3166#AE#United Arab Emirates#--
iso3166#AE-AJ#'Ajm?n#AE
iso3166#AE-AZ#Ab? Z?aby[Abu Dhabi]#AE
iso3166#AE-DU#Dubayy [Dubai]#AE
iso3166#AE-FU#Al Fujayrah#AE
iso3166#AE-RK#Ra's al Khaymah#AE
iso3166#AE-SH#Ash Sh?riqah [Sharjah]#AE
iso3166#AE-UQ#Umm al Qaywayn#AE
iso3166#AF#Afghanistan#--
iso3166#AF-BAL#Balkh#AF
iso3166#AF-BAM#B?m??n#AF
```

   Each line represents one element of the taxonomy, or one Concept in the taxonomy Concept tree. Each line has the form:

```
<taxonomy ID>#<element value>#<element name>#<parent element value>
```

Token	Description
<taxonomy ID>	The taxonomy ID is the same for every element of a taxonomy.
<element value>	The Concept value (UDDI keyValue).

<element name>	The Concept name (UDDI keyName).
<parent element value>	This defines the element's parent element in the taxonomy tree. For every element (except the root element) in the data file, there should be another line which defines the element's parent element. The root element is denoted by defining itself as its own parent. There should be only one root element, and no parentless elements.
#	The delimiter character. This does not have to be "#" and can be defined for each taxonomy in the taxonomyConfig.properties file.

2. Save a ClassificationScheme (UDDI tModel) in the UDDI registry to represent the new internal taxonomy. This can be done using the javax.xml.registry.BusinessLifeCycleManager.saveClassificationSchemes() method.
3. Add the new taxonomy to the taxonomyConfig.properties file:
   a. Copy the supplied taxonomyConfig.properties file from the root of the com.ibm.uddi_1.0.0.jar file.

      The content of the supplied taxonomyConfig.properties file is:

      ```
      naics-1997 = UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2, naics-1997-data.txt, #
      naics-2002 = UUID:1FF729F2-1948-46CF-B660-31EC107F1663, naics-2002-data.txt, #
      unspsc  = UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384, unspsc-data.txt,     #
      unspsc7_data = UUID:CD153257-086A-4237-B336-6BDCBDCC6634, unspsc7-data.txt,    #
      iso3166-2003 = UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88, iso3166-2003-data.txt,#
      ```

      This file has one line per supplied internal taxonomy, which is of the form:

      ```
      <taxonomy ID> = <tModelKey>,<data filename>,<data file delimiter>
      ```

Token	Description
<taxonomy ID>	This is used internally by the JAXR provider to identify each taxonomy. It does not have to be the same as the taxonomy ID in the corresponding taxonomy data file.
<tModelKey>	The tModelKey of the corresponding UDDI tModel. (The id of the corresponding JAXR ClassificationScheme).
<data filename>	The name of the corresponding taxonomy data file.
<data file delimiter>	The delimiter character used in the taxonomy data file. All supplied internal taxonomies use "#", but user-supplied internal taxonomies may use different delimiters.

   b. Add a new line for the new taxonomy to the copy of the taxonomyConfig.properties file. Do not remove any existing taxonomies from the file as this will make them unavailable to the JAXR provider.
4. Add the copied taxonomyConfig.properties file to the Java classpath ahead of jaxruddi.jar.
5. If any JAXR client programs are still running that were started before the new taxonomy was added to the taxonomyConfig.properties file, a new Connection must be created in order to pick up the new taxonomy.

Important notes on internal taxonomies

Each internal taxonomy is loaded into memory once per JAXR Connection. The taxonomy's ClassificationScheme is created when the Connection is created. At this time the associated UDDI tModel is obtained from the registry and used to populate the ClassificationScheme attributes. The taxonomy's Concept object tree is not created until the first time the ClassificationScheme is requested by the user. All subsequent requests for the same internal taxonomy using the same Connection will return the same object tree.

Modification of the Concept object tree

Because there is only one ClassificationScheme and Concept object tree per internal taxonomy per Connection, a user should not attempt to modify programmatically any part of the Concept tree, because all future requests for this taxonomy using the same Connection will return the modified (and now possibly invalid) objects. Programmatic modification of the Concept tree will not result in any changes to the associated taxonomy data file. If a user wishes to make a change to the values in a user-defined internal taxonomy, they must first make the changes in the taxonomy data file, and then create a new Connection to pick up the changes in a new Concept tree.

Modification of the ClassificationScheme

Similarly, a user should not attempt to modify programmatically an internal ClassificationScheme, except in the case where a user wishes to modify and then save a user-defined internal ClassificationScheme. A new Connection is not required to pick up programmatic changes.

Logging and messages

UDDI4J Logging

The JAXR Provider for UDDI uses UDDI4J Version 2 to communicate with the UDDI registry. UDDI4J has its own logging which can be switched on by setting the value of the System property "org.uddi4j.logEnabled" to "true". This outputs to the standard error log the XML request and response bodies of every UDDI request.

Trace

Entry, exit, exception, warning and debug trace is provided using commons-logging. See http://jakarta.apache.org/commons/logging/ for more information on commons-logging. Trace will only be created if the JAXR client configures it. Entry, exit and debug trace uses the debug level of logging. Exception and warning trace uses the info level of logging. Additionally, info level logging is provided before each UDDI4J request is made.

Standard error log messages

The InternalTaxonomyManager, EnumerationManager and PostalSchemeManager send warning messages to System.err if error conditions occur that do not warrant an exception, but that the user should be informed of. Examples of these are if a taxonomy data file contains an invalid line, or if a tModel corresponding to an internal taxonomy could not be found in the registry.

Setting up and deploying a new UDDI registry

A UDDI registry node consists of the UDDI registry application (an enterprise application that is supplied as part of WebSphere Application Server), a store of data (using a relational database management system) referred to as the UDDI database, and a means to connect the application to the data (a datasource and related elements). To set up a new UDDI registry you create the UDDI database and datasource, and deploy the supplied application.

Before you begin

Start WebSphere Application Server, and create a server to host the UDDI registry. Use Starting and stopping quick reference for information about starting WebSphere Application Server using either commands or the administrative console.

About this task

The subtopics describe how to create the UDDI database (which can be local or remote) and datasource, and how to deploy the UDDI registry application.

You can create either a *default* UDDI node or a *customized* UDDI node. The main difference between default and customized, in the context of these set up tasks, refers to a number of mandatory UDDI registry properties such as the UDDI node ID and description, and the prefix to be used for generated discovery URLs.

Default UDDI node
The mandatory properties are automatically set to default values and cannot be changed. A default UDDI node is a suitable option for initial evaluation of the UDDI registry, and for development and test purposes.

Customized UDDI node
You must set the mandatory properties, but once set they cannot be changed for this configuration. With a customized UDDI node you have more control over the database management system used for the UDDI database, and the properties used to set up the UDDI database. With a customized UDDI node, you create the UDDI database and datasource to your own specifications before deploying the UDDI registry application. A customized node is a suitable option for production purposes. To move from a default UDDI node to a customized node, see "Changing the UDDI registry application environment after deployment" on page 813.

- If you want to create a default UDDI registry with a database other than embedded Apache Derby, or if you want an embedded Apache Derby database but you want to create the datasource manually, follow the instructions in "Setting up a default UDDI node" on page 794.
- If you want to create a customized UDDI registry, follow the instructions in "Setting up a customized UDDI node" on page 803.

Database considerations for production use of the UDDI registry

The UDDI registry fully supports a number of databases and can be used for development and test purposes. However, there are factors to consider when you decide which database is appropriate for your anticipated UDDI registry production use.

It is important to consult the information that is supplied by your chosen database vendor, but you also need to consider the likely size and volume of requests, and whether the general performance and scalability of the UDDI registry is important.

For example, the Apache Derby database supports the full function of the UDDI registry, but it is not an enterprise level database and it does not have the same characteristics, for example, scaling or performance, as enterprise databases such as DB2.

Note: Apache Derby is not supported for production use.

If you need multiple connections to the UDDI registry database (for example to use the UDDI registry in a cluster configuration) and Apache Derby is your preferred database, you need to use the network option for Apache Derby. This is because embedded Apache Derby has a limitation that allows only one Java virtual machine to access or load a database instance at any one time. That is, two application servers cannot access the same Apache Derby database instance at the same time.

Note: The UDDI registry can support multiple users, even if a single database connection exists.

Related concepts

"Overview of the Version 3 UDDI registry" on page 747
The Universal Description, Discovery, and Integration (UDDI) specification defines a way to publish and discover information about Web services. The term *Web service* describes specific business functionality that is exposed by a company, usually through an Internet connection, to allow another company, its subsidiaries, or software program, to use the service.

Related reference

About Apache Derby
Use Apache Derby as a test and development database only. Apache Derby must run at a minimal version of v10.3 or Cloudscape v10.1x. The Apache Derby package that is bundled with the application server is backed by full IBM Quality Assurance (QA).

Data source minimum required settings for Apache Derby
These properties vary according to the database vendor requirements for JDBC driver implementations. You must set the appropriate properties on every data source that you configure. These settings are for Apache Derby and Cloudscape data sources.

Setting up a default UDDI node

Use this task to create a UDDI node with predetermined property values. This UDDI node is suitable for initial evaluation of the UDDI registry and for development and test purposes.

About this task

You cannot change the mandatory node properties, such as node ID, either during the creation of the node, or afterward. If you want to choose your own mandatory node properties, set up a customized node, as detailed in "Setting up a customized UDDI node" on page 803.

Note: If you are deploying the UDDI registry application into a cluster, and you want to use Apache Derby for your database, you must use the network version of Apache Derby. Embedded Apache Derby is not supported for cluster configurations.

1. Create a database schema to hold the UDDI registry by completing one of the following tasks, ensuring that you use the default node options where specified:
   - "Creating a DB2 distributed database for the UDDI registry" on page 795
   - "Creating a DB2 for z/OS database for the UDDI registry" on page 796
   - "Creating an Apache Derby database for the UDDI registry" on page 798

   Note: If you are creating the UDDI node in a cluster, it is assumed that a single database is used for all members of the cluster.
2. Set up a data source for the UDDI registry application to use to access the database, as described in "Creating a data source for the UDDI registry" on page 799.
3. Deploy the UDDI registry application, as described in "Deploying the UDDI registry application" on page 801.
4. Click **Applications** → **Application Types** → **WebSphere enterprise applications** to display the installed applications. Start the UDDI registry application by selecting the check box next to it and clicking **Start**. Alternatively, if the application server is not already running, start the application server. This action automatically starts the UDDI registry application. The UDDI node is now active.

   Note: Restarting the UDDI application, or the application server, always reactivates the UDDI node, even if the node was previously deactivated.
5. Click **UDDI** → **UDDI Nodes** → *UDDI_node_id* to display the properties page for the UDDI registry node. Set **Prefix for generated discoveryURLs** to a valid URL for your configuration. This property specifies the URL prefix that is applied to generated discovery URLs that are used by the HTTP GET service for UDDI Version 2.

What to do next

Because you have chosen to use a default UDDI node, the node will be initialized when the UDDI application is started for the first time. Follow the instructions in “Using the UDDI registry Installation Verification Program (IVP)” on page 813 to verify that you have successfully set up the UDDI node.

Creating a DB2 distributed database for the UDDI registry

Perform this task if you want to use DB2 on the Windows, Linux or UNIX operating systems, as the database store for your UDDI registry data.

Before you begin

The following steps use a number of variables. Before you start, decide appropriate values to use for these variables. The variables, and suggested values, are:

<DataBaseName>
The name of the UDDI registry database. A recommended value is UDDI30, and this name is assumed throughout the UDDI information. If you use another name, substitute that name when UDDI30 is used in the information center.

<DB2UserID>
A DB2 userid with administrative privileges.

<DB2Password>
The password for the DB2 userid.

<BufferPoolName>
The name of a buffer pool to be used by the UDDI registry database. A suggested name is uddibp, but any name can be used, because the buffer pool is created as part of this task.

<TableSpaceName>
The name of a table space. A suggested value is uddits, but any name can be used.

<TempTableSpaceName>
The name of a temporary table space. A suggested value is udditstemp, but any name can be used, because the temporary table space is created as part of this task.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

1. Change directory to *app_server_root*/UDDIReg/databaseScripts.
2. Start the DB2 Command Line Processor by entering `db2` at the command prompt.
3. Run the following command to setup the DB2 environment variables:
   ```
   set DB2CODEPAGE=1208
   ```
4. Create the DB2 database by entering the following command:
   ```
   create database <DataBaseName> using codeset UTF-8 territory en
   ```
   where <DataBaseName> is the name of the database being created.
5. Configure the DB2 database by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `update db cfg for <DataBaseName> using applheapsz 2048`
   c. `update db cfg for <DataBaseName> using logfilsiz 8192`

d. `connect reset`
e. `terminate`

6. Create additional database structures by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `create bufferpool <BufferPoolName> size 250 pagesize 32K`
   c. `connect reset`
   d. `terminate`
   e. `force application all`
   f. `terminate`
   g. `stop`
   h. `start`
7. Create further database structures by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `create regular tablespace uddits pagesize 32K managed by system using ('<TableSpaceName>') extentsize 64 prefetchsize 32 bufferpool <BufferPoolName>`
   c. `create system temporary tablespace <TempTableSpacename> pagesize 32K managed by system using ('<TempTableSpacename>') extentsize 32 overhead 14.06 prefetchsize 32 transferrate 0.33 bufferpool <BufferPoolName>`
8. Exit the DB2 Command Line Processor and enter the following commands exactly as shown, noting that one step uses -vf rather than -tvf (on Windows platforms, run the commands from the db2cmd window). These commands define the database structures needed to store the UDDI data:
   a. `db2 -tvf uddi30crt_10_prereq_db2.sql`
   b. `db2 -tvf uddi30crt_20_tables_generic.sql`
   c. `db2 -tvf uddi30crt_25_tables_db2udb.sql`
   d. `db2 -tvf uddi30crt_30_constraints_generic.sql`
   e. `db2 -tvf uddi30crt_35_constraints_db2udb.sql`
   f. `db2 -tvf uddi30crt_40_views_generic.sql`
   g. `db2 -tvf uddi30crt_45_views_db2udb.sql`
   h. `db2 -vf uddi30crt_50_triggers_db2udb.sql`
   i. `db2 -tvf uddi30crt_60_insert_initial_static_data.sql`
9. [Optional] Enter the following command if you want the database to be used as a default UDDI node:
   `db2 -tvf uddi30crt_70_insert_default_database_indicator.sql`

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating a DB2 for z/OS database for the UDDI registry

Perform this task if you want to use DB2 for z/OS as the database store for your UDDI registry data.

Before you begin

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

There are some known restrictions with DB2 for zSeries® Version 7:

- Publish and inquiry strings are limited to 255 characters. For more information, see UDDI registry Application Programming Interface.
- When a UDDI inquiry request uses a discoveryURL that contains complex Unicode characters, the request might fail to return expected entities. Avoid using Unicode characters in discoveryURL elements if you are using this version of DB2.

1. Copy the createddl.sh script that is supplied in *app_server_root*/UDDIReg/rexx to a temporary directory of your choice.
2. Using the UNIX System Services (USS) command prompt, edit the copy of the createddl.sh script, as follows:
   a. Search for the text 'Define some constants'.
   b. If WebSphere Application Server is not installed in the default location, update the root_dir constant to reflect this. The UDDIReg directory must remain at the end of the path.
   c. If you do not want to use the default temporary directory, update the temp_dir constant to specify the temporary directory that you require.
3. Using the USS command prompt, run the copy of the createddl.sh script by entering the following command:

   ```
   createddl.sh database_name tablespace_name hlq
   ```

   where the parameters are as follows:

   database_name
   The name that is used when defining the required DB2 tables and other components. The default is UDDI30.

   tablespace_name
   The tablespace in which the database's tables are defined. The default is UDDI30TS.

   hlq The high-level qualifier under which the SQL and JCL partitioned datasets (PDS) are created. The default is IBMUSER.

   The script generates the *hlq*.UDDI.SQL and *hlq*.UDDI.JCL partitioned data sets, which contain members that are required for subsequent steps. If the script is run successfully using the default parameters, the result is the following output:

```
database.tablespace = UDDI30.UDDI30TS
 HLQ = IBMUSER
 (  14) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_10_prereq_db2.sql
 ( 436) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_20_tables_generic.sql
 ( 136) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_25_tables_db2udb.sql
 ( 452) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_30_constraints_generic.sql
 (  14) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_35_constraints_db2udb.sql
 ( 559) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_40_views_generic.sql
 (  94) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_45_views_db2udb.sql
 ( 329) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_50_triggers_db2udb.sql
 (  16) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_60_insert_initial_static_
        data.sql
 (  39) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_70_insert_default_database_
        indicator.sql
 Conversion complete
 /tmp/udditmp/makedb71.jcl     ===> IBMUSER.UDDI.JCL(MAKEDB71)
 /tmp/udditmp/makedb81.jcl     ===> IBMUSER.UDDI.JCL(MAKEDB81)
 /tmp/udditmp/table.sql        ===> IBMUSER.UDDI.SQL(TABLE)
 /tmp/udditmp/table7.sql       ===> IBMUSER.UDDI.SQL(TABLE7)
 /tmp/udditmp/index.sql        ===> IBMUSER.UDDI.SQL(INDEX)
 /tmp/udditmp/view.sql         ===> IBMUSER.UDDI.SQL(VIEW)
 /tmp/udditmp/trigger.sql      ===> IBMUSER.UDDI.SQL(TRIGGER)
 /tmp/udditmp/alter.sql        ===> IBMUSER.UDDI.SQL(ALTER)
 /tmp/udditmp/initial.sql      ===> IBMUSER.UDDI.SQL(INITIAL)
 /tmp/udditmp/insert.sql       ===> IBMUSER.UDDI.SQL(INSERT)
```

4. There are two sample jobs in the JCL library for creating the DB2 database; one for DB2 version 7 and one for DB2 version 8. The JCL for these jobs can be found in members MAKEDB71 and MAKEDB81

respectively, in the *hlq*.UDDI.JCL PDS. These JCL scripts are templates; modify the template in the appropriate MAKEDB member according to your DB2 setup and whether you want a default or a customized UDDI node:

- Add or modify the JOB accounting information, if required.
- If you used a different high level qualifier from the default when running the script in step one, ensure that all occurrences of IBMUSER are changed to the qualifier that you specified.
- If you do not want your database to be used as a default UDDI node, comment out the line of the job which specifies the INSERT member of the SQL PDS; this should be the last line in the job.
- Ensure that all occurrences of the LIB parameter correctly reflect the directory into which you installed DB2.

5. Use TSO to submit the job that you modified in the previous step. The job will create the DB2 database.

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating an Apache Derby database for the UDDI registry

Perform this task to use Apache Derby (embedded or network) as the database store (either local or remote) for your UDDI registry.

Before you begin

The following steps use a number of variables. Before you start, decide appropriate values to use for these variables. The variables, and suggested values, are:

arg1 The path of the SQL files. On a standard installation, this is *app_server_root*/UDDIReg/databasescripts

arg2 The path to the location where you want to install the Apache Derby database.

For example, *app_server_root*/profiles/*profile_name*/databases/com.ibm.uddi

arg3 The name of the Apache Derby database. A recommended value is UDDI30, and this name is assumed throughout the UDDI information. If you use another name, substitute that name when UDDI30 is used in the information center.

arg4 An optional argument, which must either be the string 'DEFAULT', or be omitted. Specify DEFAULT if you want the database to be used as a default UDDI node. This argument is case sensitive.

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

1. Run the following Java -jar command from the *app_server_root*/UDDIReg/databaseScripts directory, to create a UDDI Apache Derby database using UDDIDerbyCreate.jar.

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib:app_server_root/java/jre/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   If the Apache Derby database already exists, you are asked if you want to recreate it. If you choose to recreate the database, your existing database is deleted and a new one is created in its place. If you choose not to recreate the database, the command exits and a new database is not created.

Note: If the application server has already accessed the existing Apache Derby database, the uddiDeploy.jacl script cannot recreate the database. Use the uddiRemove.jacl script to remove the database, as described in Removing a UDDI registry node, restart the server, and run the uddiDeploy.jacl script again.

2. Ensure that the database has the correct permissions to allow WebSphere Application Server to access it, by running the following command:

```
chmod -R 777 arg2/arg3
```

where *arg2* and *arg3* are the path and name of the Apache Derby database, as described above.

3. If you are using a remote database (which requires network Apache Derby), or you want to use network Apache Derby for other reasons, for example, if you want to use Apache Derby with a cluster, configure the Apache Derby Network Server framework. For details, see the section about managing the Derby Network Server in the Derby Server and Administration Guide.

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating a data source for the UDDI registry

You create a data source so that the UDDI registry can use it to access the UDDI database.

Before you begin

You must have already created the database for the UDDI registry. These instructions assume that, if you are installing into a cluster, a single database will be used by all members of the cluster.

About this task

Perform this task as part of setting up and deploying a new UDDI registry. The data source is used by the UDDI registry to access the UDDI database.

1. Create a J2C Authentication Data Entry. This is not required for embedded Apache Derby, but is required for network Apache Derby.
   a. Click **Security** → **Global security** → **[Authentication] Java Authentication and Authorization Service** → **J2C authentication data**.
   b. Click **New** to create a new J2C authentication data entry.
   c. Enter the following details:

      Alias A suitable short name, for example UDDIAlias.

      Userid

      The database user ID (for example db2admin for DB2), which is used to read and write to the UDDI registry database. For network Apache Derby, the user ID can be any value.

      Password

      The password associated with the user ID specified earlier. For network Apache Derby, the password can be any value.

      Description

      A description of the user ID.

      Click **Apply**, then save the changes to the master configuration.

2. Create a JDBC Provider, if a suitable one does not already exist, using the following table to determine the provider type and implementation type for your chosen database:

Database	Provider type	Implementation type
DB2	DB2 Universal JDBC Driver Provider	Connection Pool data source

Database	Provider type	Implementation type
Embedded Apache Derby	Derby JDBC Driver	Connection Pool data source
Network Apache Derby	Derby Network Server JDBC Driver provider	Connection Pool data source
Microsoft SQL Server	DataDirect Connect JDBC Driver Microsoft SQL Server JDBC Driver	Connection Pool data source

Note: If you are setting up a UDDI node in a cluster, select cluster as the scope of the JDBC provider. For details on how to create a JDBC provider, see Creating and configuring a JDBC provider using the administrative console.

3. Use the following steps to create the data source for the UDDI registry:
   a. Click **Resources** → **JDBC** → **JDBC Providers**.
   b. Select the scope of the JDBC provider that you selected or created earlier, that is, the level at which the JDBC provider is defined. For example, for a JDBC provider that is defined at the level of server1, select the following:

```
Node=Node01, Server=server1
```

      All the JDBC providers that are defined at the selected scope are displayed.
   c. Select the JDBC provider that you created earlier.
   d. Under **Additional Properties**, select **Data sources**. Do not select the **Data sources (WebSphere Application Server V4)** option.
   e. Click **New** to create a new data source.
   f. In the **Create a data source** wizard, enter the following data:

      Name A suitable name, for example UDDI Datasource.

      JNDI name
      Set this value to **datasources/uddids**. This is a mandatory field.

      You must not have any other data sources that use this JNDI name. If you have another data source that uses this JNDI name, you must either remove it or change its JNDI name. For example, if you created a default UDDI node previously using an Apache Derby database, before you continue, use the uddiRemove.jacl script with the default option to remove the data source and the UDDI application instance.

      Component-managed authentication alias
      - For DB2 or network Apache Derby, select the alias that you created in step 2. It is prefixed by the node name, for example MyNode/UDDIAlias.
      - for embedded Apache Derby leave this set to (none).

   g. Click **Next**.
   h. On the database-specific properties page of the wizard, enter the following data:
      - for DB2:

        Database name
        this is the local LOCATION value. To find this value, enter the following operator command at the console, or ask your DB2 administrator for the information:

```
-DIS DDF
```

        This value is case sensitive.

        Note: If you are using a remote database, the database name is the alias that you created to reference the database. See Creating a DB2 distributed database.

Driver type
Set this value to 4.

Server name
Set this value to the IP address of the machine that is hosting the database. Use the `-DIS DDF` operator command to find this information, or ask your DB2 administrator for the information.

Port number
Set this value to the port that the DB2 database is listening on. Use the `-DIS DDF` operator command to find this information (or ask your DB2 administrator for the information).

- For Apache Derby (embedded or network) - **Database name** - for example:

app_server_root/profiles/*profile_name*/databases/com.ibm.uddi/UDDI30

For network Apache Derby, also make sure that the **Server name** and **Port number** values match the network server.

Leave all other fields unchanged.

Use this Data Source in container-managed persistence (CMP)
Ensure that the check box is cleared.

i. Click **Next**, then check the summary and click **Finish**.

j. Click the data source to display its properties, and add the following information:

Description
A description of the data source.

Category
Set this value to uddi.

Data store helper class name
This value is provided automatically:

Database	Data store helper class name
DB2	com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper
Embedded Apache Derby	com.ibm.websphere.rsadapter.DerbyDataStoreHelper
Network Apache Derby	com.ibm.websphere.rsadapter.DerbyNetworkServerDataStoreHelper

Mapping-configuration alias
Set this option to DefaultPrincipalMapping.

k. Click **Apply** and save the changes to the master configuration.

4. Test the connection to your UDDI database by selecting the check box next to the data source and clicking **Test connection**. A message similar to “Test Connection for datasource UDDI Datasource on server server1 at node Node01 was successful” is displayed. If a different message is displayed, use the information in that message to investigate and resolve the problem.

What to do next

Continue with setting up and deploying your UDDI registry node.

Deploying the UDDI registry application

You deploy a UDDI registry application as part of setting up a UDDI node. You can either use the supplied script, or use the administrative console.

Before you begin

Before you deploy a UDDI registry application, you must create the database and data source for the UDDI registry.

About this task

Use this task as part of “Setting up a default UDDI node” on page 794 or “Setting up a customized UDDI node” on page 803.

Run the uddiDeploy.jacl script as shown, from the *app_server_root*/bin directory.
This script deploys the UDDI registry to a server or cluster that you specify.

```
wsadmin.sh [-conntype none] [-profileName profile_name] -f uddiDeploy.jacl
              {node_name server_name | cluster_name}
```

where:

- ’-conntype none’ is optional, and is needed only if the application server is not running.
- ’-profileName *profile_name*’ is the deployment manager profile. If you do not specify a profile, the default profile is used.
- *node_name* is the name of the WebSphere Application Server node on which the target server runs. The node name is case sensitive.
- *server_name* is the name of the target server on which you wish to deploy the UDDI registry, for example, server1. The server name is case sensitive.
- *cluster_name* is the name of the target cluster into which you wish to deploy the UDDI registry. The cluster name is case sensitive.

For example, to deploy UDDI on node ’MyNode’ and server ’server1’ (assuming that server1 is already started):

```
wsadmin.sh -f uddiDeploy.jacl MyNode server1
```

To deploy UDDI into cluster ’MyCluster’ :

```
wsadmin.sh -f uddiDeploy.jacl MyCluster
```

You can also deploy the UDDI application (the uddi.ear file) using the administrative console, in the normal way. However, if you use the administrative console, some steps that the uddiDeploy.jacl script performs automatically do not occur. If you use the administrative console to install the UDDI application, you must perform some actions manually. To do this, use the following steps:

1. Install the application.
2. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***uddi_application*** → **[Detail Properties] Class loading and update detection**.
3. Ensure that **Class loader order** is set to **Classes loaded with application class loader first**.
4. Ensure that **WAR class loader policy** is set to **Single class loader for application**.

Results

The UDDI application is deployed. If you see the following error message, check that you ran the uddiDeploy.jacl script using the deployment manager profile.

```
WASX7017E: Exception received while running file "uddiDeploy.jacl"; exception in
formation: com.ibm.ws.scripting.ScriptingException: WASX7070E: The configuration
 service is not available.
```

What to do next

Continue setting up the UDDI node.

Setting up a customized UDDI node

You set up a UDDI node with your own property values so that it is suitable for production configurations.

About this task

Use this task to set up a UDDI node with property values that you choose. You cannot change the mandatory node properties, such as node ID, after the initialization of the node. Such a node is suitable for production purposes.

Note: If you deploy the UDDI registry application into a cluster, and you want to use Apache Derby for your database, you must use the network version of Apache Derby. Embedded Apache Derby is not supported for cluster configurations.

1. Review the information in "Database considerations for production use of the UDDI registry" on page 793 to decide which database system to use, then create a database schema to hold the UDDI registry by completing one of the following tasks. Do not use the default node options where specified.
   - "Creating a DB2 distributed database for the UDDI registry" on page 795
   - "Creating a DB2 for z/OS database for the UDDI registry" on page 796
   - "Creating an Apache Derby database for the UDDI registry" on page 798

   Note: If you are creating the UDDI node in a cluster, it is assumed that a single database is used for all members of the cluster.
2. Set up a data source for the UDDI registry application to use to access the database, as described in "Creating a data source for the UDDI registry" on page 799.
3. Deploy the UDDI registry application, as described in "Deploying the UDDI registry application" on page 801.
4. Click **Applications** → **Application Types** → **WebSphere enterprise applications** to display the installed applications. Start the UDDI registry application by selecting the check box next to it and clicking **Start**. Alternatively, if the application server is not already running, start the application server. This action automatically starts the UDDI registry application. The UDDI node is now active.

   Note: Restarting the UDDI application, or the application server, always reactivates the UDDI node, even if the node was previously deactivated.

What to do next

Because you chose a user-customized UDDI node, you need to set the properties for the UDDI node using UDDI administration, and initialize the node, before it is ready to accept UDDI requests. See "Initializing the UDDI registry node" on page 811 for details.

Creating a DB2 distributed database for the UDDI registry

Perform this task if you want to use DB2 on the Windows, Linux or UNIX operating systems, as the database store for your UDDI registry data.

Before you begin

The following steps use a number of variables. Before you start, decide appropriate values to use for these variables. The variables, and suggested values, are:

<DataBaseName>
: The name of the UDDI registry database. A recommended value is UDDI30, and this name is assumed throughout the UDDI information. If you use another name, substitute that name when UDDI30 is used in the information center.

<DB2UserID>
: A DB2 userid with administrative privileges.

<DB2Password>
The password for the DB2 userid.

<BufferPoolName>
The name of a buffer pool to be used by the UDDI registry database. A suggested name is uddibp, but any name can be used, because the buffer pool is created as part of this task.

<TableSpaceName>
The name of a table space. A suggested value is uddits, but any name can be used.

<TempTableSpaceName>
The name of a temporary table space. A suggested value is udditstemp, but any name can be used, because the temporary table space is created as part of this task.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

1. Change directory to *app_server_root*/UDDIReg/databaseScripts.
2. Start the DB2 Command Line Processor by entering `db2` at the command prompt.
3. Run the following command to setup the DB2 environment variables:
   ```
   set DB2CODEPAGE=1208
   ```
4. Create the DB2 database by entering the following command:
   ```
   create database <DataBaseName> using codeset UTF-8 territory en
   ```
   where <DataBaseName> is the name of the database being created.
5. Configure the DB2 database by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `update db cfg for <DataBaseName> using applheapsz 2048`
   c. `update db cfg for <DataBaseName> using logfilsiz 8192`
   d. `connect reset`
   e. `terminate`
6. Create additional database structures by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `create bufferpool <BufferPoolName> size 250 pagesize 32K`
   c. `connect reset`
   d. `terminate`
   e. `force application all`
   f. `terminate`
   g. `stop`
   h. `start`
7. Create further database structures by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `create regular tablespace uddits pagesize 32K managed by system using ('<TableSpaceName>') extentsize 64 prefetchsize 32 bufferpool <BufferPoolName>`
   c. `create system temporary tablespace <TempTableSpacename> pagesize 32K managed by system using ('<TempTableSpacename>') extentsize 32 overhead 14.06 prefetchsize 32 transferrate 0.33 bufferpool <BufferPoolName>`

8. Exit the DB2 Command Line Processor and enter the following commands exactly as shown, noting that one step uses -vf rather than -tvf (on Windows platforms, run the commands from the db2cmd window). These commands define the database structures needed to store the UDDI data:
   a. `db2 -tvf uddi30crt_10_prereq_db2.sql`
   b. `db2 -tvf uddi30crt_20_tables_generic.sql`
   c. `db2 -tvf uddi30crt_25_tables_db2udb.sql`
   d. `db2 -tvf uddi30crt_30_constraints_generic.sql`
   e. `db2 -tvf uddi30crt_35_constraints_db2udb.sql`
   f. `db2 -tvf uddi30crt_40_views_generic.sql`
   g. `db2 -tvf uddi30crt_45_views_db2udb.sql`
   h. `db2 -vf uddi30crt_50_triggers_db2udb.sql`
   i. `db2 -tvf uddi30crt_60_insert_initial_static_data.sql`
9. [Optional] Enter the following command if you want the database to be used as a default UDDI node:
   `db2 -tvf uddi30crt_70_insert_default_database_indicator.sql`

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating a DB2 for z/OS database for the UDDI registry

Perform this task if you want to use DB2 for z/OS as the database store for your UDDI registry data.

Before you begin

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

There are some known restrictions with DB2 for zSeries Version 7:

- Publish and inquiry strings are limited to 255 characters. For more information, see UDDI registry Application Programming Interface.
- When a UDDI inquiry request uses a discoveryURL that contains complex Unicode characters, the request might fail to return expected entities. Avoid using Unicode characters in discoveryURL elements if you are using this version of DB2.

1. Copy the createddl.sh script that is supplied in *app_server_root*/UDDIReg/rexx to a temporary directory of your choice.
2. Using the UNIX System Services (USS) command prompt, edit the copy of the createddl.sh script, as follows:
   a. Search for the text 'Define some constants'.
   b. If WebSphere Application Server is not installed in the default location, update the root_dir constant to reflect this. The UDDIReg directory must remain at the end of the path.
   c. If you do not want to use the default temporary directory, update the temp_dir constant to specify the temporary directory that you require.
3. Using the USS command prompt, run the copy of the createddl.sh script by entering the following command:
   `createddl.sh database_name tablespace_name hlq`

where the parameters are as follows:

database_name
The name that is used when defining the required DB2 tables and other components. The default is UDDI30.

tablespace_name
The tablespace in which the database's tables are defined. The default is UDDI30TS.

hlq The high-level qualifier under which the SQL and JCL partitioned datasets (PDS) are created. The default is IBMUSER.

The script generates the *hlq*.UDDI.SQL and *hlq*.UDDI.JCL partitioned data sets, which contain members that are required for subsequent steps. If the script is run successfully using the default parameters, the result is the following output:

```
database.tablespace = UDDI30.UDDI30TS
 HLQ = IBMUSER
 (  14) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_10_prereq_db2.sql
 ( 436) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_20_tables_generic.sql
 ( 136) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_25_tables_db2udb.sql
 ( 452) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_30_constraints_generic.sql
 (  14) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_35_constraints_db2udb.sql
 ( 559) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_40_views_generic.sql
 (  94) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_45_views_db2udb.sql
 ( 329) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_50_triggers_db2udb.sql
 (  16) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_60_insert_initial_static_
        data.sql
 (  39) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_70_insert_default_database_
        indicator.sql
 Conversion complete
 /tmp/udditmp/makedb71.jcl     ===> IBMUSER.UDDI.JCL(MAKEDB71)
 /tmp/udditmp/makedb81.jcl     ===> IBMUSER.UDDI.JCL(MAKEDB81)
 /tmp/udditmp/table.sql        ===> IBMUSER.UDDI.SQL(TABLE)
 /tmp/udditmp/table7.sql       ===> IBMUSER.UDDI.SQL(TABLE7)
 /tmp/udditmp/index.sql        ===> IBMUSER.UDDI.SQL(INDEX)
 /tmp/udditmp/view.sql         ===> IBMUSER.UDDI.SQL(VIEW)
 /tmp/udditmp/trigger.sql      ===> IBMUSER.UDDI.SQL(TRIGGER)
 /tmp/udditmp/alter.sql        ===> IBMUSER.UDDI.SQL(ALTER)
 /tmp/udditmp/initial.sql      ===> IBMUSER.UDDI.SQL(INITIAL)
 /tmp/udditmp/insert.sql       ===> IBMUSER.UDDI.SQL(INSERT)
```

4. There are two sample jobs in the JCL library for creating the DB2 database; one for DB2 version 7 and one for DB2 version 8. The JCL for these jobs can be found in members MAKEDB71 and MAKEDB81 respectively, in the *hlq*.UDDI.JCL PDS. These JCL scripts are templates; modify the template in the appropriate MAKEDB member according to your DB2 setup and whether you want a default or a customized UDDI node:
   - Add or modify the JOB accounting information, if required.
   - If you used a different high level qualifier from the default when running the script in step one, ensure that all occurrences of IBMUSER are changed to the qualifier that you specified.
   - If you do not want your database to be used as a default UDDI node, comment out the line of the job which specifies the INSERT member of the SQL PDS; this should be the last line in the job.
   - Ensure that all occurrences of the LIB parameter correctly reflect the directory into which you installed DB2.
5. Use TSO to submit the job that you modified in the previous step. The job will create the DB2 database.

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating an Apache Derby database for the UDDI registry

Perform this task to use Apache Derby (embedded or network) as the database store (either local or remote) for your UDDI registry.

Before you begin

The following steps use a number of variables. Before you start, decide appropriate values to use for these variables. The variables, and suggested values, are:

arg1 The path of the SQL files. On a standard installation, this is *app_server_root*/UDDIReg/databasescripts

arg2 The path to the location where you want to install the Apache Derby database.

For example, *app_server_root*/profiles/*profile_name*/databases/com.ibm.uddi

arg3 The name of the Apache Derby database. A recommended value is UDDI30, and this name is assumed throughout the UDDI information. If you use another name, substitute that name when UDDI30 is used in the information center.

arg4 An optional argument, which must either be the string 'DEFAULT', or be omitted. Specify DEFAULT if you want the database to be used as a default UDDI node. This argument is case sensitive.

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

1. Run the following Java -jar command from the *app_server_root*/UDDIReg/databaseScripts directory, to create a UDDI Apache Derby database using UDDIDerbyCreate.jar.

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib:app_server_root/java/jre/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   If the Apache Derby database already exists, you are asked if you want to recreate it. If you choose to recreate the database, your existing database is deleted and a new one is created in its place. If you choose not to recreate the database, the command exits and a new database is not created.

   Note: If the application server has already accessed the existing Apache Derby database, the uddiDeploy.jacl script cannot recreate the database. Use the uddiRemove.jacl script to remove the database, as described in Removing a UDDI registry node, restart the server, and run the uddiDeploy.jacl script again.
2. Ensure that the database has the correct permissions to allow WebSphere Application Server to access it, by running the following command:

   ```
   chmod -R 777 arg2/arg3
   ```

   where *arg2* and *arg3* are the path and name of the Apache Derby database, as described above.
3. If you are using a remote database (which requires network Apache Derby), or you want to use network Apache Derby for other reasons, for example, if you want to use Apache Derby with a cluster, configure the Apache Derby Network Server framework. For details, see the section about managing the Derby Network Server in the Derby Server and Administration Guide.

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating a data source for the UDDI registry

You create a data source so that the UDDI registry can use it to access the UDDI database.

Before you begin

You must have already created the database for the UDDI registry. These instructions assume that, if you are installing into a cluster, a single database will be used by all members of the cluster.

About this task

Perform this task as part of setting up and deploying a new UDDI registry. The data source is used by the UDDI registry to access the UDDI database.

1. Create a J2C Authentication Data Entry. This is not required for embedded Apache Derby, but is required for network Apache Derby.
   a. Click **Security** → **Global security** → **[Authentication] Java Authentication and Authorization Service** → **J2C authentication data**.
   b. Click **New** to create a new J2C authentication data entry.
   c. Enter the following details:

      Alias A suitable short name, for example UDDIAlias.

      Userid

      The database user ID (for example db2admin for DB2), which is used to read and write to the UDDI registry database. For network Apache Derby, the user ID can be any value.

      Password
      The password associated with the user ID specified earlier. For network Apache Derby, the password can be any value.

      Description
      A description of the user ID.

      Click **Apply**, then save the changes to the master configuration.
2. Create a JDBC Provider, if a suitable one does not already exist, using the following table to determine the provider type and implementation type for your chosen database:

Database	Provider type	Implementation type
DB2	DB2 Universal JDBC Driver Provider	Connection Pool data source
Embedded Apache Derby	Derby JDBC Driver	Connection Pool data source
Network Apache Derby	Derby Network Server JDBC Driver provider	Connection Pool data source
Microsoft SQL Server	DataDirect Connect JDBC Driver Microsoft SQL Server JDBC Driver	Connection Pool data source

   Note: If you are setting up a UDDI node in a cluster, select cluster as the scope of the JDBC provider. For details on how to create a JDBC provider, see Creating and configuring a JDBC provider using the administrative console.
3. Use the following steps to create the data source for the UDDI registry:
   a. Click **Resources** → **JDBC** → **JDBC Providers**.
   b. Select the scope of the JDBC provider that you selected or created earlier, that is, the level at which the JDBC provider is defined. For example, for a JDBC provider that is defined at the level of server1, select the following:

```
Node=Node01, Server=server1
```

      All the JDBC providers that are defined at the selected scope are displayed.
   c. Select the JDBC provider that you created earlier.

d. Under **Additional Properties**, select **Data sources**. Do not select the **Data sources (WebSphere Application Server V4)** option.

e. Click **New** to create a new data source.

f. In the **Create a data source** wizard, enter the following data:

Name A suitable name, for example UDDI Datasource.

JNDI name

Set this value to **datasources/uddids**. This is a mandatory field.

You must not have any other data sources that use this JNDI name. If you have another data source that uses this JNDI name, you must either remove it or change its JNDI name. For example, if you created a default UDDI node previously using an Apache Derby database, before you continue, use the uddiRemove.jacl script with the default option to remove the data source and the UDDI application instance.

Component-managed authentication alias

- For DB2 or network Apache Derby, select the alias that you created in step 2. It is prefixed by the node name, for example MyNode/UDDIAlias.
- for embedded Apache Derby leave this set to (none).

g. Click **Next**.

h. On the database-specific properties page of the wizard, enter the following data:

- for DB2:

Database name

this is the local LOCATION value. To find this value, enter the following operator command at the console, or ask your DB2 administrator for the information:

```
-DIS DDF
```

This value is case sensitive.

Note: If you are using a remote database, the database name is the alias that you created to reference the database. See Creating a DB2 distributed database.

Driver type

Set this value to 4.

Server name

Set this value to the IP address of the machine that is hosting the database. Use the `-DIS DDF` operator command to find this information, or ask your DB2 administrator for the information.

Port number

Set this value to the port that the DB2 database is listening on. Use the `-DIS DDF` operator command to find this information (or ask your DB2 administrator for the information).

- For Apache Derby (embedded or network) - **Database name** - for example:

```
app_server_root/profiles/profile_name/databases/com.ibm.uddi/UDDI30
```

For network Apache Derby, also make sure that the **Server name** and **Port number** values match the network server.

Leave all other fields unchanged.

Use this Data Source in container-managed persistence (CMP)

Ensure that the check box is cleared.

i. Click **Next**, then check the summary and click **Finish**.

j. Click the data source to display its properties, and add the following information:

Description
A description of the data source.

Category
Set this value to uddi.

Data store helper class name
This value is provided automatically:

Database	Data store helper class name
DB2	com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper
Embedded Apache Derby	com.ibm.websphere.rsadapter.DerbyDataStoreHelper
Network Apache Derby	com.ibm.websphere.rsadapter.DerbyNetworkServerDataStoreHelper

Mapping-configuration alias
Set this option to DefaultPrincipalMapping.

k. Click **Apply** and save the changes to the master configuration.

4. Test the connection to your UDDI database by selecting the check box next to the data source and clicking **Test connection**. A message similar to "Test Connection for datasource UDDI Datasource on server server1 at node Node01 was successful" is displayed. If a different message is displayed, use the information in that message to investigate and resolve the problem.

What to do next

Continue with setting up and deploying your UDDI registry node.

Deploying the UDDI registry application

You deploy a UDDI registry application as part of setting up a UDDI node. You can either use the supplied script, or use the administrative console.

Before you begin

Before you deploy a UDDI registry application, you must create the database and data source for the UDDI registry.

About this task

Use this task as part of "Setting up a default UDDI node" on page 794 or "Setting up a customized UDDI node" on page 803.

Run the uddiDeploy.jacl script as shown, from the *app_server_root*/bin directory.
This script deploys the UDDI registry to a server or cluster that you specify.

```
wsadmin.sh [-conntype none] [-profileName profile_name] -f uddiDeploy.jacl
           {node_name server_name | cluster_name}
```

where:

- '-conntype none' is optional, and is needed only if the application server is not running.
- '-profileName *profile_name*' is the deployment manager profile. If you do not specify a profile, the default profile is used.
- *node_name* is the name of the WebSphere Application Server node on which the target server runs. The node name is case sensitive.
- *server_name* is the name of the target server on which you wish to deploy the UDDI registry, for example, server1. The server name is case sensitive.

- *cluster_name* is the name of the target cluster into which you wish to deploy the UDDI registry. The cluster name is case sensitive.

For example, to deploy UDDI on node 'MyNode' and server 'server1' (assuming that server1 is already started):

```
wsadmin.sh -f uddiDeploy.jacl MyNode server1
```

To deploy UDDI into cluster 'MyCluster' :

```
wsadmin.sh -f uddiDeploy.jacl MyCluster
```

You can also deploy the UDDI application (the uddi.ear file) using the administrative console, in the normal way. However, if you use the administrative console, some steps that the uddiDeploy.jacl script performs automatically do not occur. If you use the administrative console to install the UDDI application, you must perform some actions manually. To do this, use the following steps:

1. Install the application.
2. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***uddi_application*** → **[Detail Properties] Class loading and update detection**.
3. Ensure that **Class loader order** is set to **Classes loaded with application class loader first**.
4. Ensure that **WAR class loader policy** is set to **Single class loader for application**.

Results

The UDDI application is deployed. If you see the following error message, check that you ran the uddiDeploy.jacl script using the deployment manager profile.

```
WASX7017E: Exception received while running file "uddiDeploy.jacl"; exception in
formation: com.ibm.ws.scripting.ScriptingException: WASX7070E: The configuration
 service is not available.
```

What to do next

Continue setting up the UDDI node.

Initializing the UDDI registry node

Use this topic to initialize a UDDI registry node after set up or migration.

Before you begin

You must have already set up a UDDI registry node, either as a new node or to use for migrating a UDDI registry Version 2 node.

About this task

The UDDI registry node has various properties, some of which must be set before initializing the node. There are two categories of UDDI registry node properties:

- Mandatory node properties. These properties must be set before the UDDI node can be initialized. You may set these properties as many times as you wish before initialization. However, once the UDDI node has been initialized, these properties will become read only for the lifetime of that UDDI node. It is very important to set these properties correctly.
- All other properties. These properties may be set before, and after, initialization.

Configure these properties and initialize the node using the UDDI administrative console or JMX management interface.

1. Click **UDDI** → **UDDI Nodes** > *UDDI_node_id* to display the properties page for the UDDI registry node.

2. Set the mandatory node properties to suitable, and valid, values. These properties are indicated by the presence of a '*' next to the input field. The properties are listed below; more information on each property is given in the context help of the administrative console.

   UDDI node ID
   This must be a text string beginning with 'uddi:' that is unique to this UDDI node. The default value may be sufficient, but if you accept it you should ensure that it is unique.

   UDDI node description
   This is a text string describing the node.

   Root key generator
   This must be a text string beginning with 'uddi:' that is unique to this UDDI node. The default value may be sufficient but may contain text, such as 'keyspace_id', that you should modify to match your system. If you accept the default value, ensure that it is unique for this UDDI node.

   Prefix for generated discoveryURLs
   This should be a valid URL.

3. If you are migrating from Version 2 of the UDDI registry, use the table below to perform the following steps:
   - Set any properties from uddi.properties that **must** remain the same as Version 2.
   - Set any properties from uddi.properties that you would like to keep the same (such as dbMaxResultCount).

Version 2 UDDI property (set in uddi.property file)	Version 3 UDDI Property (set via Administrative Console or UDDI Administrative Interface)	Recommended Version 3 UDDI property setting
dbMaxResultCount	maximum inquiry response set size	You might want to retain the value from Version 2, but can safely change this (or use the default)
persister	no equivalent	Not applicable
defaultLanguage	default language code	You are recommended to retain the value from Version 2
operatorName	UDDI node ID	You must use a valid value for the UDDI node ID. This will be applied to your Version 2 data as it is migrated.
maxSearchKeys	maximum search keys	You might want to retain the value from Version 2, but can safely change this (or use the default)
getServletURLprefix	Prefix for generated discoveryURLs	You should enter a valid value for your configuration, which should therefore be the same as the value used for Version 2.
getServletName	no equivalent	Not applicable

4. Set any other properties, such as policy values, that you wish to change from the default settings (or these can be changed at a later time). For an explanation of policies and properties see UDDI node settings.
5. Click **Apply** to save your changes.

   Note: You cannot change mandatory node properties after initialization. If you do not save your changes before proceeding to the initialize step, you will have to delete and recreate the database.

6. After saving your changes, initialize the UDDI node by clicking **Initialize**, at the top of the pane.

   If you are migrating from Version 2 of the UDDI registry, the Version 2 data is migrated now. The initialization may take some time to complete; to track its progress, return to the node collection page

and click the refresh icon at the top of the **Status** column. Alternatively, open a second administrative console window, and use the refresh icon in the same manner. The UDDI node passes through the following states

a. Initialization pending.
b. Initialization in progress.
c. Migration in progress. (This state will only occur if you are migrating.)
d. Value set creation in progress.
e. Activated.

What to do next

If you have migrated the node from a previous version, return to Migrating to Version 3 of the UDDI registry to verify that the migration was successful. If you have created a new node, follow the instructions in "Using the UDDI registry Installation Verification Program (IVP)" to verify that you have successfully set up the UDDI node.

Using the UDDI registry Installation Verification Program (IVP)

Use the Installation Verification Program (IVP) to verify that you have successfully deployed a UDDI registry.

Before you begin

This topic describes a simple test that you can carry out as an Installation Verification Program (IVP) to verify that you have deployed a UDDI registry successfully. You should perform this task *after* you have followed the instructions in Setting up and Deploying a new UDDI registry.

1. Open a browser window and enter the URL that accesses the UDDI registry User Interface (see "Using the UDDI registry user interface" on page 833).
2. Under the **Quick Find** heading on the **Find** tab, click the **Business** radio button and enter % in the **Starting with** field.
3. Click **Find**. If you have deployed your UDDI registry successfully, the detail frame displays the business entity which represents this UDDI node. You can click on the business entity to see its detail.

What to do next

As a further installation verification test, you can publish and find more UDDI entities by using the UDDI registry User Interface, or you can compile and run one or more of the UDDI registry samples available through the UDDI registry link on the Samples for WebSphere Application Server page of the IBM developerWorks® WebSphere Web site.

Changing the UDDI registry application environment after deployment

You can change the environment of the UDDI registry application after you deploy it. This means you can evaluate a UDDI registry using one database, then put it into production using a different database, or you can move from a standalone application server to a network deployment cell.

About this task

After you deploy the UDDI registry application, you might want to change its environment. For example, you might perform initial evaluation of the UDDI registry using an Apache Derby database, and then want to put the UDDI registry into production using a DB2 database, or you might want to move from a standalone application server to a network deployment cell.

1. Optional: To incorporate a standalone application server into a network deployment cell, run the addNode command included with WebSphere Application Server. Use the includeapps parameter to ensure that the UDDI registry application, and any other applications on the server, are included in the move. See the addNode command for details.
2. Optional: To move from a default UDDI node to a customized UDDI node, delete the UDDI registry database and recreate it by completing one of the following tasks, ensuring that you do NOT use the default node options where specified:
   - "Creating a DB2 distributed database for the UDDI registry" on page 795
   - "Creating a DB2 for z/OS database for the UDDI registry" on page 796
   - "Creating an Apache Derby database for the UDDI registry" on page 798

   Note: Any data saved in the default node (policies, properties and user data) will be lost when you delete the database. If you do not want to delete the database, you can instead create an entirely new customized UDDI node in a separate application server. The default UDDI node will still exist for you to use for test purposes.
3. Optional: To change the database type for the UDDI registry, perform the following steps:
   a. Stop the UDDI registry application (click **Applications** → **Application Types** → **WebSphere enterprise applications**, select the relevant check box and click **Stop**).
   b. Either change the JNDI name of the existing datasource from datasources/uddids to another value, or delete the datasource. To display the datasource properties click **Resources** → **JDBC** → **JDBC providers >** *database_type* **JDBC Provider > [Additional Properties] Data sources >** *uddi_datasource*.
   c. Create the new database by referring to one of the following topics:
      - "Creating a DB2 distributed database for the UDDI registry" on page 795
      - "Creating a DB2 for z/OS database for the UDDI registry" on page 796
      - "Creating an Apache Derby database for the UDDI registry" on page 798
   d. To transfer your UDDI data, use the standard capabilities of the database products to export the data from the old database, and import it into the new one.
   e. Create the new datasource. See "Creating a data source for the UDDI registry" on page 799.
   f. Restart the UDDI registry application.
   g. Check that you can access your UDDI data, then delete the old database.

Creating a DB2 distributed database for the UDDI registry

Perform this task if you want to use DB2 on the Windows, Linux or UNIX operating systems, as the database store for your UDDI registry data.

Before you begin

The following steps use a number of variables. Before you start, decide appropriate values to use for these variables. The variables, and suggested values, are:

<DataBaseName>
The name of the UDDI registry database. A recommended value is UDDI30, and this name is assumed throughout the UDDI information. If you use another name, substitute that name when UDDI30 is used in the information center.

<DB2UserID>
A DB2 userid with administrative privileges.

<DB2Password>
The password for the DB2 userid.

<BufferPoolName>
The name of a buffer pool to be used by the UDDI registry database. A suggested name is uddibp, but any name can be used, because the buffer pool is created as part of this task.

<TableSpaceName>
The name of a table space. A suggested value is uddits, but any name can be used.

<TempTableSpaceName>
The name of a temporary table space. A suggested value is udditstemp, but any name can be used, because the temporary table space is created as part of this task.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

1. Change directory to *app_server_root*/UDDIReg/databaseScripts.
2. Start the DB2 Command Line Processor by entering `db2` at the command prompt.
3. Run the following command to setup the DB2 environment variables:
   ```
   set DB2CODEPAGE=1208
   ```
4. Create the DB2 database by entering the following command:
   ```
   create database <DataBaseName> using codeset UTF-8 territory en
   ```
   where <DataBaseName> is the name of the database being created.
5. Configure the DB2 database by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `update db cfg for <DataBaseName> using applheapsz 2048`
   c. `update db cfg for <DataBaseName> using logfilsiz 8192`
   d. `connect reset`
   e. `terminate`
6. Create additional database structures by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `create bufferpool <BufferPoolName> size 250 pagesize 32K`
   c. `connect reset`
   d. `terminate`
   e. `force application all`
   f. `terminate`
   g. `stop`
   h. `start`
7. Create further database structures by entering the following commands:
   a. `connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
   b. `create regular tablespace uddits pagesize 32K managed by system using ('<TableSpaceName>') extentsize 64 prefetchsize 32 bufferpool <BufferPoolName>`
   c. `create system temporary tablespace <TempTableSpacename> pagesize 32K managed by system using ('<TempTableSpacename>') extentsize 32 overhead 14.06 prefetchsize 32 transferrate 0.33 bufferpool <BufferPoolName>`
8. Exit the DB2 Command Line Processor and enter the following commands exactly as shown, noting that one step uses -vf rather than -tvf (on Windows platforms, run the commands from the db2cmd window). These commands define the database structures needed to store the UDDI data:

a. `db2 -tvf uddi30crt_10_prereq_db2.sql`
b. `db2 -tvf uddi30crt_20_tables_generic.sql`
c. `db2 -tvf uddi30crt_25_tables_db2udb.sql`
d. `db2 -tvf uddi30crt_30_constraints_generic.sql`
e. `db2 -tvf uddi30crt_35_constraints_db2udb.sql`
f. `db2 -tvf uddi30crt_40_views_generic.sql`
g. `db2 -tvf uddi30crt_45_views_db2udb.sql`
h. `db2 -vf uddi30crt_50_triggers_db2udb.sql`
i. `db2 -tvf uddi30crt_60_insert_initial_static_data.sql`

9. [Optional] Enter the following command if you want the database to be used as a default UDDI node:
   `db2 -tvf uddi30crt_70_insert_default_database_indicator.sql`

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating a DB2 for z/OS database for the UDDI registry

Perform this task if you want to use DB2 for z/OS as the database store for your UDDI registry data.

Before you begin

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

There are some known restrictions with DB2 for zSeries Version 7:

- Publish and inquiry strings are limited to 255 characters. For more information, see UDDI registry Application Programming Interface.
- When a UDDI inquiry request uses a discoveryURL that contains complex Unicode characters, the request might fail to return expected entities. Avoid using Unicode characters in discoveryURL elements if you are using this version of DB2.

1. Copy the createddl.sh script that is supplied in *app_server_root*/UDDIReg/rexx to a temporary directory of your choice.
2. Using the UNIX System Services (USS) command prompt, edit the copy of the createddl.sh script, as follows:
   a. Search for the text 'Define some constants'.
   b. If WebSphere Application Server is not installed in the default location, update the root_dir constant to reflect this. The UDDIReg directory must remain at the end of the path.
   c. If you do not want to use the default temporary directory, update the temp_dir constant to specify the temporary directory that you require.
3. Using the USS command prompt, run the copy of the createddl.sh script by entering the following command:
   `createddl.sh` *`database_name tablespace_name hlq`*

   where the parameters are as follows:

database_name
The name that is used when defining the required DB2 tables and other components. The default is UDDI30.

tablespace_name
The tablespace in which the database's tables are defined. The default is UDDI30TS.

hlq The high-level qualifier under which the SQL and JCL partitioned datasets (PDS) are created. The default is IBMUSER.

The script generates the *hlq*.UDDI.SQL and *hlq*.UDDI.JCL partitioned data sets, which contain members that are required for subsequent steps. If the script is run successfully using the default parameters, the result is the following output:

```
database.tablespace = UDDI30.UDDI30TS
 HLQ = IBMUSER
 (  14) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_10_prereq_db2.sql
 ( 436) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_20_tables_generic.sql
 ( 136) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_25_tables_db2udb.sql
 ( 452) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_30_constraints_generic.sql
 (  14) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_35_constraints_db2udb.sql
 ( 559) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_40_views_generic.sql
 (  94) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_45_views_db2udb.sql
 ( 329) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_50_triggers_db2udb.sql
 (  16) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_60_insert_initial_static_
        data.sql
 (  39) /WebSphere/V6R0M0/AppServer/UDDIReg/databaseScripts/uddi30crt_70_insert_default_database_
        indicator.sql
 Conversion complete
 /tmp/udditmp/makedb71.jcl     ===> IBMUSER.UDDI.JCL(MAKEDB71)
 /tmp/udditmp/makedb81.jcl     ===> IBMUSER.UDDI.JCL(MAKEDB81)
 /tmp/udditmp/table.sql        ===> IBMUSER.UDDI.SQL(TABLE)
 /tmp/udditmp/table7.sql       ===> IBMUSER.UDDI.SQL(TABLE7)
 /tmp/udditmp/index.sql        ===> IBMUSER.UDDI.SQL(INDEX)
 /tmp/udditmp/view.sql         ===> IBMUSER.UDDI.SQL(VIEW)
 /tmp/udditmp/trigger.sql      ===> IBMUSER.UDDI.SQL(TRIGGER)
 /tmp/udditmp/alter.sql        ===> IBMUSER.UDDI.SQL(ALTER)
 /tmp/udditmp/initial.sql      ===> IBMUSER.UDDI.SQL(INITIAL)
 /tmp/udditmp/insert.sql       ===> IBMUSER.UDDI.SQL(INSERT)
```

4. There are two sample jobs in the JCL library for creating the DB2 database; one for DB2 version 7 and one for DB2 version 8. The JCL for these jobs can be found in members MAKEDB71 and MAKEDB81 respectively, in the *hlq*.UDDI.JCL PDS. These JCL scripts are templates; modify the template in the appropriate MAKEDB member according to your DB2 setup and whether you want a default or a customized UDDI node:
   - Add or modify the JOB accounting information, if required.
   - If you used a different high level qualifier from the default when running the script in step one, ensure that all occurrences of IBMUSER are changed to the qualifier that you specified.
   - If you do not want your database to be used as a default UDDI node, comment out the line of the job which specifies the INSERT member of the SQL PDS; this should be the last line in the job.
   - Ensure that all occurrences of the LIB parameter correctly reflect the directory into which you installed DB2.
5. Use TSO to submit the job that you modified in the previous step. The job will create the DB2 database.

What to do next

Continue with setting up and deploying your UDDI registry node.

Creating an Apache Derby database for the UDDI registry

Perform this task to use Apache Derby (embedded or network) as the database store (either local or remote) for your UDDI registry.

Before you begin

The following steps use a number of variables. Before you start, decide appropriate values to use for these variables. The variables, and suggested values, are:

arg1 The path of the SQL files. On a standard installation, this is *app_server_root*/UDDIReg/databasescripts

arg2 The path to the location where you want to install the Apache Derby database.

For example, *app_server_root*/profiles/*profile_name*/databases/com.ibm.uddi

arg3 The name of the Apache Derby database. A recommended value is UDDI30, and this name is assumed throughout the UDDI information. If you use another name, substitute that name when UDDI30 is used in the information center.

arg4 An optional argument, which must either be the string 'DEFAULT', or be omitted. Specify DEFAULT if you want the database to be used as a default UDDI node. This argument is case sensitive.

If you want to create a remote database, refer first to the database product documentation about the relevant capabilities of the product.

About this task

You need to perform this task only once for each UDDI registry, as part of setting up and deploying a UDDI registry.

1. Run the following Java -jar command from the *app_server_root*/UDDIReg/databaseScripts directory, to create a UDDI Apache Derby database using UDDIDerbyCreate.jar.

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib:app_server_root/java/jre/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   If the Apache Derby database already exists, you are asked if you want to recreate it. If you choose to recreate the database, your existing database is deleted and a new one is created in its place. If you choose not to recreate the database, the command exits and a new database is not created.

   Note: If the application server has already accessed the existing Apache Derby database, the uddiDeploy.jacl script cannot recreate the database. Use the uddiRemove.jacl script to remove the database, as described in Removing a UDDI registry node, restart the server, and run the uddiDeploy.jacl script again.

2. Ensure that the database has the correct permissions to allow WebSphere Application Server to access it, by running the following command:

   ```
   chmod -R 777 arg2/arg3
   ```

   where *arg2* and *arg3* are the path and name of the Apache Derby database, as described above.

3. If you are using a remote database (which requires network Apache Derby), or you want to use network Apache Derby for other reasons, for example, if you want to use Apache Derby with a cluster, configure the Apache Derby Network Server framework. For details, see the section about managing the Derby Network Server in the Derby Server and Administration Guide.

What to do next

Continue with setting up and deploying your UDDI registry node.

UDDI registry client programming

The UDDI registry provides several application programming interfaces (APIs) which you can use to access the UDDI registry programmatically.

1. Learn about the standard aspects of the UDDI APIs using the following topics.
   - "UDDI registry Version 3 entity keys" explains UDDI entity keys, and the capability with UDDI Version 3 to save UDDI entities with publisher-assigned keys.
   - "Use of digital signatures with the UDDI registry" on page 821 explains the support for digital signing of UDDI entities, and for validation of signatures.
   - "UDDI registry Application Programming Interface" on page 822 contains a summary of the UDDI Version 3 APIs as defined in the UDDI Version 3 specification.
2. Access the APIs programmatically in one of several different ways. The recommended client API is the "UDDI Version 3 Client" on page 829, which allows access to the UDDI Version 3 APIs from Java client code. Other client APIs are provided for compatibility with previous versions of the UDDI registry:
   - UDDI4J provides Java class libraries for accessing UDDI Version 1 and Version 2 APIs. These class libraries are both deprecated in this release, and replaced by the UDDI Version 3 Client for Java. See "UDDI4J programming interface (Deprecated)" on page 832 for further details.
   - "UDDI EJB Interface (Deprecated)" on page 832 provides an EJB interface to the UDDI Version 2 APIs. The UDDI EJB interface is deprecated in this release.

   Although the recommended programmatic access to the UDDI APIs is through the UDDI Version 3 Client for Java, it is also valid to use the UDDI APIs directly using SOAP. This can be done by constructing a properly-formed UDDI message within the body of a SOAP request, and sending it using HTTP POST to the appropriate SOAP endpoint for the UDDI service (see "UDDI registry SOAP service end points" on page 830. The response will be returned within the body of the HTTP reply.

   Support is also provided for the use of HTTP GET to return XML representations of UDDI entities: see "HTTP GET services for UDDI registry data structures" on page 830 for details.

UDDI registry Version 3 entity keys

The UDDI Version 3 specification expands the space available for keys. Entity keys can now be any URI (Universal Resource Identifier) that follows the recommended UDDI scheme. Depending on registry policy, keys can be assigned, not only by the UDDI registry, but also by the publisher of the entity.

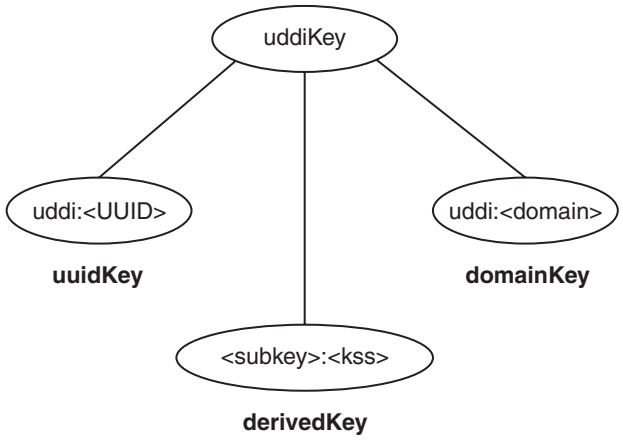
Entity keys are identifiers used to address entities within a UDDI registry. Each entity, for example businessEntity, businessService, bindingTemplate, or tModel, has a unique identifier that is generated or assigned when it is first published in the UDDI registry. Within a particular registry, a key must be unique. In the UDDI Version 1 and Version 2, the space is limited to a universal unique identifier (UUID) . In the UDDI Version 3 specification, entity keys can be any URI that follows the recommended UDDI scheme.

In the UDDI Version 3 specification, depending on registry policy, keys can be assigned, not only by the UDDI registry, but also by the publisher of the entity. These differences raise issues for maintaining key uniqueness and managing key space.

UDDI Scheme

The UDDI Version 3 registry implements the recommended UDDI scheme, as detailed in Section 4.4 of the UDDI Version 3 Specification. (http://uddi.org/pubs/uddi_v3.htm). This scheme defines the format of the keys, the valid characters, and the concept of key space.

In the UDDI Version 3 registry, a key is any URI and is limited to 255 characters. The following diagram shows the different types of keys in the UDDI key scheme:



All keys are composed of a set of tokens that are separated by ':'. The first token for all keys that follow the UDDI scheme is uddi. There are three types of keys:

1. The uuidKey keys contain two tokens, the mandatory uddi and a <UUID>. These keys ensure uniqueness through the UUID algorithm.
2. The domainKey keys also contain two tokens but the second token is a domain name. These keys are for creating additional mutually exclusive key spaces.
3. The derivedKey keys are composite keys based on a subkey, which is any uddiKey, and an additional token, kss, which is a key-specific string. The kss is what differentiates keys and it can be assigned by a publisher or calculated algorithmically (UUID).

Another concept included in the UDDI key scheme is a key generator. A key generator is used to represent a key space. A publisher can save entities using keys from a certain key space only if that publisher owns the key generator that represents the key space. This restriction helps to secure unique keys. The key generator is a tModel entity, with a key that is in the form <subkey>:keyGenerator. By owning this tModel entity, a publisher can assign keys in the form <subkey>:<kss>. The publisher can also publish new tModel key generators in the form <subkey>:<kss>:keygenerator.

Key uniqueness and registry root key space

Instances of the UDDI registry can be configured as a root registry, or an affiliate registry.

Root registries define their own root key space by defining their own root key generator. This defines the total key space that the registry manages. All keys that the registry generates are within this key space. If allowed by policy, publishers can request sub-divisions of this key space by publishing new tModel key generators in the form <rootkeygenerator>:<subdivisionIdentifier>:keygenerator, and then can include publisher-supplied keys in subsequent publish requests that are in their allocated key space subdivision (<rootkeygenerator>:<subdivisionIdentifier>:<kss>).

To avoid key collisions, affiliate registries must establish their root key generator by first submitting a tModel:keygenerator request to the root registry they wish to be an affiliate of, and then using this subdivision of the root registry's key space as their own root key generator. This ensures that there are no collisions between keys generated or accepted by an affiliate registry, and other keys in the root registry key space.

To maintain key uniqueness, simple rules are applied. The registry generates new keys only in the key space defined by its own root key generator, and only accepts publisher-supplied keys that are in subdivisions of key space that the publisher owns (as the result of a previous successful tModel tModel:keygenerator publish request).

Simple example for a private root registry:

```
with a Root keygenerator:

uddi:aPrivateRegistryKeySpaceIdentifier:keygenerator

generates Entity Keys of format:

uddi:aPrivateRegistryKeySpaceIdentifier:<uuid>

depending on Policy, accepts tModel:keygenerator requests from Publishers for 'top-level' subdivisions
of format:
uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:keygenerator
```

Publishing tModel:keyGenerator requests for subdivisions of key space

Depending on policy, a publisher can submit a request for a top-level subdivision in the key space of the root registry for its own use. (The policy is whether the registry supports publisher supplied keys and whether a particular publisher's User entitlements allow the publisher to submit requests for key space).

As well as top-level subdivisions in the key space of the root registry, a publisher can also create additional subdivisions of key space.

Continuing from the previous example, a simple example of this is:

```
uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:keygenerator
```

This request for a further subdivision `a` is successful when it is requested by the publisher who previously requested, and owns, the tModel for the subdivision above (in this case, uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:keygenerator).

Publishing with a publisher-supplied key

After a publisher successfully requests a subdivision in the key space of a root registry, that publisher must establish and maintain its own scheme to ensure that the keys that are generated for use as publisher-supplied keys in subsequent publish requests are unique within the subdivision.

Valid schemes need to generate keys that are unique derived keys in the allocated key space subdivision, for example by including a unique (incremented) numeric index.

Continuing from the previous example, a simple example of this is:

For a key space subdivision that results from the tModel:keyGenerator request:

```
uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:keygenerator

valid keys are:

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:1

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:2
```

Use of digital signatures with the UDDI registry

In UDDI Version 3, publishers can digitally sign UDDI elements while they are publishing. The UDDI Version 3 schema supports the signing of businessEntity, businessServices, bindingTemplate, tModel, and publisherAssertion elements.

You can validate UDDI elements that are digitally signed to prove that they have not been modified or tampered with, and that their integrity is intact.

For full details about signing UDDI entities and verifying signatures, see *Appendix I: Support for XML Digital Signatures* in the UDDI Version 3.0.2. specification.

The UDDI registry does not validate signatures at the time that signed elements are published. When the signed elements are retrieved, it is the responsibility of the retrieving client to validate the signature and to provide a mechanism to ensure that the signer certificate is signed by a Certificate Authority (CA) that the client approves and trusts. If a signature is decrypted successfully by using the signer public key, it indicates that only the owner of the corresponding private key could have signed and published this element.

Generating a signature

The attributes of an element are included in the generation of an element signature. Therefore, all entity keys must be available when the signature is generated. Publishers can generate publisher-assigned keys for all the keys of an element before signing. Alternatively, publishers can publish the element without keys; this approach causes the registry node to generate the required entity keys and then retrieve, sign, and republish the signed element.

Validating a signature

The signature element to validate is in the top-level element that a call to the getXXDetails method returns. The client is responsible for the validation. The client must have previously imported the X509.3 certificate of the publisher, and validated that certificate based on the CA it trusts. In this way, the client has access to the public validation key of the publisher that corresponds to the private signing key that the publisher used to sign the entity before publishing it.

You can use the UDDI Version 3 Client to construct Java API for XML-based RPC (JAX-RPC) objects and to invoke the UDDI Version 3 WebService. As part of this client, you can use a helper class, com.ibm.uddi.v3.client.apilayer.xmldig.SignatureUtilities, to create and validate digital signatures on the UDDI Version 3 entities that support them. See the API documentation page for details of APIs in this class and the SignatureUtilitiesException exception.

For an example of how to use this class, see UDDIv3ClientSignedBusinessSample.java. inSamples for WebSphere Application Server .

For UDDI, digital signatures are used to sign the data. They are not used to authenticate the SOAP message.

UDDI registry Application Programming Interface

The UDDI Version 3 registry supports multiple versions of UDDI. It supports UDDI Version 1, Version 2, and Version 3.

For details of the Version 1 and Version 2 API, see http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2.

For details of the UDDI Version 3.0.2 API, see http://uddi.org/pubs/uddi_v3.htm.

The UDDI registry information in this information center defines the support that the UDDI registry provides for the UDDI Version 3.0.2 specification and associated addenda.

The following UDDI Version 3 API sets are supported:
- The UDDI V3 Inquiry API
- The UDDI V3 Publish API
- The UDDI V3 Custody and Ownership Transfer API
- The UDDI V3 Security API

Note: In DB2 for zSeries Version 7, the length of publish and inquiry strings are limited to 255 characters. If this limit is exceeded, error 10500 (E_Fatal) is returned. If you use a character set that uses multiple byte characters, it is easy to exceed this limit. Therefore, use care if you use this type of character set.

Inquiry API for the UDDI Version 3 registry

The Inquiry API provides four forms of query that follow broadly used conventions that match the needs of software that is traditionally used within registries.

- The browse pattern
- The drill-down pattern
- The invocation pattern
- Inquiry API functions

For more information refer to the UDDI Version 3 Specification.

Browse pattern for the UDDI registry:

Software that people use to explore and examine data, especially hierarchical data, requires browse capabilities. The browse pattern characteristically involves starting with some broad information, performing a search, finding general result sets, then selecting more specific information for drill-down patterns.

The UDDI API specifications accommodate the browse pattern with the *find_xx* API calls. These calls form the search capabilities that the API provides. The calls are matched with summary return messages that return overview information about the registered information that is associated with the inquiry message type and the search criteria specified in the inquiry.

A typical browse sequence might involve finding whether there is any information registered for a business you know about. This sequence starts with a call to *find_business*, perhaps passing the first characters of the business name that you know. This action returns a *businessList* result. This result is overview information, including keys, names, and descriptions, that is derived from the registered businessEntity information, matching on the name fragment that you provide. If the business you are looking for is in this list, you can use the *find_service* API call to drill down into the corresponding businessService information, and look for specific technical models, such as purchasing or shipping. Similarly, if you know the technical *fingerprint* (tModel signature) of a particular software interface, and you want to see if the business you are looking for provides a Web service that supports that interface, you can use the *find_binding* inquiry message.

Drilldown pattern for the UDDI registry:

When you have a key for one of the four main data types that are managed by a UDDI registry, you can use that key to access the full registered details of a specific data instance. The UDDI data types are businessEntity, businessService, bindingTemplate, and tModel. You can access the full registered information for any of these structures by passing a relevant key type to one of the *get_xx* API calls.

Continuing the example from the Browse pattern for the UDDI registry, one data item that is returned by all of the *find_x* return sets is key information. For the business you are interested in, the businessKey value that is returned in the contents of a businessList structure can be passed as an argument to the *get_businessDetail* message. The successful return to this message is a *businessDetail* message that contains the full registered information for the entity with the key value that is passed. This information will be a full *businessEntity* structure.

Invocation pattern for the UDDI registry:

To prepare an application to take advantage of a remote Web service that is registered within the UDDI registry by other businesses or entities, you must prepare that application to use the information found in the registry for the specific service being invoked.

The *bindingTemplate* data that is obtained from the UDDI registry represents the specific details about an instance of a given interface type, including the location at which a program starts interacting with the service. The calling application or program caches this information and uses it to contact the service at the registered address whenever the calling application needs to communicate with the service instance.

In remote procedure technologies that were previously popular, tools automate the tasks that are associated with caching, or hard coding, location information. However, there are problems when a remote service moves and the callers do not know about the move. There are many reasons why a remote service might move, for example, a server upgrade, disaster recovery, service acquisition, or a change to the business name.

When a call fails using cached information previously obtained from a UDDI registry, the proper behavior is to query the UDDI registry for fresh bindingTemplate information. If the data that is returned is different from the cached information, the service invocation can automatically retry the invocation, using the fresh information. If the result of this retry is successful, the new information replaces the cached information.

By using this pattern with Web services, a business that uses a UDDI registry can automate the recovery of a large number of partners without excessive communication and coordination costs. For example, if a business activates a disaster recovery site, most of the calls from partners would fail when they try to invoke services at the failed site. By updating the UDDI information with the new address for the service, partners who use the invocation pattern automatically locate the new service information and recover without further administrative action.

Inquiry API functions in the UDDI registry:

Use the inquiry API set to locate and obtain details about entries in a UDDI registry. The API is split into a number of functions, where each function requires some optional and mandatory arguments.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java (see UDDI Version 3 Client). To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Inquiry API calls:

find_binding
: Locates specific bindings in a registered businessService. Returns a bindingDetail message that contains zero or more bindingTemplate structures that match the criteria specified in the argument list.

find_business
: Locates information about one or more businesses. Returns a businessList message that matches the conditions specified in the arguments.

find_relatedBusinesses
: Locates information about businessEntity registrations that are related to a specific business entity whose key is passed in the inquiry. The related businesses feature is used to manage registration of business units and subsequently relate them based on organizational hierarchies or business partner relationships. Returns a relatedBusinessList message that contains results that match the conditions specified in the argument list.

find_service
: Locates specific services in a registered businessEntity. Returns a serviceList message that matches the conditions specified in the arguments.

find_tModel
: Locates a list of tModel entities that match a set of specified criteria. The response is a list of abbreviated information about registered tModel data that matches the specified criteria. The result is returned in a tModelList message.

get_bindingDetail
: Requests the runtime bindingTemplate information for the purpose of invoking a registered business API. Returns a bindingDetail message.

get_businessDetail
Returns complete businessEntity information for one or more specified businessEntity registrations that match the specified businessKey values. Returns a businessDetail message.

get_opertionalInfo
Gets full operational information pertaining to one or more entities in the registry. Returns an operationalInfos structure.

get_serviceDetail
Requests full information about a known businessService structure. Returns a serviceDetail message.

get_tModelDetail
Gets full details for a given set of registered tModel data. Returns a tModelDetail message.

For full details of the query syntax, refer to the UDDI Version 3 API specification at http://www.uddi.org/pubs/uddi_v3.htm.

Find_qualifiers for API functions in the UDDI registry:

Each of the APIs (find_business, find_service, find_binding, find_tModel and find_relatedBusinesses) accepts an optional findQualifiers argument, which can contain multiple findQualifier values.

The following list contains the findQualifier short names, a brief description, and the appropriate find function. The arguments available are:

andAllKeys
This changes the behavior for identifyerBag to AND keys rather than OR them. This is the **default** for categoryBag and tModelbag. Applicable to find_business, find_service, find_binding and find_tModel (but not for find_relatedBusinesses).

approximateMatch
Signifies that wildcard search behavior is desired. This is no longer the default behavior (see 'exactMatch'). This applies to find_business, find_service, find_binding, find_tModel and find_relatedBusiness.

binarySort
Allows for greater speed in sorting. It causes a binary sort by name, as represented in Unicode codepoints. It is applicable to find_business, find_service and find_tModel only.

bindingSubset
This is used only in conjunction with a categoryBag argument in the find_business or find_services APIs.

caseInsensitiveMatch
Signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed without regard to case. It is applicable to find_business, find_service and find_tModel.

caseInsensitiveSort
Signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed without regard to case. This overrides the default case sensitive sorting behavior.

caseSensitiveMatch
Signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed with regard to case. This is the **default** behavior. It is applicable to find_business, find_service, find_binding, find_tModel and find_relatedBusinesses.

caseSensitiveSort
Signifies that the result set should be sorted with regard to case. this is the **default** behavior. It is applicable to find_business, find_service and find_tModel.

combineCategoryBags
This may only be used in the find_business and find_service calls.
- In the case of find_business, this makes the categoryBag entries for the full businessEntity element behave as though all categoryBag elements found at the businessEntity level and in all contained or referenced businessService elements and bindingTemplate elements were combined.

- In the case of find_service, this makes the categoryBag entries for the full businessService element behave as though all categoryBag elements found at the businessService level and in all contained or referenced elements in the bindingTemplate elements were combined.

diacriticInsensitiveMatch
Signifies that matching behavior for name, keyValue and keyName (where applicable) should be performed without regard to diacritics. Support for this findQualifier is optional. It applies to find_business, find_service, find_binding, find_tModel and find_relatedBusinesses.

diacriticSensitiveMatch
Signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed with regard to diacritics. This is the **default** behavior. It applies to find_business, find_service, find_binding, find_tModel and find_relatedBusinesses.

exactMatch
Signifies that only entries with names, keyValues and keyNames (where applicable) that exactly match the name argument passed in, after normalization, will be returned. It is sensitive to case and diacritics where applicable and is the **default** behavior. It applies to find_business, find_service, find_binding, find_tModel and find_relatedBusinesses.

signaturePresent
This is used with any find API to restrict the result set to entities which either contain an XML Digital Signature element, or are contained in an entity which contains one. It applies to find_business, find_service, find_binding, find_tModel and find_relatedBusinesses.

orAllKeys
This changes the behavior for tModelBag and categoryBag to OR the keys within a bag, rather than to AND them. It is not possible to OR the categories and retain the default AND behavior of the tModels. For the find_business qualifier this is the **default** behavior for identifierBag, and it is applicable to find_service, find_binding (for categoryBag and tModelbag) and find_tModel where it is the **default** behavior for identifierBag and applicable to categoryBag.

orLikeKeys
Used when a bag container (that is a categoryBag or identifierBag) contains multiple keyedReference elements. In this situation any keyedReference filters that come from the same namespace (have the same tModelKey value) are OR'd together rather than AND'd. It is applicable to find_business, find_service, find_binding and find_tModel.

serviceSubset
This is only used with the find_business API and used only in conjunction with the categoryBag argument. It causes the component of the search that involves categorization to use only the categoryBag elements from contained or referenced businessService elements within the registered data and ignores any entries found in the categoryBag which are not direct descendent elements of registered businessEntity elements.

sortByNameAsc
This causes the result set returned by a find or get inquiry API to be sorted on the name field in ascending order. It is applicable to find_business, find_service, find_tModel and find_relatedBusinesses. This findQualifier takes precedence over sortByDateAsc and sortByDateDesc qualifiers, but if a sortByDateXxx findQualifier is used without a sortByNameXxx qualifier, sorting is performed based on date with or without regard to name.

sortByNameDesc
This causes the result set returned by a find or get inquiry API to be sorted on the name field in descending order. It is applicable to find_business, find_service, find_tModel and find_relatedBusinesses. This findQualifier takes precedence over sortByDateAsc and sortByDateDesc qualifiers, but if a sortByDateXxx findQualifier is used without a sortByNameXxx qualifier, sorting is performed based on date with or without regard to name.

sortByDateAsc
This causes the result set returned by a find or get inquiry to be sorted based on the most recent date when each entity, or any entities they contain, were last updated, in ascending chronological order (the oldest is returned first). When used in conjunction with names in the result set returned, the date-based sort is secondary to the name-based sort (that is, the results are sorted within name by date, oldest to newest). This is the **default** behavior for find_binding and is applicable for find_business, find_service, find_tModel and find_relatedBusinesses.

sortByDateDesc
This causes the result set returned by a find or get inquiry to be sorted based on the most recent date when each entity, or any entities they contain, were last updated, in descending chronological order (the most recently changed are returned first). When used in conjunction with names in the result set returned, the date-based sort is secondary to the name-based sort (that is, the results are sorted within name by date, newest to oldest). This is applicable for find_business, find_service, find_binding, find_tModel and find_relatedBusinesses.

suppressProjectedServices
Signifies that service projections **MUST NOT** be returned by the find_service or find_business APIs with which this findQualifier is associated. This findQualifier is automatically enabled by default whenever find_service is used without a businessKey.

For further details on the findQualifiers refer to the UDDI Version 3 Specification documentation.

Publish API for the UDDI Version 3 registry

Use the UDDI publish API to publish, delete, and update information that is contained in a UDDI registry. The messages that are defined in this section all behave synchronously.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java (see UDDI Version 3 Client). To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Publication API calls:

add_publisherAssertions
Causes one or more publisherAssertions to be added to an individual publisher's assertion collection.

delete_binding
Causes one or more instances of bindingTemplate data to be deleted from the UDDI registry.

delete_business
Removes one or more business registrations and all direct contents from a UDDI registry.

delete_publisherAssertions
Causes one or more publisherAssertion elements to be removed from a publisher's assertion collection.

delete_service
Removes one or more businessService elements from the UDDI registry and from its containing businessEntity parent.

delete_tModel
Logically deletes one or more tModel structures. Logical deletion hides the deleted tModels from find_tModel result sets but does not physically delete them, so they are returned on a get_registeredInfo request.

get_assertionStatusReport
Provides administrative support for determining the status of current and outstanding publisher assertions that involve any of the business registrations managed by the individual publisher account. Using this message, a publisher can see the status of assertions that they have made, as well as see assertions that others have made that involve businessEntity structures controlled by the calling publisher account.

get_publisherAssertions
Obtains the full set of publisher assertions that are associated with an individual publisher account. Publisher assertions are used to control publicly visible business relationships.

get_registeredInfo
Gets an abbreviated list of all businessEntity and tModel data that are controlled by the individual associated with the credentials passed.

save_binding
Saves or updates a complete bindingTemplate element. this message can be used to add or update one or more bindingTemplate elements as well as the container/contained relationship that each bindingTemplate has with one or more existing businessService elements.

save_business
Saves or updates information about a complete businessEntity element. This API has the broadest scope of all the save_xx API calls in the publisher API, and can be used to make sweeping changes to the published information for one or more businessEntity elements controlled by an individual.

save_service
Adds or updates one or more businessService elements exposed by a specified businessEntity.

save_tModel
Adds or updates one or more registered tModel elements.

set_publisherAssertions
Manages all of the tracked relationship assertions associated with an individual publisher account.

For full details of the query syntax, refer to the UDDI Version 3 API specification at http://www.uddi.org/pubs/uddi_v3.htm.

Custody and Ownership Transfer API for the UDDI Version 3 registry

Use the UDDI Custody and Ownership Transfer API to transfer custody or ownership of one or more entities that are contained in a UDDI Version 3 registry. The UDDI Version 3 registry supports only intra-node ownership transfer; it does not support inter-node custody transfer.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java (see UDDI Version 3 Client). To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Custody and Ownership Transfer API calls:

discard_transferToken
Discards a transferToken that is obtained through the get_transferToken API.

get_transferToken
Initiates the transfer of ownership of one or more businessEntity or tModel entities from one publisher to another. When the API is invoked, no actual transfer takes place. Instead, the relinquishing publisher uses this API to obtain permission from the custodial node, in the form of a transferToken, to perform the transfer. The relinquishing publisher gives the transferToken to the recipient publisher, who must invoke the transfer_entities API to actually transfer the entities.

transfer_entities
Performs the actual transfer of entities when called by the recipient publisher. The recipient publisher must specify an unexpired transferToken on the call.

For details of the query syntax, refer to the UDDI Version 3 API specification at http://www.uddi.org/pubs/uddi_v3.htm.

Security API for the UDDI Version 3 registry

The UDDI Version 3 registry has an independent security API, unlike UDDI Version 1 and Version 2, where the security API was part of the publish API.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java (see UDDI Version 3 Client). To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Security API calls:

discard_authToken
Informs a node that a previously obtained authentication token is no longer required and is no longer valid if it is used after this message is received. The token is discarded and the session is effectively ended.

get_authToken
Requests an authentication token in the form of an authInfo element from a UDDI node.

For full details of the query syntax, refer to the UDDI Version 3 API specification at http://www.uddi.org/pubs/uddi_v3.htm.

UDDI Version 3 Client

You can use the UDDI Version 3 Client for Java to access the UDDI Version 3 application programming interfaces (APIs) from Java client code.

The UDDI Version 3 Client for Java is a JAX-RPC Java class library that provides an API that can be used by client programs to interact with a Version 3 UDDI registry. This class library can be used to construct UDDI JAX-RPC objects and to invoke the UDDI Version 3 WebService.

This client also contains an XML Digital Signature utility class called SignatureUtilities, provided to construct and validate Digital Signatures on UDDI elements. See Use of digital signatures with the UDDI registry for full details.

Multiple language encoding support

The UDDI Version 3 API supports both UTF-8 and UTF-16 encoding. Internally, UTF-16 characters are stored as UTF-8 characters. This behavior is transparent to the user application.

Client Jar

WebSphere Application Server provides a class library:

uddiv3client.jar
This jar contains the JAX-RPC UDDI Version 3 types and UDDI WebService invocation classes.

This jar is located in *app_server_root*/UDDIReg/clients

The UDDI Version 3 client provides port types which map onto the UDDI Version 3 SOAP inquiry, publish, custody transfer and security APIs. These APIs are protected as described in Access control for UDDI registry interfaces. A client program using the UDDI Version 3 client should get the appropriate port type for the request that is to be issued (such as the UDDI_Publication_PortType for a save_business request). If the role mappings are such that the request will require a WebSphere Application Server authenticated user ID, the client program should pass the user ID and password by setting the relevant properties on the JAX-RPC stub for that port.

UDDI Version 3 Client samples

Samples illustrating the use of the Version 3 Client are available through the UDDI registry link on the Samples for WebSphere Application Server page of the IBM developerWorks WebSphere Web site.:

UDDIv3ClientBindingSample.java
An example of how to save and find Binding Templates.

UDDIv3ClientBusinessSample.java
An example of how to save and find Business Entities.

UDDIv3ClientServiceSample.java
An example of how to save and find Business Services.

UDDIv3ClientSignedBusinessSample.java
An example of how to sign and verify a Business Entity.

UDDIv3ClientTModelSample.java
An example of how to save and find TModels.

UDDIv3ClientSignedTModelSample.java
An example of how to sign and verify TModels.

These classes contain details on how to compile and execute the samples.

HTTP GET services for UDDI registry data structures

The UDDI registry offers an HTTP GET service for access to the XML representations of the businessEntity, businessService, bindingTemplate, and tModel UDDI data structures. The URL at which these structure are accessible uses the entity key as a URL parameter. The XML element that is returned is a businessDetail, serviceDetail, bindingDetail or tModelDetail element, according to the type of entity key supplied.

XML for both UDDI Version 2 and Version 3 can be retrieved, each at different URLs. The formats of the URLs to send the HTTP GET requests to are:

For UDDI Version 2:
http://<server>:<port>/uddisoap/get?<entityKey type>=<v2 entityKey>

For UDDI Version 3:
http://<server>:<port>/uddiv3soap/get?<entityKey type>=<v3 entityKey>

For example, if <server> = myserver.com and <port>=9080, you can access the uddi-org:types tModel at the following URLs:

UDDI Version 2:
http://myserver.com:9080/uddisoap/get?tModelKey=uuid:c1acf26d-9672-4404-9d70-39b756e62ab4

UDDI Version 3:
http://myserver.com:9080/uddiv3soap/get?tModelKey=uddi:uddi.org:categorization:types

A number of UDDI property and policy settings relate to the HTTP GET services:

- Version 3 HTTP GET for UDDI entities
  - Node supports HTTP GET
  - URL prefix for Version 3 GET servlet
  - Node generates discovery URLs
- Version 2 HTTP GET for discovery URLs
  - Prefix for generated discovery URLs
  - Node generates discovery URLs

For details, refer to UDDI node miscellaneous policy settings and UDDI node settings.

UDDI registry SOAP service end points

UDDI Version 3 supports multiple versions of SOAP API services. Depending on the security settings for WebSphere Application Server and the user data constraint transport guarantee settings for the UDDI SOAP service, UDDI Version 3 supports different end points for different services.

The default context root and URL values that are listed in this topic apply when you enable WebSphere Application Server security and do not change the default supplied security settings. If you do not use the default security settings, the context root and URL values might differ (see Configuring UDDI registry security for more information about the various security settings).

In the following URLs, the variables have the following values:

- *host_name* is the name of the machine that is running the relevant profile.
- *http_port* is the internal HTTP port for the profile, for example 9080.
- *ssl_port* is the internal SSL port for the profile, for example 9443.

Version 1 and Version 2 SOAP API services

Inquiry service
Default (soap.war) context-root='/uddisoap' and url-pattern = 'inquiryAPI' or 'inquiryapi'.

Default URL: http://*host_name*:*http_port*/uddisoap/inquiryapi

Publish service
Default (soap.war) context-root='/uddisoap' and url-pattern = 'publishAPI' or 'publishapi'.

Default URL: https://*host_name*:*ssl_port*/uddisoap/publishapi or http://*host_name*:*http_port*/uddisoap/publishapi

Version 3 SOAP API services

Inquiry service
Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Inquiry_Port'

Default URL: http://*host_name*:*http_port*/uddiv3soap/services/UDDI_Inquiry_Port

Publish service
Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Publish_Port'

Default URL: https://*host_name*:*ssl_port*/uddiv3soap/services/UDDI_Publish_Port or http://*host_name*:*http_port*/uddiv3soap/services/UDDI_Publish_Port

Custody transfer service
Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Custody_Port'

Default URL: https://hostname:9443/uddiv3soap/services/UDDI_Custody_Port or http://hostname:9080/uddiv3soap/services/UDDI_Custody_Port

Security service
Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Security_Port'

Default URL: https://*host_name*:*ssl_port*/uddiv3soap/services/UDDI_Security_Port or http://*host_name*:*http_port*/uddiv3soap/services/UDDI_Security_Port

There is also an endpoint for using HTTP GET to return XML representations of UDDI entities, described in HTTP GET Services for UDDI registry data structures.

Additional information

If you configure the UDDI registry to use WebSphere Application Server security, and you do not change the default data confidentiality settings for the UDDI SOAP service, services with default end point URLs with HTTPS and SSL port require that their data is transported confidentially. Requests that do not use HTTPS are rejected.

If you configure the UDDI registry to use WebSphere Application Server security and you change the data confidentiality setting for the UDDI SOAP service to NONE, or you disable WebSphere Application Server security, services with default end point URLs with HTTPS and SSL port can also use HTTP and HTTP port.

To understand how access to the SOAP APIs is protected, see Access Control for UDDI registry Interfaces.

The UDDI registry SOAP API

To use the SOAP API, construct a properly formed UDDI message in the body of a SOAP request, and send it using HTTP POST to the URL of the API that the request relates to. The response is returned in the body of the HTTP reply.

The UDDI registry samples include samples that demonstrate how to program directly to the SOAP API. Although the samples are written in Java code, you can use other programming languages to create your

SOAP client, as long as you still send requests that are compliant to the SOAP specification. Valid UDDI requests must conform to the UDDI schema, and must be as detailed in the UDDI specification:

http://www.uddi.org/pubs/uddi_v3.htm

For more information on using the SOAP API, refer to UDDI registry application programming interface.

UDDI4J programming interface (Deprecated)

The UDDI4J Version 2 APIs are deprecated in this version of WebSphere Application Server. The UDDI Version 3 Client for Java is the preferred API for accessing UDDI using Java code.

WebSphere Application Server provides UDDI4J classes in the com.ibm.uddi.jar file. This file contains classes that support Version 1 and Version 2 of the UDDI specification, providing compatability with earlier versions of WebSphere Application Server. The UDDI4J classes in this file are deprecated.

The UDDI4J methods map onto the UDDI Version 1 and Version 2 SOAP inquiry and publish APIs. These APIs are protected, as described in Access control for UDDI registry interfaces. If the role mappings for these APIs are such that requests to these interfaces require a WebSphere Application Server authenticated user ID, a client program using UDDI4J must pass the user name and password, by setting the system properties http.basicAuthUserName and http.basicAuthPassword. A UDDI4J client program can also specify details for a proxy server, including a user name and password, using the following system properties:

- http.proxyHost
- http.proxyPort
- http.proxyUserName
- http.proxyPassword

UDDI EJB Interface (Deprecated)

Use the Enterprise JavaBeans (EJB) application programming interface (API) of the UDDI registry component to publish, find, and delete UDDI entries. The UDDI EJB interface is deprecated in WebSphere Application Server Version 6.0 and later versions, and supports UDDI version 2 API requests only.

The client classes that are required for the EJB interface are contained in *app_server_root*/UDDIReg/clients/uddiejbclient.jar. You can read the Java documentation for these classes at the WebSphere Application Server application programming interface reference information.

The EJB API is contained in two stateless session beans, one for the inquiry API (com.ibm.uddi.ejb.InquiryBean) and one for the publish API (com.ibm.uddi.ejb.PublishBean), whose public methods form an EJB interface for the UDDI registry. All the public methods on the InquiryBean class correspond to UDDI Version 2 inquiry API functions, and all the public methods on the PublishBean class correspond to UDDI Version 2 publish API functions. Not all UDDI Version 2 API functions are implemented, for example get_authToken, discard_authToken, get_businessDetailExt.

In each interface, there are groups of overloaded methods that correspond to the operations in the UDDI 2.0 specification. There is a separate method for each major variation in function. For example, the single UDDI operation find_business is represented by ten variations of findBusiness methods, with different variations to find by name, find by categoryBag argument, and so on.

The arguments for the EJB interface methods are Java objects in thee com.ibm.uddi.datatypes packag. Generally, there is a one-to-one correspondence between classes in this package and elements of the UDDI Version 2 XML schema. Exceptions to this correspondence are, for example, where UDDI XML elements can be represented by a single String. See the Java documentation for package com.ibm.uddi.datatypes for more information, at Javadoc welcome page.

The methods on the EJB InquiryBean class map to the EJB inquiry role, and those of the EJB PublishBean class map to the EJB publish role. The EJB inquiry and publish roles protect the EJB interface, as described in Access control for UDDI registry interfaces. If the role mapping is such that a method requires a WebSphere Application Server authenticated user ID, a client program can supply the user ID and password, either when prompted by WebSphere Application Server, or by providing application code that logs in to the default realm using the user ID and password. Use the `sas.client.props` configuration file to determine how to specify the user ID and password (see Configuring security with scripting).

Using the EJB client

In this section, it is assumed that both WebSphere Application Server and the UDDI registry are installed, and that they are both running. You cannot use the EJB client from a machine that does not have WebSphere Application Server installed.

1. Set up your environment to communicate with WebSphere Application Server:
   `. app_server_root/bin/setupCmdLine.sh` (note the space between the period and *app_server_root*)
2. Verify that your CLASSPATH includes the uddiejbclient.jar file (from the *app_server_root*/UDDIReg/clients directory), and the code for your client.
3. Compile your EJB client programs:
   ```
   $JAVA_HOME/bin/javac -extdirs $WAS_EXT_DIRS:$JAVA_HOME/jre/lib/ext -classpath
   $WAS_CLASSPATH:$CLASSPATH yourcode.java
   ```
4. Run the compiled programs:
   ```
   $JAVA_HOME/bin/java -Djava.ext.dirs=$WAS_EXT_DIRS:$JAVA_HOME/jre/lib/ext
   -Dwas.install.root=$WAS_HOME -Dserver.root=$WAS_HOME $CLIENTSAS $CLIENTSOAP -cp
   $WAS_CLASSPATH:$WAS_HOME/UDDIReg/clients/uddiejbclient.jar:$CLASSPATH <class name> <args>
   ```

Ensure that your PATH statement starts with *app_server_root*/java/bin.

Using the UDDI registry user interface

The UDDI registry user interface (also referred to as the UDDI registry user console) is a graphical interface that you can use to explore the UDDI registry.

Before you begin

Configure the application server hosting the UDDI registry for UTF-8 encoding support, as described in Configuring application servers for UTF-8 encoding. The UDDI user console does not support UTF-16 encoding.

About this task

The user console provides a graphical user interface to the majority of the UDDI Version 3 API. The user console is not intended to support the full API set. There is some focus on inquiry operations, because the main purpose of the UDDI user console is to allow you to issue inquiry requests and to familiarize yourself with general UDDI concepts. The following list describes areas for which support through the user console is not provided, and other known restrictions to the user console.

- Help is provided in the form of explanatory text on the screens.
- You cannot specify maximum rows on find queries. You can set the single maximum rows value for the registry by using the *Maximum inquiry result set size* general property on the administrative console.
- The identifier feature is not supported in the find business area or the find technical model (tModel) area.
- In the add business area, no support exists for adding Discovery URLs, Identifiers or Digital Signatures.
- In the add technical model (tModel) area, no support exists for adding Identifiers or Digital Signatures.

The exact behavior of the user console depends on the following configurable factors:

- Whether WebSphere Application Server security is enabled.
- How the UDDI registry GUI role mappings are set. The UDDI registry supports a number of security roles, including two for the user console: GUI_Publish_User and GUI_Inquiry_User.
- How the UDDI registry GUI SSL transport guarantee constraints are set. The UDDI registry allows SSL settings to be configured and this includes two settings for the user console.

The following table summarizes the behavior of the UDDI registry user console.

Table 22. Behavior of the UDDI user console

WebSphere Application Server security status	**URL used to access the UDDI user console**	**Behavior of the UDDI user console**
Enabled	http://*host_name*:*http_port*/uddigui	Inquiry requests do not require authentication; they use the HTTP URL and are not secure. Publish requests do require WebSphere Application Server authentication. When you access the publish pane you will be dynamically redirected to use HTTPS, and will be prompted for a user ID and password. For the request to be successful , the authenticated user must be registered as a UDDI publisher. If the GUI_Inquiry_User role is mapped to all authenticated users, and the transport guarantee in the user data constraint section for that role is set to CONFIDENTIAL, *all* requests, including inquiry, require authentication and use of HTTPS.
	https://*host_name*:*ssl_port*/uddigui	Requests are secure; you are prompted to authenticate with a user ID and password. For the request to be successful , the authenticated user must be registered as a UDDI publisher.
Disabled	http://*host_name*:*http_port*/uddigui	No requests, either publish or inquire, are authenticated and the data flow is *not* secure (non SSL). Even though SSL transport-guarantee settings are defined, they are not enforced if security is disabled. All publish operations are performed using a user ID of UNAUTHENTICATED or a value that can be configured using the administrative console or the JMX management interface (this applies to new requests only).
	https://*host_name*:*ssl_port*/uddigui	No requests, either publish or inquire, are authenticated, but the data flow is secure because the SSL URL and port are used explicitly. All publish operations are performed using a user ID of UNAUTHENTICATED or a value that can be configured using the administrative console or the JMX management interface (this applies to new requests only).

The variables in the table have the following values:

- *host_name* is the name of the machine that is running the relevant profile.
- *http_port* is the internal HTTP port for the profile, for example 9080.
- *ssl_port* is the internal SSL port for the profile, for example 9443.

1. Start the UDDI application, if it is not already running.
2. Open a browser window and ensure that cookies are enabled.
3. Access the UDDI registry user console using one of the following default URLs.
   - http://*host_name*:*http_port*/uddigui
   - https://*host_name*:*ssl_port*/uddigui
4. Optional: You can change the look and feel of the user console by modifying the appropriate *.css* stylesheet files.

   You can edit style class definitions in these files to alter the overall theme of the UDDI registry user console, including font attributes, layout and colors.

   The *.css* files are located in the following directory: *profile_root*/installedApps/*cell_name*/UDDIRegistry.*node_name*.*server_name*.ear/v3gui.war/theme.

   Refresh the browser window after changing a stylesheet file, for the changes to take effect.

Results

The user console displays the default frameset containing the following items:

- The header frame.
- The navigation frame showing find options.
- The details frame.

What to do next

You can now use the UDDI user console to find, edit or publish UDDI information.

Finding an entity using the UDDI registry user interface

You can use the UDDI registry user interface to find services, businesses and technical models.

Before you begin

Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in “Using the UDDI registry user interface” on page 833.

About this task

This task describes how to use the UDDI registry user interface (also referred to as the UDDI registry user console) to find services, businesses and technical models.

1. Activate the **Find** tab by clicking it, or by clicking the **Find** link at the top of the page or on the Welcome page.
2. To perform a quick find, complete the following steps:
   a. In the **Quick Find** section of the **Find** tab, select the kind of entity you want to find; service, business or technical model.
   b. In the **Starting with** field, enter name of the entity. Use the ’%’ wildcard character to search for a partial name.
   c. Click **Find**.

   The results are displayed in the detail frame on the right.
3. To perform an advanced find, complete the following steps:
   a. In the **Advanced Find** section of the **Find** tab, click the appropriate link for the kind of entity you want to find; service, business or technical model. The advanced search form is displayed in the frame to the right.
   b. Enter your search criteria in the advanced search form, and select any find qualifiers you require. You must enter at least one name to search for, using the **Add Name** link. You can use this link to enter multiple names. You can also add multiple categorizations. To add a categorization, use the **Show category tree** link in the **Categorizations** section to display, in the pane on the left, a tree of categories (or taxonomies) defining the types of item to find according to various classification systems. Expand the tree to find the category that you want, click the category to add the information to the advanced search form, then use the **Add Categorization** link to include the category in the search.
   c. Click **Find *entities***.

   The results are displayed in the detail frame on the right.

Publishing an entity using the UDDI registry user interface

You can use the UDDI registry user interface to publish businesses and technical models.

Before you begin

Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in “Using the UDDI registry user interface” on page 833.

About this task

This topic describes how to use the UDDI registry user interface (also referred to as the UDDI registry user console) to publish businesses and technical models. To publish a service, first publish a business, and then add a service to that business, as described in Editing or deleting an entity using the UDDI registry user interface.

1. Activate the **Publish** tab by clicking it, or by clicking the **Publish** link at the top of the page or on the Welcome page.
2. To publish an entity by name only, use the quick publish section as follows:
   a. In the **Quick Publish** section of the **Publish** tab, select the kind of entity you want to publish; business or technical model.
   b. In the **Name** field, enter name of the entity.
   c. Click **Publish**.

   The details of the published entity are displayed in the frame on the right.
3. To publish an entity with more information, complete the following steps:
   a. In the **Advanced Publish** section of the **Publish** tab, click the appropriate link for the kind of entity you want to publish; business or technical model. The advanced publish form is displayed in the frame to the right.
   b. Enter the details for the entity in the advanced publish form. You can enter multiple names, descriptions, contacts or categorizations by using the relevant **Add** link. To add a categorization, first use the **Show category tree** link in the **Categorizations** section to display, in the pane on the left, a tree of categories (or taxonomies) defining the types of item to publish according to various classification systems. Expand the tree to find the category that you want, click the category to add the information to the advanced publish form, then click the **Add Categorization** link.
   c. Click **Publish *entity*** to publish the business or technical model to the UDDI registry.

Editing or deleting an entity using the UDDI registry user interface

You use the UDDI registry user interface to edit or delete the businesses and technical models that you own, or to add services to businesses.

Before you begin

Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in “Using the UDDI registry user interface” on page 833.

About this task

This task describes how to use the UDDI registry user interface (also referred to as the UDDI registry user console) to edit or delete the businesses and technical models that you own, or add services to businesses.

1. Activate the **Publish** tab by clicking it, or by clicking the **Publish** link at the top of the page or on the Welcome page.
2. At the bottom of the **Publish** tab is the **Registered Information** section. Click **Show owned entities** to show the businesses and technical models that you have registered in the UDDI registry.
3. Optional: To delete a business or technical model, click the **Delete** link in the **Actions** column for that entity.

Note: When you delete a technical model, it is hidden rather than physically deleted, as specified by the UDDI Version 3.0 specification. If you click the **Shown owned entities** link the technical model will still appear, but you will not be able to find it using the **Find** function. All other entities are deleted from the UDDI registry in the normal way.

4. Optional: To edit a business or technical model, click the **Edit** link in the **Actions** column for that entity, fill in the required details and click **Update *entity*** to save the changes in the UDDI registry.
5. Optional: To add a service to a business, click the **Add service** link in the **Actions** column for the business. Fill in the details and click **Add Service** to publish the service to the UDDI registry. The service details are displayed.

Creating business relationships using the UDDI registry user interface

If your business has an association with another business in the UDDI registry, for example a preferred supplier, you can describe this association in the UDDI registry by creating a *business relationship*.

Before you begin

1. The UDDI registry contains the following default relationship types:

   Parent-child
   A hierarchical relationship exists between the two business entities, which might represent, for example, a large organization and a subsidiary.

   Peer-peer
   The two business entities represent peer organizations, for example a company and its supplier.

   Identity
   The two business entities represent the same organization.

   If you require a different relationship type, create a user-defined value set to represent the relationship type that you require, as described in “User-defined value set support in the UDDI registry” on page 768.
2. Each business that is involved in the relationship must already exist in the UDDI registry.
3. Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in “Using the UDDI registry user interface” on page 833.

About this task

Perform this task when you want to publicize an association between two businesses in the UDDI registry. For example, your organization, represented by a business entity in the UDDI registry, might have several departments, each one represented by a different business entity in the UDDI registry. You might want to declare these departments as being linked to the parent organization, by creating parent-child relationships between the appropriate business entities.

1. Activate the **Publish** tab by clicking it, or by clicking the **Publish** link at the top of the page or on the Welcome page.
2. Under **Registered Information**, click **Show owned entities**. The entities that you own are displayed in the detail frame on the right.
3. In the section for the businesses that you own, find the business that you want to link from, and click the **Add relationship** link for that business. The Add Business Relationship pane appears. The business key for the business that you selected is already listed in the From section.
4. Click **Add** to add the second business, the business that you want to link to. The advanced find pane appears.
5. Find the second business, as described in the advanced find step of “Finding an entity using the UDDI registry user interface” on page 835.

6. Click **Select** to add the second business to the relationship. If you want to change the positions of the businesses, click **Swap**.
7. Select the type of relationship from the **Type** list.
8. If required, type a description in the **Usage** field.
9. Click **Add relationship** to create the relationship. If you own both businesses, no further action is required. The relationship appears as a *publisher assertion* in the list of entities that you own. The status of the assertion is complete.
10. If you do not own the second business, the status of the assertion is pending. The owner of the second business must create a relationship from their business to yours, for the relationship to be complete and visible to other parties. The relationship type must match the type that you chose earlier.

Results

The business relationship is published to the UDDI registry. The UDDI user interface only shows publisher assertions for entities that you own. To view other relationships, use the UDDI inquiry API provided.

What to do next

For more information about publisher assertions, refer to the UDDI specification.

To remove an assertion that you own, display your owned entities and click the **Delete** link for the relevant publisher assertion. If the business in the To field is owned by someone else, the status of the assertion becomes pending, and the relationship is no longer visible to other parties.

Example: Publishing a business, service and technical model using the UDDI registry user interface

This example describes how to use the UDDI registry user interface to publish a used car business called Modern Cars to the UDDI registry, and how to publish a service and technical model for the business.

Before you begin, make sure that the UDDI registry application is started, then display the UDDI user interface as described in “Using the UDDI registry user interface” on page 833.

Adding the Business

1. Click the **Publish** tab to activate the **Publish** pane.
2. Under **Advanced Publish** click **Add a business**. The advanced publish form is displayed on the right.
3. Type ’Modern Cars’ in the **Name** field for the business. Select the language of the business name from the drop down list, then click **Add Name** to add the name to the business.
4. Type a description for the business, such as ’Used cars for sale’ in the **Description** field. Select the language of the description from the drop down list, then click **Add Description** to add the description to the business. You can add multiple descriptions in a variety of languages as required.
5. In the **Contact** section, type your name as a contact for customers of the business. Select the language as before, and click **Add Contact**. The business contact form is displayed. Fill in the details, using the **Add *entity*** links to add the information as you reach the end of each subsection. Note that all the text fields in the form are cleared when you click an **Add *entity*** link. Click **Add Contact** to save the contact information into the Modern Cars business.
6. Use the **Categorizations** section to describe the Modern Cars business according to the NAICS 2002 categorization system:
   a. Click **Show category tree** to display the various categorization systems.

b. Expand the **NAICS 2002** tree, then in that tree expand **Retail Trade [44]** → **Motor Vehicle and Parts Dealers [441]** → **automobile Dealers [4411]** → **Used Car Dealers [44112]**. Click the **Used Car Dealers [441120]** category to add the category type, key name and key value to the advanced publish form.

c. Click **Add Categorization** to add the information to the business.

d. Close the category tree by clicking the **close** link near the top of the tree.

7. Click **Publish Business** to publish the Modern Cars business to the UDDI registry. The details of the business will be displayed.

Adding a service to the business

1. Click the **Publish** tab to activate the **Publish** pane.
2. Under **Registered Information** click **Show owned entities** to display the Modern Cars business, and any other entities that are owned by you.
3. Click **Add service** in the **Actions** column of the Modern Cars business. The publish service page is displayed.
4. Add a name and description for the service, in the same way as for the business itself.
5. Click **Add a Service Binding** to display the service binding form. Enter an access point (the URL for the service on the network) and a description for the service binding. Click **Add Technical Model Instance Information** to display a page where you can describe and publish a technical model instance for the service binding. In the technical model information page, click **Add Technical Model**. Search for the technical model which is used by the instance, select the technical model from the results and click **Add**. Fill in the other fields on the form and click **Add Technical Model Instance**. Click **Add Binding** to save the information into the service.
6. Add a categorization as you did for the business itself.
7. Click **Add Service** to publish the service to the UDDI registry.

Adding a technical model

1. In the **Advanced Publish** section on the left, click **Add a technical model** to display the publish technical model form on the right.
2. Add a name and description for the technical model, in the same way as for the business and the service.
3. Click **Add an Overview Document** to display the overview document form. The overview document describes the technical model. Type the location of the overview document in the **Overview URL** field, and click **Add Overview Document URL**. Add a description and click **Add Overview Document** to save the information in the technical model.
4. Add a categorization in the same way as for the business.
5. Click **Publish Technical Model** to publish the technical model to the UDDI registry.

Related tasks

“Publishing an entity using the UDDI registry user interface” on page 835
You can use the UDDI registry user interface to publish businesses and technical models.

“Creating business relationships using the UDDI registry user interface” on page 837
If your business has an association with another business in the UDDI registry, for example a preferred supplier, you can describe this association in the UDDI registry by creating a *business relationship*.

Web Services Invocation Framework (WSIF): Enabling Web services

WSIF is a WSDL-oriented Java API. You use this API to invoke Web services dynamically, regardless of the service implementation format (for example, enterprise bean) or the service access mechanism (for example, Java Message Service (JMS)).

Before you begin

The WSIF framework includes an EJB provider for EJB invocation using Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). However, for EJB(IIOP)-based Web service invocation you should instead invoke RMI-IIOP Web services using JAX-RPC.

About this task

Using WSIF, you can move away from the usual Web services programming model of working directly with the SOAP APIs, towards a model where you interact with representations of the services. You can therefore work with the same programming model regardless of how the service is implemented and accessed.

To use WSIF, see the following topics:

- Learn about WSIF.
- Use WSIF to invoke Web services.
- Install and manage WSIF.
- Invoke a WSDL-based Web service through the WSIF API.

Learning about the Web Services Invocation Framework (WSIF)

The Web Services Invocation Framework (WSIF) is a WSDL-oriented Java API. You use this API to invoke Web services dynamically, regardless of the service implementation format (for example enterprise bean) or the service access mechanism (for example Java Message Service (JMS)).

About this task

Using WSIF, you can move away from the usual Web services programming model of working directly with the SOAP APIs, towards a model where you interact with representations of the services. You can therefore work with the same programming model regardless of how the service is implemented and accessed.

To learn about WSIF, see the following topics:

- "Goals of WSIF."
- "WSIF Overview" on page 842.
  1. "WSIF architecture" on page 843.
  2. "WSIF and WSDL" on page 843.
  3. "WSIF usage scenarios" on page 844.

Goals of WSIF

WSIF aims to extend the flexibility provided by SOAP services into a general model for invoking Web services, irrespective of the underlying binding or access protocols.

SOAP bindings for Web services are part of the WSDL specification, therefore when most developers think of using a Web service, they immediately think of assembling a SOAP message and sending it across the network to the service endpoint, using a SOAP client API. For example: using Apache SOAP the client creates and populates a Call object that encapsulates the service endpoint, the identification of the SOAP operation to invoke, the parameters to send, and so on.

While this process works for SOAP, it is limited in its use as a general model for invoking Web services for the following reasons:

- "Web services are more than just SOAP services" on page 841.
- "Tying client code to a particular protocol implementation is restricting" on page 841.

- “Incorporating new bindings into client code is hard.”
- “Multiple bindings can be used in flexible ways.”
- “A freer Web services environment enables intermediaries” on page 842.

The goals of the Web Services Invocation Framework (WSIF) are therefore:
- To give a binding-independent mechanism for Web service invocation.
- To free client code from the complexities of any particular protocol used to access a Web service.
- To enable dynamic selection between multiple bindings to a Web service.
- To help the development of Web service intermediaries.

Web services are more than just SOAP services

You can deploy as a Web service any application that has a WSDL-based description of its functional aspects and access protocols. If you are using the Java Platform, Enterprise Edition (Java EE) environment, then the application is available over multiple transports and protocols.

For example, you can take a database-stored procedure, expose it as a stateless session bean, then deploy it into a SOAP router as a SOAP service. At each stage, the fundamental service is the same. All that changes is the access mechanism: from Java DataBase Connectivity (JDBC) to Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) and then to SOAP.

The WSDL specification defines a SOAP binding for Web services, but you can add binding extensions to the WSDL so that, for example, you can offer an enterprise bean as a Web service using RMI-IIOP as the access protocol. You can even treat a single Java class as a Web service, with in-thread Java method invocations as the access protocol. With this broader definition of a Web service, you need a binding-independent mechanism for service invocation.

Tying client code to a particular protocol implementation is restricting

If your client code is tightly bound to a client library for a particular protocol implementation, it can become hard to maintain.

For example, if you move from Apache SOAP to Java Message Service (JMS) or enterprise bean, the process can take a lot of time and effort. To avoid these problems, you need a protocol implementation-independent mechanism for service invocation.

Incorporating new bindings into client code is hard

If you want to make an application that uses a custom protocol work as a Web service, you can add extensibility elements to WSDL to define the new bindings. But in practice, achieving this capability is hard.

For example you have to design the client APIs to use this protocol. If your application uses just the abstract interface of the Web service, you have to write tools to generate the stubs that enable an abstraction layer. These tasks can take a lot of time and effort. What you need is a service invocation mechanism that allows you to update existing bindings, and to add new bindings.

Multiple bindings can be used in flexible ways

To take advantage of Web services that offer multiple bindings, you need a service invocation mechanism that can switch between the available service bindings at run time, without having to generate or recompile a stub.

Imagine that you have successfully deployed an application that uses a Web service which offers multiple bindings. For example, imagine that you have a SOAP binding for the service and a local Java binding that lets you treat the local service implementation (a Java class) as a Web service.

The local Java binding for the service can only be used if the client is deployed in the same environment as the service. In this case, it is more efficient to communicate with the service by making direct Java calls than by using the SOAP binding.

If your clients could switch the actual binding used based on run-time information, they could choose the most efficient available binding for each situation.

A freer Web services environment enables intermediaries

Web services offer application integrators a loosely-coupled paradigm. In such environments, intermediaries can be very powerful.

Intermediaries are applications that intercept the messages that flow between a service requester and a target Web service, and perform some mediating task (for example logging, high-availability or transformation) before passing on the message. The Web Services Invocation Framework (WSIF) is designed to make building intermediaries both possible and simple. Using WSIF, intermediaries can add value to the service invocation without needing transport-specific programming.

WSIF Overview

The Web Services Invocation Framework (WSIF) provides a Java API for invoking Web services, independent of the format of the service or the transport protocol through which it is invoked.

This framework addresses all of the issues identified in “Goals of WSIF” on page 840.

WSIF provides the following features:
- An API that provides binding-independent access to any Web service.
- A close relationship with WSDL, so it can invoke any service that you can describe in WSDL.
- A stubless and completely dynamic invocation of a Web service.
- The capability to plug a new or updated implementation of a binding into WSIF at run time.
- The option to defer the choice of a binding until run time.

WSIF provides runtime support for Web services, and for WSDL extensions and bindings, that were not known at build time. This capability is known as *dynamic invocation*. Using WSIF, a client application can choose dynamically the optimal binding to use for invoking Web service operations. For example, a Web service might offer a SOAP binding, and also a local Java binding so that you can treat the local service implementation (a Java class) as a Web service. If a client application is deployed in the same environment as the service, then this client can use the local Java binding for the service. This provides more efficient communication between the client and the service by making direct Java calls rather than indirect calls using the SOAP binding. WSIF provides this runtime support through the use of providers. The providers support Web services, WSDL extensions and bindings that were not known at build time by using the WSDL description to access the target service.

WSIF is designed to work both in an unmanaged environment (stand-alone) and inside a managed container. You can use the Java Naming and Directory Interface (JNDI) to find the WSIF service, or you can use the location described in the WSDL.

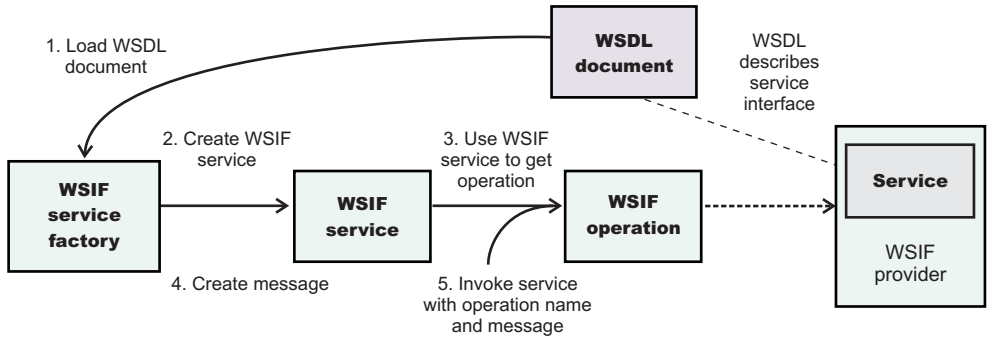
For more conceptual information about WSIF and WSDL, see the following topics:
- WSIF and WSDL
- WSIF architecture
- WSIF usage scenarios

WSIF supports Internet Protocol Version 6, and Java API for XML-based Remote Procedure Calls (JAX-RPC) Version 1.1 for SOAP.

WSIF architecture

A diagram depicting the Web Services Invocation Framework (WSIF) architecture, and a description of each of the major components of the architecture.

The Web Services Invocation Framework (WSIF) architecture is shown in the figure.



The components of this architecture include:

WSDL document
: The Web service WSDL document contains the location of the Web service. The binding document defines the protocol and format for operations and messages defined by a particular portType.

WSIF service
: The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation. For more information, see the “WSIFService interface” on page 876 topic.

WSIF operation
: The run-time representation of an operation, called *WSIFOperation* is responsible for invoking a service based on a particular binding. For more information, see the “WSIFOperation interface” on page 877 topic.

WSIF provider
: A WSIF provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol. WSIF includes SOAP providers, JMS providers, Java providers and EJB providers. For more information, see Linking a WSIF service to the underlying implementation of the service.

WSIF and WSDL

There is a close relationship between the metadata-based Web Services Invocation Framework (WSIF) and the evolving semantics of Web Services Description Language (WSDL).

In WSDL, a service is defined in three distinct sections:

- The **portType**. This section defines the abstract interface offered by the service. A portType defines a set of *operations*. Each operation can be In-Out (request-response), In-Only, Out-Only and Out-In (Solicit-Response). Each operation defines the input and/or output *messages*. A message is defined as a set of *parts*, and each part has a schema-defined type.
- The **binding**. This section defines how to map between the abstract portType and a real service format and protocol. For example the SOAP binding defines the encoding style, the SOAPAction header, the namespace of the body (the targetURI), and so on.
- The **port**. This section defines the actual location (endpoint) of the available service. For example, the HTTP Web address at which a SOAP service is available.

Currently in WSDL, each port has one and only one binding, and each binding has a single portType. But (more importantly) each service (portType) can have multiple ports, each of which represents an alternative location and binding for accessing that service.

The Web Services Invocation Framework (WSIF) follows the semantics of WSDL as much as possible:
- The WSIF dynamic invocation API directly exposes run-time equivalents of the model from WSDL. For example, invocation of an operation involves executing an operation with an input message.
- WSDL has extension points that support the addition of new ports and bindings. This enables WSDL to describe new systems. The equivalent concept in WSIF is a provider, that enables WSIF to understand a class of extensions and thereby to support a new service implementation type.

As a metadata-based invocation framework, WSIF follows the design of the metadata. As WSDL is extended, WSIF is updated to follow.

The primary type system of WSIF is XML schema. WSIF supports invocation using dynamic proxies, which in turn support Java type systems, but when you use the WSIFMessage interface it is your responsibility to populate WSIFMessage objects with data based on the XML schema types as defined in the WSDL document. You should define your object types by a canonical and fixed mapping from schema types into the run-time environment.

WSIF usage scenarios

This topic describes two brief scenarios that illustrate the role WSIF plays in the emerging Web services environment.

Scenario: Redevelopment and redeployment

When you first implement a Web service, you create a simple prototype. When you want to move a prototype Web service into production, you often need to redevelop and redeploy it.

The Web Services Invocation Framework (WSIF) uses the same API calls irrespective of the underlying technologies, therefore if you use WSIF:
- You can reimplement and redeploy your services without changing the client code.
- You can use existing reliable and high-performance infrastructures like Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) and Java Message Service (JMS) without sacrificing the location-independence that the Web service model offers.

Scenario: Service Flow composition

A service flow typically invokes a Web service, then passes the response from one Web service to the next Web service, perhaps performing some transformation in the middle.

There are two key aspects to this flow that WSIF provides:
- A representation of the service invocation based on the metadata in WSDL.
- The ability to build invocations based solely on the portType, which can therefore be used in any implementation.

For example, imagine that you build a meta-service that uses a number of services to build a process. Initially, several of those services are simple Java bean prototypes that are written and exposed through SOAP, but you plan to reimplement some of them as EJB components, and to out-source others.

If you use SOAP, it ties up multiple threads for every onward invocation, because they pass through the Web server and servlet engine and on to the SOAP router. If you use WSIF to call the beans directly, you get much better performance compared to SOAP and you do not lose access or location transparency. Using WSIF, you can replace the Java bean implementations with EJB implementations without changing the client code. To move some of the Web services from local implementations to external SOAP services, you just update the WSDL.

Using WSIF to invoke Web services

You invoke a Web service dynamically by using the WSIF API directly.

About this task

You specify the location of the WSDL file for the service, the name of the operation to invoke, and any operation arguments. All other information needed to access the Web service (the abstract interface, the binding, and the service endpoint) is available through the WSDL.

This kind of invocation does not require stub classes and does not need a separate compilation cycle.

More information on using the Web Services Invocation Framework (WSIF) to invoke Web services is provided in the following topics:

- "Linking a WSIF service to the underlying implementation of the service."
- "Developing a WSIF service" on page 860.
- "Using complex types" on page 869.
- "Using WSIF to bind a JNDI reference to a Web service" on page 870.
- "Example: Passing SOAP messages with attachments using WSIF" on page 871.
- "Interacting with the Java EE container in WebSphere Application Server" on page 874.
- "Running WSIF as a client" on page 874.

Linking a WSIF service to the underlying implementation of the service

A Web Services Invocation Framework (WSIF) service is linked to the underlying service through a WSIF provider. A provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol. Providers implement the interface between the WSIF API and the actual implementation of a service.

About this task

Providers are pluggable within the WSIF framework, and are registered according to the namespace of the WSDL extension that they implement. Some providers use the Java Platform, Enterprise Edition (Java EE) programming model to utilize Java EE services. If a provider is available, but its required class libraries are not, then the provider is disabled.

To use the providers that are supplied with WebSphere Application Server, see the following topics:

- Link a WSIF service to a SOAP over HTTP service.
- Link a WSIF service to a JMS-provided service (SOAP over JMS, or native JMS).
- Link a WSIF service to a local Java application.
- Link a WSIF service to a service implemented as an enterprise bean.

Linking a WSIF service to a SOAP over HTTP service

The SOAP provider allows WSIF stubs and dynamic clients to invoke SOAP services. Add WSDL extensions to your Web service WSDL file so that the service can use the SOAP provider.

About this task

The Web Services Invocation Framework (WSIF) SOAP provider supports SOAP 1.1 over HTTP.

The SOAP provider is JSR 101/109 compliant and uses Web Services for Java EE for parsing and creating SOAP messages.

Note: The current WSIF default SOAP provider (the IBM Web Service SOAP provider) does not fully interoperate with services that are running on the former (Apache SOAP) provider. This restriction is due to the fact that the IBM Web Service SOAP provider is designed to interoperate fully with a JAX-RPC compliant Web service, and Apache SOAP cannot provide such a service. For more information see "WSIF SOAP provider: working with legacy applications."

The SOAP provider supports:
- SOAP-ENC encoding.
- RPC style and Document style SOAP messages.
- SOAP messages with attachments.

The SOAP provider is not transactional.

The SOAP provider does not support the WSIF synchronous timeout. The SOAP provider uses the default client timeout value that is set for Web Services for Java EE.

If you have a Web service that you expect multiple clients to use to connect over SOAP, then before you deploy the service you must set up your application deployment descriptor file `dds.xml` to handle multiple connections correctly. For more information, see WSIF troubleshooting tips.

To link a WSIF service to a SOAP over HTTP service, extend the service WSDL file in accordance with the code examples given in the following topics:
- "Example: Writing the WSDL extension that enables your WSIF service to access a SOAP over JMS service" on page 848.

  Note: The WSDL binding extension for SOAP over JMS varies only slightly from the SOAP over HTTP binding.
- "Example: Writing the WSDL extensions for SOAP attachments" on page 872.

WSIF SOAP provider: working with legacy applications:

The current WSIF default SOAP provider (the IBM Web Service SOAP provider) does not fully interoperate with services that are designed to run on the former (Apache SOAP) provider. This is due to the fact that the IBM Web Service SOAP provider is designed to interoperate fully with a JAX-RPC compliant Web service, and Apache SOAP cannot provide such a service.

About this task

As a result of this change in SOAP providers, previous WSIF clients might not work in either of the following cases:
1. The Web service uses any of the following parameter types: `xsd:date`, `xsd:dateTime`, `xsd:hexBinary` or `xsd:QName` (for more information, see the **Type Mappings** section of WSIF - Known restrictions).
2. The Web service was built upon the former (Apache SOAP) provider.

To get your legacy services working again, you have two options:
- Change the default WSIF SOAP provider back to the former Apache SOAP provider (in which case any future invocations to a JAX-RPC compliant Web service will not work if that Web service uses parameter types `xsd:date`, `xsd:dateTime`, `xsd:hexBinary` or `xsd:QName`).
- Modify your services to use the current IBM Web Service SOAP provider.

Changing the default WSIF SOAP provider:

The WSIF default SOAP provider (the IBM Web Service SOAP provider) is designed to interoperate fully with a JAX-RPC compliant Web service, and therefore the default provider does not fully interoperate with services that are running on the former (Apache SOAP) provider. To get your legacy services working

again, you can either modify your Web services to use the current IBM Web Service SOAP provider, or you can change the WSIF default provider back to Apache SOAP as described in this topic.

About this task

WSIF uses a properties file named `wsif.properties` to choose what SOAP provider to use. The SOAP provider is a node-wide setting, so all servers on the node must be restarted for any changes to take effect. The `wsif.properties` file is shipped in the `com.ibm.ws.runtime.jar` file that is located in the *app_server_root*`/plugins` directory (where *app_server_root* is the root directory for your installation of IBM WebSphere Application Server), and the "as shipped" properties file is accessed in this location by being put on the class path. However when you make changes to the file, you do not replace the original copy in the `com.ibm.ws.runtime.jar` file. Instead, you save the modified version in the *app_server_root*`/lib/properties` directory.

To change the WSIF default SOAP provider back to Apache SOAP, complete the following steps:

1. Extract the `wsif.properties` file from the `com.ibm.ws.runtime.jar` file that is located in the *app_server_root*`/plugins` directory (where *app_server_root* is the root directory for your installation of IBM WebSphere Application Server).
2. Open the `wsif.properties` file in a text editor.
3. Remove the leading "#" character from the following lines:

```
# wsif.provider.default.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=1
# wsif.provider.uri.1.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=\
# http://schemas.xmlsoap.org/wsdl/soap/
#
```

   After the update, the preceding lines should look like this:

```
wsif.provider.default.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=1
wsif.provider.uri.1.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=\
http://schemas.xmlsoap.org/wsdl/soap/
#
```

4. Save the updated `wsif.properties` file in the *app_server_root*`/lib/properties` directory.
5. Stop then restart all application servers on the node.

Example

Modifying Web services to use the IBM Web Service SOAP provider:

The current WSIF default SOAP provider (the IBM Web Service SOAP provider) is designed to interoperate fully with a JAX-RPC compliant Web service, and therefore the current default provider does not fully interoperate with services that are running on the former (Apache SOAP) provider. To get your legacy services working again, you can either modify your Web services to use the current IBM Web Service SOAP provider as described in this topic, or you can change the WSIF default provider back to Apache SOAP.

About this task

To modify a legacy Web service, use the assembly tool to complete the following steps and thereby generate new deployment artifacts for access to the service from the IBM Web Service provider:

1. Import into the Workspace the project that contains your legacy Web services.
2. For every legacy SOAP service in the project, repeat the following steps :
   a. From the pop-up menu for *your_service*`.wsdl`, select **Generate Deploy Code**.
   b. In the Generate Deploy Code window, change the **Inbound Binding Type** from `SOAP` to `IBM Web Service`.
   c. Click **Finish**.

3. Export the EAR file that contains all of the deployment artifacts for the IBM Web Service Web service.

Linking a WSIF service to a JMS-provided service

The JMS providers enable a WSIF service to be invoked through either SOAP over JMS, or native JMS. Add WSDL extensions to your Web service WSDL file so that the service can use the JMS providers.

About this task

The Java Message Service (JMS) is an API for transport technology. The mapping to a JMS destination is defined during deployment and maintained by the container.

The JMS destination endpoint for a Web service can be realized in any of the following ways:
- The JMS destination for the queue can be the Web service implementation.
- The JMS destination can be (but is not required to be) associated with a message-driven bean by the EJB container, thereby allowing the message-driven bean to be the Web service implementation.
- For SOAP over JMS, the JMS destination can unwrap the JMS message and route the SOAP message to a Web service that is implemented as a stateless session bean.

The JMS destination endpoint must respect the interaction model expected by the client and defined by the WSDL. It must return a response if one is required.

When the JMS destination endpoint creates the JMS response message the following rules must be followed:
- The response message must be sent to `JMSReplyTo` from the incoming request.
- The `JMSCorrelationID` value of the response message must be set to the `JMSMessageID` value from the request message.
- The response must be sent with a `deliveryMode` value equal to the `JMSDeliveryMode` value of the request message.
- The response must be sent with a `priority` value equal to the `JMSPriority` value of the request message.
- The `TimeToLive/JMSExpiration` value must be set to a value that equals the `JMSExpiration` value of the request message.

The client does not see any of these headers. The container receives the JMS message and (for SOAP over JMS) removes the SOAP message to send to the client.

To link a WSIF service to a JMS-provided service, use the information and code examples given in the following topics:
- Link a WSIF service to a SOAP over JMS service..
- Link a WSIF service to a service provided at a JMS destination..
- Configure the client and server so that a service can be invoked through JMS by a WSIF client application.

Example: Writing the WSDL extension that enables your WSIF service to access a SOAP over JMS service:

If a SOAP message contains only XML, it can be carried on the Java Message Service (JMS) transport mechanism with the JMS message body type **TextMessage**. Use this information, and associated code fragments, to help you to write the WSDL extension that enables your WSIF service to access a SOAP service that uses the Java Message Service (JMS) as its transport mechanism.

The SOAP message, including the SOAP envelope, is wrapped with a JMS message and put on the appropriate queue. The container receives the JMS message and removes the SOAP message to send to the client.

For information about working with the Java Message Service (JMS) API, see Linking a WSIF service to a JMS-provided service.

Note: You can also use this topic as a guide to writing the WSDL binding extension for SOAP over HTTP, because the SOAP over JMS binding is almost identical to the SOAP over HTTP binding.

Selecting the SOAP over JMS binding

You set the `transport` attribute of the `<soap:binding>` tag to indicate that JMS is used. If you also set the `style` attribute to `rpc` (Remote Procedure Call), then the Web Services Invocation Framework (WSIF) assumes that an operation is invoked on the Web service endpoint:

```
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/jms"/>
```

Setting the JMS address

Note: See also the alternative method for specifying the JMS address that is given in the final section of this topic.

For SOAP over JMS, the `<wsdl:port>` tag must contain a `<jms:address>` element. This element provides the information required for a client to connect correctly to the Web service using the JMS programming model. Typically, it is the stubs generated to support the SOAP over JMS binding that act as the JMS client. Alternatively, the Web service client can use the JMS programming model directly.

The `<jms:address>` element takes this form:

```
<jms:address

    destinationStyle="queue"
    jmsVendorURI="http://ibm.com/ns/mqseries"?
    initialContextFactory="com.ibm.NamingFactory"?
    jndiProviderURL="iiop://something:900/wherever"?
    jndiConnectionFactoryName="orange"
    jndiDestinationName="fred">

    <jms:propertyValue name="targetService" type="xsd:string"
                       value="StockQuoteServicePort"/>

</jms:address>
```

where attributes marked with a question mark (?) are optional.

The optional `jmsVendorURI` attribute is a string that uniquely identifies the JMS implementation. WSIF ignores this URI, which is used by the client developer and perhaps the client implementation to determine if it has access to the correct JMS provider in the client run-time environment.

The optional attributes `initialContextFactory` and `jndiProviderURL` can only be omitted if the run-time environment has a default Java Naming and Directory Interface (JNDI) provider configured.

The `jndiConnectionFactoryName` attribute gives the name of a JMS `ConnectionFactory` object, which can be looked up within the JNDI context given by the `jndiContext` attribute. This `ConnectionFactory` object is used to create a JMS connection to the JMS provider instance that owns the queue. In a simple configuration, the same `ConnectionFactory` object is used by the server message listener and by the clients. However the server and the clients can use different `ConnectionFactory` objects, provided that they all create connections to the same JMS provider instance.

The `value` attribute of the `targetService` `<jms:propertyValue>` element is the name of the port component for the target service as defined in the `<port-component-name>` element of the `webservices.xml` file for the target service.

Setting the JMS headers and properties

You use the <jms:property> tag to set the JMS headers and properties. This tag maps either a message part, or a literal value, into a JMS property:

```
<jms:property name="Priority" {part="requestPriority" | value="fixedValue"}/>
```

If the <jms:property> has a literal value, then it can also be nested within the <jms:address> tag:

```
<jms:property name="Priority" value="fixedValue" />
```

This form of the <jms:property> tag is also used in the native JMS binding.

Example of a WSDL that defines a SOAP over JMS binding

```
<!-- Example: SOAP over JMS Text Message -->

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        name="StockQuoteInterfaceDefinitions"
        targetNamespace="urn:StockQuoteInterface"
        xmlns:tns="urn:StockQuoteInterface"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:jms="http://schemas.xmlsoap.org/wsdl/jms/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <wsdl:message name="GetQuoteInput">
        <part name="symbol" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="GetQuoteOutput">
        <part name="value" type="xsd:float"/>
    </wsdl:message>

    <wsdl:portType name="StockQuoteInterface">
        <wsdl:operation name="GetQuote">
            <wsdl:input message="tns:GetQuoteInput"/>
            <wsdl:output message="tns:GetQuoteOutput"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="StockQuoteSoapJMSBinding" type="tns:StockQuoteInterface">
        <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/jms"/>
        <wsdl:operation name="GetQuote">
            <soap:operation soapAction="urn:StockQuoteInterface#GetQuote"/>
            <wsdl:input>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuoteServicePort"
                    binding="sqi:StockQuoteSoapJMSBinding">
            <jms:address destinationStyle="queue"
                     jndiConnectionFactoryName="myQCF"
                     jndiDestinationName="myQ"
                     initialContextFactory="com.ibm.NamingFactory"
                     jndiProviderURL="iiop://something:900/">

                     <jms:propertyValue name="targetService"
                                        type="xsd:string"
                                        value="StockQuoteServicePort"/>
```

```
            </jms:address>

        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Setting the JMS address (alternative method)

For the SOAP over JMS provider you can instead specify the JMS address using the <soap:address> tag in the following format:

`jms:/[queue|topic]?<property>=<value>&<property>=<value>&...`

where the specification of *queue* or *topic* corresponds to the JMS address `destinationStyle` attribute.

The following table lists the valid properties for use with the <soap:address> tag:

Property name	Property description	Corresponding JMS address value
destination	The JNDI name of the destination queue or topic	jndiDestinationName
connectionFactory	The JNDI name of the connection factory.	jndiConnectionFactory
targetService	The name of the port component of the target service	targetService jms:propertyValue within jms:address
JNDI-related properties (optional):		
initialContextFactory	The name of the initial context factory.	initialContextFactory
jndiProviderURL	The JNDI provider URL	jndiProviderURL
JMS-related properties (optional):		
deliveryMode	An indication as to whether the request message should be persistent or not. The valid values are DeliveryMode.NON_PERSISTENT (default) and DeliveryMode.PERSISTENT	JMSDeliveryMode
password	The password to be used to gain access to the connection factory.	JMSPassword
priority	The JMS priority associated with the request message. Valid values are 0 to 9. The default value is 4.	JMSDeliveryMode
replyTo	The JNDI destination queue to which reply messages should be sent.	JMSReplyTo
timeToLive	The lifetime (in milliseconds) of the request message. A value of 0 indicates an infinite lifetime.	JMSTimeToLive
userid	The userid to be used to gain access to the connection factory.	JMSUserid

Here is an example of this format:

<jms:address> format:

```
<wsdl:port name="StockQuoteServicePort"

        binding="sqi:StockQuoteSoapJMSBinding">
```

```
    <jms:address destinationStyle="queue"
        jndiConnectionFactoryName="myQCF"
        jndiDestinationName="myQ"
        initialContextFactory="com.ibm.NamingFactory"
        jndiProviderURL="iiop://something:900/">

        <jms:propertyValue name="targetService"
                           type="xsd:string"
                           value="StockQuoteServicePort"/>

    </jms:address>

</wsdl:port>
```

<soap:address> format:

```
<wsdl:port name="StockQuoteServicePort"
        binding="sqi:StockQuoteSoapJMSBinding">
    <soap:address location="jms:/queue?connectionFactory=myQCF&destination
=myQ&initialContextFactory=com.ibm.NamingFactory&jndiProviderURL
=iiop://something:900/&targetService=StockQuoteServicePort" />

</wsdl:port>
```

Example: Writing the WSDL extensions that enable your WSIF service to access an underlying service at a JMS destination:

Using the native JMS provider, WSIF clients can treat a service that is available at a Java Message Service (JMS) destination as a Web service. Use this information, and associated code fragments, to help you to write the WSDL extensions.

For information about working with the Java Message Service (JMS) API, see Linking a WSIF service to a JMS-provided service.

The WSDL extensions for JMS are identified with the namespace prefix `jms`. For example, `<jms:binding>`.

Operations

The supported operations are either one-way operations (`send` for JMS point-to-point messaging, or `publish` for JMS publish and subscribe messaging) or request-response operations (`send` and `receive` for JMS point-to-point messaging). The WSDL operations therefore specify either an input message only, or an input and an output message.

Fault messages

Operations that describe message interfaces with a native JMS binding do not have fault messages. No assumptions are made about the message schema or the semantics of message properties, therefore no distinction can be made between output and fault messages.

Setting the JMS message body type

You use the `<jms:binding>` extension to specify the JMS message body type:

```
<wsdl:binding ... >
   <jms:binding type="messageBodyType" />
    ...
</wsdl:binding>
```

where *messageBodyType* is either `ObjectMessage` or `TextMessage`.

Specifying the parts to use for the input and output messages

For JMS text messages and JMS object messages created from one or more WSDL message parts, you use the <jms:input> and <jms:output> extensions to specify the message parts to use for the JMS messages:

```
<wsdl:input ... >
   <jms:input parts="part1 part2 ..." />
</wsdl:input>

<wsdl:output ... >
   <jms:output parts="part1 part2 ..." />
</wsdl:output>
```

In the next example, the WSDL message has just one part that contains the complete message body. This message body might result from a mapping of some other representation (see **Mapping data types**).

```
<wsdl:input ... >
   <jms:input parts="part1" />
</wsdl:input>
```

If no parts are defined, then all the message parts are used.

Mapping data types

You use the <format> extensions to map data types:

```
<wsdl:binding ... >
   <jms:binding type="..." />

   <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="..." formatType="targetType"/>
   </format:typemapping>
   ...
</wsdl:binding>
```

The value of *targetType* is dependent on the JMS message body type (see **Setting the JMS message body type**). For JMS object messages, the target data type implements the java.io.Serializable class. For JMS text messages, the target data type is always java.lang.String.

The <format> extensions are also used in other bindings that deal with Java interfaces.

Setting the JMS headers and properties

JMS does not make assumptions about message headers. For example, if the JMS provider is MQSeries then each JMS message carries an RFH2 header. However you can access data in this message header indirectly, by getting and setting JMS message properties.

When you want your application to pass a property into the Web Services Invocation Framework (WSIF) as a part on the WSIF message, you use a <jms:property> tag. When you want to hard code an actual property value into the WSDL, you use a <jms:propertyValue> tag. The <jms:propertyValue> tag contains a specification of a literal value and its associated XML schema type.

You can specify <jms:property> and <jms:propertyValue> extensions within the <wsdl:input> tag in the binding operation, and also within the <jms:address> tag. For the<wsdl:output> tag in the binding operation, you can only specify the <jms:property> extension. Property values that are set in the <jms:property> tag take precedence over values set in the <<jms:propertyValue> tag, and property values that are set in the binding operation (in the <input> and <output> tags) take precedence over values set in the <jms:address> tag.

Here is an example of the <jms:property> and <jms:propertyValue> tags nested within the <input> and <output> tags:

```
<wsdl:input ... >

    <jms:property name="propertyName" part="partName" />
```

```
    <jms:propertyValue name="propertyName"
              type="xsdType" value="actualValue" />

</wsdl:input>

<wsdl:output ... >

    <jms:property name="propertyName" part="partName" />

</wsdl:output>
```

where *propertyName* identifies the JMS property that is associated with the header field, and *partName* identifies the message part that is associated with the property.

The JMS property identified by *propertyName* can be user-defined, or it can be one of the following predefined JMS message header fields:

Value	Java type
JMSMessageId	java.lang.String
JMSTimeStamp	long
JMSCorrelationId	byte [ ] or java.lang.String
JMSReplyTo	javax.jms.Destination
JMSDestination	javax.jms.Destination
JMSDeliveryMode	int
JMSRedelivered	boolean
JMSType	java.lang.String
JMSExpiration	long
JMSTimeToLive	long

See the JMS specification for restrictions that apply when setting JMS header field values. Attempts to set restricted values are ignored.

For application-defined JMS message properties, the Java types used in the native JMS binding implementation (used for calls to the corresponding JMS methods) are derived from the XML schema type in the abstract interface (`<wsdl:part>` tag), and from the type mapping information in the format binding (`<format:typemap>` tag).

Handling transactions

Independent of other JMS properties, the asynchronous processing of request-response operations has implications for callers running in a transaction scope. The send request part and the receive response part are separated into two transactions, because the send needs to be committed in order for the request message to become visible. Implementations that process WSDL for asynchronous request-response operations (such as WSIF) must therefore take the following additional actions:

- They must ensure that the send request transaction returns a correlation ID to the user, and provides a **callback** with which users can pass in the response message to process the receive response transaction.
- They might implement their own response message “listener” in order to recognize the arrival of response messages, and to manage the correlation to the request message.

Example 1: JMS Text Message

The JMS text message contains a **java.lang.String**. In this example, the WSDL message contains only one part that represents the whole message body:

```
<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest"> ... </wsdl:message>
   <wsdl:message name="JmsOperationResponse"> ... </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <jms:binding type="TextMessage" />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:String" formatType="String" />
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:input parts="requestMessageBody" />
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:output parts="responseMessageBody" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

As an extension to the previous JMS message example, the following example WSDL describes a request-response operation in which specific JMS property values of the request and response message are set for the request message and retrieved from the response message.

The JMS properties in the request message are set according to the values in the input message. Likewise, selected JMS properties of the response message are copied to the corresponding values of the output message. The direction of the mapping is determined by the appearance of the `<jms:property>` tag in the input or output section, respectively.

Example 2: JMS Message with JMS Properties

```
<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest">
      <wsdl:part name="myInt" type="xsd:int"/>
      ...
   </wsdl:message>

   <wsdl:message name="JmsOperationResponse">
```

```
      <wsdl:part name="myString" type="xsd:String"/>
      ...
   </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <!-- the JMS message type may be any of the above -->
      <jms:binding type="..." />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:int" formatType="int" />
         ...
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:property message="tns:JmsOperationRequest" parts="myInt" />
            <jms:propertyValue name="myLiteralString"
                          type="xsd:string" value="Hello World" />
            ...
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:property message="tns:JmsOperationResponse" parts="myString" />
            ...
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
    </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

Configuring the client and server so that a service can be invoked through JMS by a WSIF client application:

The ways in which the Web Services Invocation Framework (WSIF) interacts with the Java Message Service (JMS), and the steps you need to take to enable a service to be invoked through JMS by a WSIF client application.

Before you begin

This topic assumes that you chose and configured a JMS provider when you installed WebSphere Application Server (either the JMS provider that is embedded in WebSphere Application Server, or another provider such as WebSphere MQ). If not, do so now as described in “Choosing a messaging provider” on page 1145.

About this task

Here are the ways in which the Web Services Invocation Framework (WSIF) interacts with JMS:
- Only input JMS properties are supported.
- WSIF needs two queues when invoking an operation: one for the request message and one for the reply. The replyTo queue is by default a temporary queue, which WSIF creates on behalf of the application. You can specify a permanent queue by setting the JMSReplyTo property to the JNDI name of a queue.
- WSIF uses the default values for properties set by the JMS implementation. However in MQSeries and in some other JMS implementations, messages are persistent by default, and the default temporary queue is of type *temporary dynamic* and cannot have persistent messages written to it. Therefore your JMS listener can fail to write a persistent response message to the temporary replyTo queue.

  Note:
  - If you are using MQSeries, you need to create a temporary model queue that is of type *permanent dynamic*, then pass this model as the *tempmodel* of your queue connection factory. This will ensure that persistent messages are written to a temporary replyTo queue that is of type *permanent dynamic*.
  - If your client is running on an application server that is migrated from WebSphere Application Server Version 5 to Version 6, you might get basic authentication errors and therefore need to modify your security settings. For more information see Web Services Invocation Framework troubleshooting tips.

To enable a service to be invoked through JMS by a WSIF client application, complete the following steps:
1. Use the administrative console to create and configure a queue connection factory and a queue destination as described in Configuring resources for the default messaging provider or Configuring JMS resources for a generic messaging provider.
2. Use the administrative console to add the new queue destination to the list of JMS Server destination names for your application server. Ensure that the Initial State is started.
3. Put the JNDI names of the queue destination and queue connection factory, as well as your JNDI configuration, in the WSDL file.

JMS message header: The TimeToLive property reference:

The range of permitted values for the TimeToLive property of a JMS message that WSIF puts onto a queue.

The JMS message header property `JMSTimeToLive` is of type `long`. It sets the time to live of a message put onto a queue, in milliseconds. A value of `0` means live indefinitely.

The factors that determine the time to live of a JMS message are as follows:
- For a one-way (input only) operation, the default time to live is `0`, so the message remains on the queue indefinitely or until the server end-processes the message. If the `JMSTimeToLive` property is specified in the service endpoint URL or the JMS Address, then this value is used for one-way messages. The client never waits for a response to a one-way operation and so it never times out. The only time a client for a one-way operation will fail is if the queue itself is unavailable.
- For a two-way (request and response) operation, the default time to live is determined by the client response timeout setting. The time to live for the message is never greater than the client response timeout, even if a larger value is specified in the `JMSTimeToLive` property of the service endpoint URL or the JMS Address, so the message will never be read from the queue after the client has timed out waiting for a response. The client response timeout setting that is used as the default time to live is the WSIF synchronous timeout. This is the case even for an asynchronous JMS message.

Example: Writing the WSDL extension that enables your WSIF service to invoke a method on a local Java object

Using the WSIF Java provider, WSIF can invoke Java code. This means that, in a thin-client environment such as a Java virtual machine (JVM) or Tomcat test run-time environment, you can define shortcuts to local Java programs. Use this information, and associated code fragments, to help you to write the WSDL extension that links your WSIF service to a local Java application.

The Web Services Invocation Framework (WSIF) Java provider is not intended for use in a Java Platform, Enterprise Edition (Java EE) environment. There is a difference between a client using the WSIF Java provider to invoke a Java component, and implementing a Web service as a Java component on the server side.

The Java binding exploits the format binding for type mapping. Using the format binding, your WSDL can define the mapping between XML schema types and Java types.

The Java provider requires that the targeted Java classes reside in the class path of the client. The Java method is invoked synchronously, in-process, in-thread, with the current thread and Object Request Broker (ORB) contexts.

The Java provider is not transactional.

The Java provider does not support the WSIF synchronous timeout. The Java provider will not time out waiting for a Java method to complete.

To use the Java provider, you need the following binding specified in the WSDL file:

```
<!-- Java binding -->
<binding .... >
    <java:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
    <operation>*
        <java:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            methodType="instance|constructor" />
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
```

In this example:

- A question mark (?) means optional, and an asterisk (*) means 0 or more.
- The `name` attribute of the `<format:typeMap>` element is a qualified name of a simple or complex type used by one of the Java operations.
- The `formatType` attribute of the `<format:typeMap>` element is the fully qualified class name for the Java class to which the element specified by `name` maps.
- The `methodName` attribute of the `<java:operation>` element is the name of the method on the Java object that is called by the operation.
- The `parameterOrder` attribute of the `<java:operation>` element contains a white space-separated list of part names that define the order in which they are passed to the Java object method.
- The `methodType` attribute of the `<java:operation>` element must be set to either `instance` or `constructor`. The value specifies whether the method that is invoked on the object is an instance method or a constructor for the object.

In the next example, the `className` attribute of the `<java:address>` element specifies the fully qualified class name of the object containing the method to invoke:

```
<service ... >
    <port>*
        <java:address
            className="nmtoken"/>
    </port>
</service>
```

Example: Writing the WSDL extension that enables your WSIF service to invoke an enterprise bean

Using the EJB provider, WSIF clients can invoke enterprise beans through Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). Use this information, and associated code fragments, to help you to write the WSDL extension that links your WSIF service to a service implemented as an enterprise bean.

Note: Although you can use the EJB provider for EJB(IIOP)-based Web service invocation, it is recommended that you instead invoke RMI-IIOP Web services using JAX-RPC.

The EJB client JAR file must be available in the client run-time environment with the current provider. The enterprise bean is invoked using normal EJB invocation methods, using RMI-IIOP, with the current security and transaction contexts. If the EJB provider is invoked within a transaction, the transaction is passed to the onward service and the standard EJB transaction attribute applies.

If there are multiple implementations of the service, it is up to the service providers to make sure that every implementation offers the same semantics. For example, in the case of transactions, the bean deployer must specify `TX_REQUIRES_NEW` to force a new transaction.

The EJB provider does not support the WSIF synchronous timeout. The EJB provider will not time out waiting for a Java method to complete.

To use the EJB provider, you need the following binding specified in the WSDL file:

```
<!-- EJB binding -->
<binding .... >
    <ejb:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
    <operation>*
        <ejb:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            interface="remote|home" />
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
```

In this example:

- A question mark (`?`) means optional, and an asterisk (`*`) means 0 or more.
- The `name` attribute of the `<format:typeMap>` element is a qualified name of a simple or complex type used by one of the EJB operations.
- The `formatType` attribute of the `<format:typeMap>` element is the fully qualified class name for the Java class to which the element specified by `name` maps.
- The `methodName` attribute of the `<ejb:operation>` element is the name of the method on the enterprise bean that is called by the operation.
- The `parameterOrder` attribute of the `<ejb:operation>` element contains a white space-separated list of part names that define the order in which they are passed to the EJB method.

- The `interface` attribute of the `<ejb:operation>` element must be set to either `remote` or `home`. The value specifies the interface of the enterprise bean on which the method named by the `methodName` attribute is accessible.

In the next example:

- The `className` attribute of the `<ejb:address>` element specifies the fully qualified class name of the home interface class of the enterprise bean.
- The `jndiName` attribute of the `<ejb:address>` element specifies the full Java Naming and Directory Interface (JNDI) name that is used to look up the enterprise bean.
- The `initialContextFactory` attribute of the `<ejb:address>` element is optional and specifies the initial context factory class.
- The `jndiProviderURL` attribute of the `<ejb:address>` element is optional and specifies the JNDI provider Web address.

```
<service ... >
    <port>*
        <ejb:address
            className="nmtoken"
            jndiName="nmtoken"
            initialContextFactory="nmtoken" ?
            jndiProviderURL="nmtoken" ? />
    </port>
</service>
```

Developing a WSIF service

A Web Services Invocation Framework (WSIF) service is a Web service that uses WSIF.

About this task

To develop a WSIF service, develop the Web service (or use an existing Web service), then develop the WSIF client based on the WSDL document for that Web service.

There are also two pre-built WSIF Samples available for download from the Samples Central page of the DeveloperWorks WebSphere Web site:

- The Address Book Sample.
- The Stock Quote Sample.

For more information about using the pre-built Samples, see the documentation that is included in the download package.

To develop a WSIF service, complete the following steps:

1. Implement the Web service.

   Use Web services tools to discover, create, and publish the Web service. You can develop Java bean, enterprise bean, and URL Web services. You can use Web service tools to create skeleton Java code and a sample application from a WSDL document. For example, an enterprise bean can be offered as a Web service, using Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) as the access protocol. Or you can use a Java class as a Web service, with native Java invocations as the access protocol.

   You can use the WebSphere Studio Application Developer to create a Web service from a Java application, as described in its StockQuote service tutorial. The Java application that you use in this scenario returns the last trading price from the Internet Web site www.xmltoday.com, given a stock symbol. Using the Web service wizard, you generate a binding WSDL document named `StockQuoteService-binding.wsdl` and a service WSDL document named `StockQuoteService-service.wsdl` from the `StockQuoteService.java` bean. You then deploy the Web service to a Web server, generate a client proxy to the Web service, and generate a sample application that accesses

the StockQuoteService through the client proxy. You test the StockQuote Web service, publish it using the IBM UDDI Explorer, and then discover the StockQuote Web service in the IBM UDDI Test Registry.

2. Develop the WSIF client. The information you need to develop a WSIF client is provided in the following topics:
   - Example: Using WSIF to invoke the AddressBook Sample Web service dynamically gives example code to show how you define a Web service in WSDL.
   - Linking a WSIF service to the underlying implementation of the service describes the available providers, and gives example code of how their WSDL extensions are coded.
   - "Invoking a WSDL-based Web service through the WSIF API" on page 874 defines the main interfaces that your client uses to support the invocation of Web services defined in WSDL.

   The Address Book Sample is written for synchronous interaction. If you are using a JMS provider, your WSIF client might need to act asynchronously. WSIF provides two main features that meet this requirement:
   - A **correlation service** that assigns identifiers to messages so that the request can match up with the (eventual) response.
   - A **response handler** that picks up the response from the Web service at a later time.

   For more information, see the WSIF API topic WSIFOperation - Asynchronous interactions reference.

Example: Using WSIF to invoke the AddressBook Sample Web service dynamically

The code fragments in this topic show you how to use the Web Services Invocation Framework (WSIF) API to invoke the AddressBook Sample Web service dynamically.

This is example code for dynamic invocation of the AddressBook sample Web service using WSIF:

```
try {
    String wsdlLocation="clients/addressbook/AddressBookSample.wsdl";

    // The starting point for any dynamic invocation using wsif is a
    // WSIFServiceFactory. We create ourselves one via the newInstance
    // method.
    WSIFServiceFactory factory = WSIFServiceFactory.newInstance();

    // Once we have a factory, we can use it to create a WSIFService object
    // corresponding to the AddressBookService service in the wsdl file.
    // Note: since we only have one service defined in the wsdl file, we
    // do not need to use the namespace and name of the service and can pass
    // null instead. This also applies to the port type, although values have
    // been used below for illustrative purposes.
    WSIFService service = factory.getService(
        wsdlLocation,    // location of the wsdl file
        null,            // service namespace
        null,            // service name
       "http://www.ibm.com/namespace/wsif/samples/ab", // port type namespace
        "AddressBookPT" // port type name
    );

    // The AddressBook.wsdl file contains the definitions for two complexType
    // elements within the schema element. We will now map these complexTypes
    // to Java classes. These mappings are used by the Apache SOAP provider
    service.mapType(
        new javax.xml.namespace.QName(
            "http://www.ibm.com/namespace/wsif/samples/ab/types",
            "address"),
        Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"));
    service.mapType(
        new javax.xml.namespace.QName(
            "http://www.ibm.com/namespace/wsif/samples/ab/types",
            "phone"),
        Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFPhone"));
        // We now have a WSIFService object. The next step is to create a WSIFPort
        // object for the port we wish to use. The getPort(String portName) method
        // allows us to generate a WSIFPort from the port name.
```

```
      WSIFPort port = null;

      if (portName != null) {
          port = service.getPort(portName);
      }
      if (port == null) {
          // If no port name was specified, attempt to create a WSIFPort from
          // the available ports for the port type specified on the service
          port = getPortFromAvailablePortNames(service);
      }

      // Once we have a WSIFPort, we can create an operation. We are going to execute
      // the addEntry operation and therefore we attempt to create a WSIFOperation
      // corresponding to it. The addEntry operation is overloaded in the wsdl ie.
// there are two versions of it, each taking different parameters (parts).
// This overloading requires that we specify the input and output message
// names for the operation in the createOperation method so that the correct
// operation can be resolved.
      // Since the addEntry operation has no output message, we use null for its name.
      WSIFOperation operation =
          port.createOperation("addEntry", "AddEntryWholeNameRequest", null);

      // Create messages to use in the execution of the operation. This should
      // be done by invoking the createXXXXXMessage methods on the WSIFOperation.
      WSIFMessage inputMessage = operation.createInputMessage();
      WSIFMessage outputMessage = operation.createOutputMessage();
      WSIFMessage faultMessage = operation.createFaultMessage();

      // Create a name and address to add to the addressbook
      String nameToAdd="Chris P. Bacon";
      WSIFAddress addressToAdd =
          new WSIFAddress (1,
              "The Waterfront",
              "Some City",
              "NY",
              47907,
              new WSIFPhone (765, "494", "4900"));

      // Add the name and address to the input message
      inputMessage.setObjectPart("name", nameToAdd);
      inputMessage.setObjectPart("address", addressToAdd);

      // Execute the operation, obtaining a flag to indicate its success
      boolean operationSucceeded =
          operation.executeRequestResponseOperation(
              inputMessage,
              outputMessage,
              faultMessage);

      if (operationSucceeded) {
          System.out.println("Successfully added name and address to addressbook\n");
      } else {
          System.out.println("Failed to add name and address to addressbook");
      }

      // Start from fresh
      operation = null;
      inputMessage = null;
      outputMessage = null;
      faultMessage = null;

      // This time we will lookup an address from the addressbook.
      // The getAddressFromName operation is not overloaded in the
      // wsdl and therefore we can simply specify the operation name
      // without any input or output message names.
      operation = port.createOperation("getAddressFromName");
```

```
        // Create the messages
        inputMessage = operation.createInputMessage();
        outputMessage = operation.createOutputMessage();
        faultMessage = operation.createFaultMessage();

        // Set the name to find in the addressbook
        String nameToLookup="Chris P. Bacon";
        inputMessage.setObjectPart("name", nameToLookup);

        // Execute the operation
        operationSucceeded =
            operation.executeRequestResponseOperation(
                inputMessage,
                outputMessage,
                faultMessage);

        if (operationSucceeded) {
            System.out.println("Successful lookup of name '"+nameToLookup+"' in addressbook");

            // We can obtain the address that was found by querying the output message
            WSIFAddress addressFound = (WSIFAddress) outputMessage.getObjectPart("address");
            System.out.println("The address found was:");
            System.out.println(addressFound);
        } else {
            System.out.println("Failed to lookup name in addressbook");
        }

    } catch (Exception e) {
        System.out.println("An exception occurred when running the sample:");
        e.printStackTrace();
    }
}
```

The preceding code refers to the following Sample method:

```
    WSIFPort getPortFromAvailablePortNames(WSIFService service)
            throws WSIFException {
        String portChosen = null;

        // Obtain a list of the available port names for the service
        Iterator it = service.getAvailablePortNames();
        {
            System.out.println("Available ports for the service are: ");
            while (it.hasNext()) {
                String nextPort = (String) it.next();
                if (portChosen == null)
                    portChosen = nextPort;
                System.out.println(" - " + nextPort);
            }
        }
        if (portChosen == null) {
            throw new WSIFException("No ports found for the service!");
        }
        System.out.println("Using port " + portChosen + "\n");

        // An alternative way of specifying the port to use on the service
        // is to use the setPreferredPort method. Once a preferred port has
        // been set on the service, a WSIFPort can be obtained via getPort
        // (no arguments). If a preferred port has not been set and more than
        // one port is available for the port type specified in the WSIFService,
        // an exception is thrown.
        service.setPreferredPort(portChosen);
        WSIFPort port = service.getPort();
        return port;
    }
```

The Web service uses the following classes:

WSIFAddress:

```
public class WSIFAddress implements Serializable {

    //instance variables
    private int streetNum;
    private java.lang.String streetName;
    private java.lang.String city;
    private java.lang.String state;
    private int zip;
    private WSIFPhone phoneNumber;

    //constructors
    public WSIFAddress () { }

    public WSIFAddress (int streetNum,
                        java.lang.String streetName,
                        java.lang.String city,
                        java.lang.String state,
                        int zip,
                        WSIFPhone phoneNumber) {
          this.streetNum = streetNum;
          this.streetName = streetName;
          this.city = city;
          this.state = state;
          this.zip = zip;
          this.phoneNumber = phoneNumber;
    }

    public int getStreetNum() {
          return streetNum;
    }

    public  void setStreetNum(int streetNum) {
          this.streetNum = streetNum;
    }

    public java.lang.String getStreetName() {
          return streetName;
    }

    public  void setStreetName(java.lang.String streetName) {
          this.streetName = streetName;
    }

    public java.lang.String getCity() {
          return city;
    }

    public  void setCity(java.lang.String city) {
          this.city = city;
    }

    public java.lang.String getState() {
          return state;
    }

    public  void setState(java.lang.String state) {
          this.state = state;
    }

    public int getZip() {
          return zip;
    }

    public  void setZip(int zip) {
          this.zip = zip;
    }
```

```
    public WSIFPhone getPhoneNumber() {
          return phoneNumber;
    }

    public  void setPhoneNumber(WSIFPhone phoneNumber) {
          this.phoneNumber = phoneNumber;
    }
}
```

WSIFPhone:

```
public class WSIFPhone implements Serializable {

    //instance variables
    private int areaCode;
    private java.lang.String exchange;
    private java.lang.String number;

    //constructors
    public WSIFPhone () { }

    public WSIFPhone (int areaCode,
                      java.lang.String exchange,
                      java.lang.String number) {
          this.areaCode = areaCode;
          this.exchange = exchange;
          this.number = number;
    }

    public int getAreaCode() {
          return areaCode;
    }

    public  void setAreaCode(int areaCode) {
          this.areaCode = areaCode;
}

    public java.lang.String getExchange() {
          return exchange;
    }

    public  void setExchange(java.lang.String exchange) {
          this.exchange = exchange;
    }

    public java.lang.String getNumber() {
          return number;
    }

    public  void setNumber(java.lang.String number) {
          this.number = number;
    }
}
```

WSIFAddressBook:

```
public class WSIFAddressBook {
    private Hashtable name2AddressTable = new Hashtable();

    public WSIFAddressBook() {
    }

    public void addEntry(String name, WSIFAddress address)
    {
        name2AddressTable.put(name, address);
    }
```

```
    public void addEntry(String firstName, String lastName, WSIFAddress address)
    {
        name2AddressTable.put(firstName+" "+lastName, address);
    }

    public WSIFAddress getAddressFromName(String name)
        throws IllegalArgumentException
    {

        if (name == null)
        {
            throw new IllegalArgumentException("The name argument must not be " +
                                               "null.");
        }
        return (WSIFAddress)name2AddressTable.get(name);
    }

}
```

The following code is the corresponding WSDL file for the Web service:

```
<?xml version="1.0" ?>

<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:typens="http://www.ibm.com/namespace/wsif/samples/ab/types"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
             xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
             xmlns:ejb="http://schemas.xmlsoap.org/wsdl/ejb/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <xsd:schema
       targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab/types"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <xsd:complexType name="phone">
        <xsd:element name="areaCode" type="xsd:int"/>
        <xsd:element name="exchange" type="xsd:string"/>
        <xsd:element name="number" type="xsd:string"/>
      </xsd:complexType>

      <xsd:complexType name="address">
        <xsd:element name="streetNum" type="xsd:int"/>
        <xsd:element name="streetName" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zip" type="xsd:int"/>
        <xsd:element name="phoneNumber" type="typens:phone"/>
      </xsd:complexType>

    </xsd:schema>
  </types>

  <message name="AddEntryWholeNameRequestMessage">
    <part name="name" type="xsd:string"/>
    <part name="address" type="typens:address"/>
  </message>

  <message name="AddEntryFirstAndLastNamesRequestMessage">
    <part name="firstName" type="xsd:string"/>
    <part name="lastName" type="xsd:string"/>
    <part name="address" type="typens:address"/>
  </message>
```

```
<message name="GetAddressFromNameRequestMessage">
  <part name="name" type="xsd:string"/>
</message>

<message name="GetAddressFromNameResponseMessage">
  <part name="address" type="typens:address"/>
</message>

<portType name="AddressBookPT">
  <operation name="addEntry">
    <input name="AddEntryWholeNameRequest"
     message="tns:AddEntryWholeNameRequestMessage"/>
  </operation>
  <operation name="addEntry">
    <input name="AddEntryFirstAndLastNamesRequest"
     message="tns:AddEntryFirstAndLastNamesRequestMessage"/>
  </operation>
  <operation name="getAddressFromName">
    <input name="GetAddressFromNameRequest"
     message="tns:GetAddressFromNameRequestMessage"/>
    <output name="GetAddressFromNameResponse"
      message="tns:GetAddressFromNameResponseMessage"/>
  </operation>
</portType>

<binding name="SOAPHttpBinding" type="tns:AddressBookPT">
  <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryWholeNameRequest">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryFirstAndLastNamesRequest">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="getAddressFromName">
    <soap:operation soapAction=""/>
    <input name="GetAddressFromNameRequest">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output name="GetAddressFromNameResponse">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>

<binding name="JavaBinding" type="tns:AddressBookPT">
  <java:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
           formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>
  <operation name="addEntry">
```

```
    <java:operation
       methodName="addEntry"
       parameterOrder="name address"
       methodType="instance"/>
    <input name="AddEntryWholeNameRequest"/>
  </operation>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="firstName lastName address"
       methodType="instance"/>
    <input name="AddEntryFirstAndLastNamesRequest"/>
  </operation>
  <operation name="getAddressFromName">
    <java:operation
       methodName="getAddressFromName"
       parameterOrder="name"
       methodType="instance"
       returnPart="address"/>
    <input name="GetAddressFromNameRequest"/>
    <output name="GetAddressFromNameResponse"/>
  </operation>
</binding>

<binding name="EJBBinding" type="tns:AddressBookPT">
  <ejb:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
           formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>
  <operation name="addEntry">
    <ejb:operation
       methodName="addEntry"
       parameterOrder="name address"
       interface="remote"/>
    <input name="AddEntryWholeNameRequest"/>
  </operation>
  <operation name="addEntry">
    <ejb:operation
       methodName="addEntry"
       parameterOrder="firstName lastName address"
       interface="remote"/>
    <input name="AddEntryFirstAndLastNamesRequest"/>
  </operation>
  <operation name="getAddressFromName">
    <ejb:operation
       methodName="getAddressFromName"
       parameterOrder="name"
       interface="remote"
       returnPart="address"/>
    <input name="GetAddressFromNameRequest"/>
    <output name="GetAddressFromNameResponse"/>
  </operation>
</binding>
<service name="AddressBookService">
  <port name="SOAPPort" binding="tns:SOAPHttpBinding">
    <soap:address
     location="http://myServer/wsif/samples/addressbook/soap/servlet/rpcrouter"/>
  </port>
  <port name="JavaPort" binding="tns:JavaBinding">
    <java:address className="services.addressbook.WSIFAddressBook"/>
  </port>
  <port name="EJBPort" binding="tns:EJBBinding">
    <ejb:address className="services.addressbook.ejb.AddressBookHome"
      jndiName="ejb/samples/wsif/AddressBook"
classLoader="services.addressbook.ejb.AddressBook.ClassLoader"/>
```

```
    </port>
  </service>

</definitions>
```

Using complex types

WSIF supports user-defined complex types through the mapping of complex types to Java classes. You can specify this mapping manually or automatically.

About this task

Any calls to the WSIFService mapType and mapPackage methods used for manual mapping override any equivalent mapping information that is produced automatically. This override helps to maintain backwards compatibility, and also accommodates less standard mappings.

To map your user-defined complex types to Java classes, complete either of the following steps:

- Manually map complex types.
- Automatically map complex types.
- Manually map complex types.

  The method to use when you create these mappings manually depends on the provider. For the Java and EJB providers, the mappings are specified in the WSDL file in the binding element. The following example provides the syntax for specifying the mapping:

```
<binding .... >
    <ejb:binding|java:binding/>
      <format:typeMapping style="Java" encoding="Java"/>?
          <format:typeMap typeName="qname" formatType="nmtoken"/>*
      </format:typeMapping>
 ...
</binding>
```

  In this example:

  – A question mark (“?”) means “optional” and an asterisk (“*”) means “0 or more”.
  – The format:typeMap **typeName** attribute is a qualified name of a complex type or simple type used by one of the operations.
  – The format:typeMap **formatType** attribute is the fully qualified class name for the Java class to which the element specified by **typeName** maps.

  If you use the Apache SOAP provider then you specify the mapping of a complex type to a Java class in the client code through two methods on the org.apache.wsif.WSIFService interface:

```
public void mapType(QName elementType, Class javaType)
```

  and

```
public void mapPackage(String namespaceURI, String packageName)
```

  Use the **mapType** method to specify a mapping between an XML schema element and a Java class. The method takes a QName representing the complex type or simple type, and the corresponding Java class to which it maps.

  Use the **mapPackage** method to specify a more general mapping between a namespace and a Java package. Any custom, complex or simple type whose namespace matches that of the mapping is mapped to a Java class in the corresponding package. The name of the actual class is derived from the name of the complex type using standard XML to Java naming conventions.

- Automatically map complex types.

  For complex types defined in the WSDL, where a generated bean is used to represent this type in Java, the Web Services Invocation Framework (WSIF) programming model requires that a call is made to the WSIFService.mapType() method. This call tells WSIF the package and class name of the bean representing the XML schema type that is identified with a QName. To make things easier, the WSIFService.mapPackage() method provides a mechanism to specify a wildcard version of this, where

any class within a specified package is mapped to the namespace of a QName. This is a mechanism for manually mapping an XML schema type to a Java class and back again (one mapping entry provides a bidirectional mapping).

There are many ways to convert a QName representing an XML schema type name to a Java package name and class. To enable automatic type mapping, set the WSIF_FEATURE_AUTO_MAP_TYPES feature on the WSIFServiceFactory instance:

```
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
factory.setFeature(WSIFConstants.WSIF_FEATURE_AUTO_MAP_TYPES, new Boolean(true));
```

WSIF maps types by converting the URI part of the XML schema type QName to a package name, and converting the local part to a class name. WSIF does this mapping using the WSIFUtils methods getPackageNameFromNamespaceURI and getJavaClassNameFromXMLName.

Using WSIF to bind a JNDI reference to a Web service

You can use the Web Services Invocation Framework (WSIF) to bind a reference to a Web service, then look up the reference using JNDI.

About this task

You access a Web service through information provided in the WSDL document for the service. If you do not know where to find the WSDL document for the service, but you know that it has been registered in a UDDI registry, then you look it up in the registry. Java programs access Java objects and resources in a similar manner, but using a JNDI interface.

The code fragments in the following steps show how, using WSIF, you can bind a reference to a Web service then look up the reference using JNDI.

- Specify the argument values for the Web service.

  The Web service is represented in WSIF by an instance of the org.apache.wsif.naming.WSIFServiceRef class. This simple Referenceable object has the following constructor:

```
public WSIFServiceRef(
        String WSDL,
        String sNS,
        String sName,
        String ptNS,
        String ptName)
{
    [...]
}
```

  In this example
  - *WSDL* is the location of the WSDL file containing the definition of the service.
  - *sNS* is the full namespace for the service definition (you can specify `null` if only one service is defined in the WSDL file).
  - *sName* is the local name for the service definition (you can specify `null` if only one service is defined in the WSDL file).
  - *ptNS* is the full namespace for the port type within the service that you want to use (you can specify `null` if only one port type is available for the service).
  - *ptName* is the local name for the port type (you can specify `null` if only one port type is available for the service).

  For example, if the WSDL file for the Web service is available from the Web address `http://myServer/WSDL/Example.WSDL` and contains the following service and port type definitions:

```
  <definitions targetNamespace="http://hostname/namespace/example"
               xmlns:abc="http://hostname/namespace/abc"
[...]
    <portType name="ExamplePT">
      <operation name="exampleOp">
        <input name="exampleInput" message="tns:ExampleInputMsg"/>
      </operation>
```

```
    </portType>
[...]
    <service name="abc:ExampleService">
[...]
    </service>
[...]
  </definitions>
```

You can specify the following argument values for the WSIFServiceRef class:
- *WSDL* is `http://myServer/WSDL/Example.WSDL`
- *sNS* is `http://hostname/namespace/abc`
- *sName* is `ExampleService`
- *ptNS* is `http://hostname/namespace/example`
- *ptName* is `ExamplePT`

- Bind the service using JNDI.

  To bind the service reference in the naming directory using JNDI, you can use the com.ibm.websphere.naming.JndiHelper class in WebSphere Application Server:

```
[...]
    import com.ibm.websphere.naming.JndiHelper;
    import org.apache.wsif.naming.*;
[...]
    try {
       Context startingContext = new InitialContext();
       WSIFServiceRef ref = new WSIFServiceRef("http://myServer/WSDL/Example.WSDL",
                                               "http://hostname/namespace/abc"
                                               "ExampleService",
                                               "http://hostname/namespace/example",
                                               "ExamplePT");
       JndiHelper.recursiveRebind(startingContext,
            "myContext/mySubContext/myServiceRef", ref);

    }
    catch (NamingException e) {
       // Handle  error.
    }
[...]
```

- Look up the service using JNDI.

  The following code fragment shows the lookup of a service using JNDI:

```
[...]
    try {
[...]
       InitialContext ic = new InitialContext();
       WSIFService myService =
        (WSIFService) ic.lookup("myContext/mySubContext/myServiceRef");
[...]
    }
    catch (NamingException e) {
       // Handle error.
    }
[...]
```

Example: Passing SOAP messages with attachments using WSIF

Information and example code for using the Web Services Invocation Framework (WSIF) SOAP provider to pass attachments within a MIME multipart/related message in such a way that the SOAP processing rules for a standard SOAP message are not changed. This includes how to write the WSDL extensions for SOAP attachments, and how to work with types and type mappings.

The W3C SOAP Messages with Attachments document describes a standard way to associate a SOAP message with one or more attachments in their native format (for example GIF or JPEG) by using a multipart MIME structure for transport. It defines specific use of the “Multipart/Related” MIME media type,

and rules for the use of URI references to entities bundled within the MIME package. It thereby outlines a technique for carrying a SOAP 1.1 message within a MIME multipart/related message in such a way that the SOAP processing rules for a standard SOAP message are not changed.

WSIF supports passing attachments in a MIME message using the SOAP provider. The attachment is a javax.activation.DataHandler object. The mime:multipartRelated, mime:part and mime:content tags are used to describe the attachment in the WSDL.

- "Example: Writing the WSDL extensions for SOAP attachments"
- "Example: Using WSIF to pass SOAP attachments"
- "SOAP attachments - Working with types and type mappings" on page 873
- "SOAP attachments - scenarios that are not supported" on page 873

Example: Writing the WSDL extensions for SOAP attachments

The following example WSDL illustrates a simple operation that has one attachment called `attch`:

```
<binding name="MyBinding" type="tns:abc" >
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="MyOperation">
    <soap:operation soapAction=""/>
    <input>
      <mime:multipartRelated>
        <mime:part>
          <soap:body use="encoded" namespace="http://mynamespace"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </mime:part>
        <mime:part>
          <mime:content part="attch" type="text/html"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
  </operation>
</binding>
```

In this type of WSDL extension:

- There must be a `part` attribute (in this example `attch`) on the input message for the operation (in this example `MyOperation`). There can be other input parts to `MyOperation` that are not attachments.
- In the binding input there must either be a `<soap:body>` tag or a `<mime:multipartRelated>` tag, but not both.
- For MIME messages, the `<soap:body>` tag is inside a `<mime:part>` tag. There must only be one `<mime:part>` tag that contains a `<soap:body>` tag in the binding input and that must not contain a `<mime:content>` tag as well, because a content type of `text/xml` is assumed for the `<soap:body>` tag.
- There can be multiple attachments in a MIME message, each described by a `<mime:part>` tag.
- Each `<mime:part>` tag that does not contain a `<soap:body>` tag contains a `<mime:content>` tag that describes the attachment itself. The `type` attribute inside the `<mime:content>` tag is not checked or used by the Web Services Invocation Framework (WSIF). It is there to suggest to the application using WSIF what the attachment contains. Multiple `<mime:content>` tags inside a single `<mime:part>` tag means that the backend service expects a single attachment with a type specified by one of the `<mime:content>` tags inside that `<mime:part>` tag.
- The `parts="..."` attribute (optional) inside the `<soap:body>` tag is assumed to contain the names of all the MIME parts as well as the names of all the SOAP parts in the message.

Example: Using WSIF to pass SOAP attachments

The following code fragment can invoke the service described by the example WSDL in "Example: Writing the WSDL extensions for SOAP attachments":

```
import javax.activation.DataHandler;
. . .
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
```

```
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
WSIFOperation op = service.getPort().createOperation("MyOperation");
WSIFMessage in = op.createInputMessage();
in.setObjectPart("attch",dh);
op.executeInputOnlyOperation(in);
```

The associated type mapping in the `DeploymentDescriptor.xml` file depends upon your SOAP server. For example if you use Tomcat with SOAP 2.3, then the `DeploymentDescriptor.xml` file contains the following type mapping:

```
<isd:mappings>
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:x="http://mynamespace"
 qname="x:datahandler"
 javaType="javax.activation.DataHandler"
 java2XMLClassName="org.apache.soap.encoding.soapenc.MimePartSerializer"
 xml2JavaClassName="org.apache.soap.encoding.soapenc.MimePartSerializer" />
</isd:mappings>
```

In this case, the backend service is invoked with the following signature:

```
public void MyOperation(DataHandler dh);
```

You can also use stubs to pass attachments into the Web Services Invocation Framework (WSIF):

```
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
MyInterface stub = (MyInterface)service.getStub(MyInterface.class);
stub.MyOperation(dh);
```

Attachments can also be returned from an operation, but only one attachment can be returned as the return parameter.

SOAP attachments - Working with types and type mappings

By default, attachments are passed into WSIF as DataHandler objects. If the part on the message that is the DataHandler object maps to a `<mime:part>` tag in the WSDL, then WSIF automatically maps the fully qualified name of the WSDL type to the DataHandler class and sets up that type mapping with the SOAP provider.

In your WSDL, you might have defined a schema for the attachment (for instance as a `binary[]` type). WSIF silently ignores this mapping and treats the attachment as a DataHandler object, unless you explicitly issue a `mapType()` method. WSIF lets the SOAP provider set the MIME content type based on the type of the DataHandler object, instead of the `type` attribute specified for the `<mime:content>` tag in the WSDL.

SOAP attachments - scenarios that are not supported

The following scenarios are not supported:

- Using DIME.
- Passing in `javax.xml.transform.Source` and `javax.mail.internet.MimeMultipart`.
- Using the mime:mimeXml WSDL tag.
- Nesting a mime:multipartRelated tag inside a mime:part tag.
- Using types that extend `DataHandler`, `Image`, and so on.
- Using types that contain `DataHandler`, `Image`, and soon.
- Using Arrays or Vectors of `DataHandlers`, `Images`, and so on.
- Using multiple in/out or output attachments.

The MIME headers from the incoming message are not preserved for referenced attachments. The outgoing message contains new MIME headers for Content-Type, Content-Id and Content-Transfer-Encoding that are created by WSIF.

Interacting with the Java EE container in WebSphere Application Server

How, and to what extent, WSIF interacts with the Java EE container that is provided in WebSphere Application Server.

About this task

You can interact with a container in any of the following ways:

- Use the application server administrative console to define Web services to WebSphere Application Server. This task is described in Using the Java Naming and Directory Interface (JNDI) and Installing and managing WSIF. As part of the definition of a service, the administrator might define a “preferred port”.
- Use the Web Services Invocation Framework (WSIF) to make log and trace calls to the JRAS services in WebSphere Application Server, as described in Trace and logging for WSIF.
- Use WSIF providers to access Java Platform, Enterprise Edition (Java EE) services. For example, use the EJB provider to access the Java Naming and Directory Interface (JNDI) and make calls to remote enterprise beans.
- Use WSIF to wrap the use of container services so that, when WSIF is run in an unmanaged (thin) environment, the operation can succeed.

Running WSIF as a client

You can run the Web Services Invocation Framework (WSIF) in the Application Client for WebSphere Application Server, and in similar clients from other suppliers.

To simplify the process of launching client applications in the Application Client for WebSphere Application Server, use the `launchClient` tool as described in Running application clients.

Invoking a WSDL-based Web service through the WSIF API

The Web Services Invocation Framework (WSIF) provides a Java API for invoking Web services, independent of the format of the service or the transport protocol through which it is invoked.

Before you begin

WSIF includes an EJB provider for EJB invocation using Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). However, for EJB(IIOP)-based Web service invocation you should instead invoke RMI-IIOP Web services using JAX-RPC.

You must ensure that your application uses only one thread to call WSIF.

About this task

The WSIF API supports the invocation of WSDL-defined Web services. WSIF is intended for use in both WSIF clients and Web service intermediaries.

The WSIF API is driven by the abstract service description in WSDL; it is completely independent of the actual binding used. This independence makes the API more natural to work with because it uses WSDL terms to refer to message parts, operations, and so on.

The WSIF API was designed for the WSDL usage model:

1. Pick a port that supports the port type that you need.
2. Invoke the operation by providing the necessary abstract input message consisting of the required parts, without worrying about how the message is mapped to a specific binding protocol.

Other Web service APIs, for example SOAP APIs, are not designed on WSDL, but for a specific binding protocol with its associated syntax; for example, target URIs and encoding styles.

The main WSIF API interfaces are described within the following task steps. For additional technical details of the WSIF API, see the generated API information.

- Create a message for sending to a port through the WSIFMessage interface.

  In WSDL, a message describes the abstract type of the input or output to an operation. The corresponding WSIF class is WSIFMessage, which represents in memory the actual input or output of an operation. The WSIFMessage interface separates the actual representation of the data from the abstract type defined by WSDL.

  A WSIFMessage class is a container for a set of named parts. WSIFMessage classes can be sent between Java Virtual Machines (JVMs).

  1. Choose whether to represent the WSIF message at run time as a Java class or as XML.

     There are two natural ways to represent a WSDL message in a run-time environment:
     – The generated Java class, based on a WSDL to Java mapping such as that provided by a Java API for XML-based remote procedure call (JAX-RPC).
     – The XML representation of the data, for example using SOAP Encoding.

     Each option offers benefits in different scenarios. The Java class is the natural approach when WSIF is used in a standard Java client. However, in other scenarios where WSIF is used in an intermediary, it might be more efficient to keep a WSDL message in the SOAP encoded format. The style used to define messages must be consistent within the message, so all the parts in one message must be consistent. A string - `getRepresentationStyle()` - always returns `null`. This indicates that parts on this WSIFMessage class are represented as Java objects.
  2. Get and set the parts of the WSIF message.

     You add parts to a WSIFMessage class with the setObjectPart or set*Type*Part methods. Each part is named. Part names within a message are unique. If you set a part more than once, the last setting is the one that is used.

     You retrieve parts by name from a WSIFMessage class with the getObjectPart or get*Type*Part methods. If the named part does not exist, the method returns a WSIFException exception.

     You can use Iterators to retrieve parts from the message through the getParts() and getPartNames() methods.

     The order in which you set the parts is not important, but the message implementation might be more efficient if the parts are set in the parameter order specified by WSDL.

     WSIFMessage classes are cloneable and serializable. If the parts set are not cloneable, the implementation can try to clone them using serialization. If the parts are not serializable either, then a CloneNotSupportedException exception is thrown if cloning is attempted.
  3. Set the WSIF message name.

     In addition to the containing parts, a WSIFMessage class also has a message name. This is required for operation overloading, which is supported by WSDL and WSIF.

  For more information about the WSIFMessage interface (/wsi/org/apache/wsif/WSIFMessage.html) see the generated API information.
- Find a port factory or service through the WSIFService interface and the WSIFServiceFactory class.

  The WSIFService interface is a port factory that models and supports the WSDL approach in which a service is available on one or more ports. The factory hides the implementation of the port. WSIF supports dynamic ports that are based on a particular protocol and transport and configured using the WSDL at run time. For example, the dynamic SOAP port can invoke any SOAP service based on the WSDL description of that service, so you can hide and modify the set of available ports at run time.

  For more information, see the “WSIFService interface” on page 876.

  To find a service from a WSDL document at a Web address, or from a code-generated code base, use the “WSIFServiceFactory class” on page 877.
- Invoke an operation through the WSIFPort interface and the WSIFOperation interface.

A WSIFPort interface handles the details of invoking an operation. The port provides access to the actual implementation of the service.

A WSDL document can provide many different WSDL bindings, and these bindings can drive multiple ports. The client can choose a port, the service stub can choose a port, or WSIF can choose a default port.

The port offers an interface to retrieve an Operation object. A WSIFOperation interface offers the ability to execute the given operation.

If the port is serialized and deserialized at a later time, then WSIF ensures that the client provides the correct information to the server to identify the instance. If the server instance is no longer available, then it is up to the server to decide whether to throw a fault or provide a new instance. That behavior can depend on the type of service. For example, for an enterprise bean the WSIFPort interface stores the EJB Home, and uses it to select the bean before each invocation. It is the responsibility of the client to serialize or maintain the port instance if it wants instance support. The client must create a new operation and messages for each invocation.

For more information, see the “WSIFPort interface” on page 877 or the “WSIFOperation interface” on page 877.

WSIFService interface

The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation.

The Web Services Invocation Framework (WSIF) service stores a list of providers that can each generate a WSIF operation for a particular WSDL binding. This service looks up providers by the provider type. For example the service knows about one provider that handles SOAP ports and other providers that handle Java ports that you define. In a managed environment, the container can configure the WSIFService interface.

For more information about the WSIFService interface (/wsi/org/apache/wsif/WSIFService.html) see the generated API information.

A WSIFService implementation can choose a preferred port based on a number of criteria. The WSIFService implementation can set the preferred port, or it can be set by calling the setPreferredPort method.

The getPort method returns an instance of the WSIFPort class that is used to invoke a service on the port. Variants of the getPort method are used to define the characteristics of the port to be created:

- the getPort method with no arguments returns the preferred port.
- the getPort method with a string argument returns the port named by the string containing the WSDL identifier for the selected port.

The return value is `null` if the port name is not valid.

If a port is chosen (either by the WSIFService implementation, or by the setPreferredPort method), then the WSIFService implementation validates that the relevant provider exists and is configured. If the provider fails this validation check, the WSIFService interface chooses any other port for which a provider is defined. For example, if the preferred port is SOAP over JMS but the JMS libraries are not available, then WSIF chooses another port. If no preferred port is set, or the preferred port is not available, the WSIF implementation chooses the first available port listed in the WSDL.

The getAvailablePortNames() method returns, as an iteration of strings, the list of WSDL port names filtered by the set of available providers.

The getDefinition() method returns the WSDL definition for the service. If the WSDL definition is not available, this method returns `null`.

WSIFServiceFactory class

To find a service from a WSDL document at a Web address, or from a code-generated code base, you can use the WSIFServiceFactory class.

Note: When you create a WSIFService interface from a WSIFServiceFactory class, you can specify a ClassLoader object to use in locating the WSDL file. You need to specify this object when the WSDL file is in a JAR file. In such a case, specify the location of the WSDL file relative to the root of the JAR file, using forward slashes (/) with the preceding slash removed.

For example:

```
com/myCompany/wsdl/MyWSDLFile.wsdl
```

rather than

```
/com/myCompany/wsdl/MyWSDLFile.wsdl
```

For more information about the WSIFServiceFactory class (/wsi/org/apache/wsif/WSIFServiceFactory.html) see the generated API information.

The WSIFServiceFactory class returns `null` if no service is found with that identifier.

WSIFPort interface

The port implements a factory method for the WSIFOperation interface.

For detailed information about the WSIFPort interface (/wsi/org/apache/wsif/WSIFPort.html) see the generated API information.

The createOperation(String) method returns a new instance of a WSIFOperation object. If the operationName value is not valid or the operation is overloaded, then the method throws an exception.

The createOperation(String, String, String) method supports overloaded WSDL operations. You can overload based on the input parameters, but not on the output parameters.

It is the duty of the client to call the close method when a port is no longer in use. In many cases, where the transport is sessionless, like HTTP, this has no effect. However, if the port is using a session-based protocol such as MQSeries, Java Message Service (JMS), or External Call Interface (ECI), this supports the port in caching an open connection to the server and then closing it as required. Responsibly-written applications will call the close method if appropriate.

WSIFOperation interface

You use the WSIFOperation interface to invoke a service based on a particular binding.

The WSIFOperation interface is the run-time representation of an operation. This interface provides methods to create input, output, and fault messages, and to invoke the operation.

For more information about the WSIFOperation interface (/wsi/org/apache/wsif/WSIFOperation.html) see the generated API information.

createInputMessage, createOutputMessage and createFaultMessage
These are factory methods to create the messages required by the invocation methods. All invocation methods require an input message.

executeRequestResponseOperation
This method invokes “In Out” operations.

executeInputOnlyOperation
This method invokes “In only” operations.

executeRequestResponseOperation
If this method is used for invocation, then an output and a fault message are instantiated and passed on the call to the method. If the method returns `true`, then the output message contains the response message. If the message returns `false`, then a fault occurred and is returned in the fault message.

executeRequestResponseAsync
This method allows “In Out” operations to be invoked with the reply handled using an alternate thread. Use of this method is discussed further in WSIFOperation - Asynchronous interactions.

setContext and getContext
Use of these methods is discussed in WSIFOperation - Context.

All of the **execute*Nnnn*** methods fail with an exception if there is an error in processing the request in the WSIF provider.

Setting the timeouts for synchronous and asynchronous operations is discussed in WSIFOperation - Synchronous and asynchronous timeouts.

WSIFOperation - Context

Although WSDL does not define context, a number of uses of the Web Services Invocation Framework (WSIF) require the ability to pass context to the port that is invoking the service.

For example, a SOAP over HTTP port might require an HTTP user name and password. This information is specific to the invocation, but is not a parameter of the service. In general, context is defined as a set of name-value pairs. However, because Web services tend to define the types of data using XML schema types, WSIF represents the name-value pairs of the context using the same representation that WSIFMessage classes use; that is a set of named parts, each of which equates to an instance of an XML schema type.

You use the WSIFOperation interface setContext and getContext methods to pass context information to the binding. The port implementation can use this context, for example to update a SOAP header. There is no definition of how a port can utilize the context.

The parameter of the setContext and getContext methods is a WSIFMessage interface, and this interface has named parts defining the context information. The WSIFConstants class defines constants for the part names that can be set in a context WSIFMessage interface.

The following code fragment shows how to set the user name and password for HTTP basic authentication:

```
// set a basic authentication header
   WSIFMessage headers = new WSIFDefaultMessage();
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_USER, "user name" );
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_PSWD, "password" );
   operation.setContext( headers );
```

The WSIFOperation interface ignores context parts that it does not support. For example, the previous code is ignored by the WSIF Java provider.

The WSIFConstants class includes the following constants that can be used for context part names:
- CONTEXT_HTTP_USER
- CONTEXT_HTTP_PSWD
- CONTEXT_SOAP_HEADERS

The HTTP header values are expected to be of type String, and the SOAP header value is expected to be of type java.util.List, which should contain entries of type org.w3c.dom.Element.

WSIFOperation - Asynchronous interactions reference

The Web Services Invocation Framework (WSIF) supports asynchronous operation. In this mode of operation, the client puts the request message as part of one transaction, and carries on with the thread of execution. The response message is then handled by a different thread, with a separate transaction.

Asynchronous operation is supported by the WSIF providers for SOAP over JMS and native JMS.

The WSIFPort class uses the supportsAsync method to test if asynchronous operation is supported.

An asynchronous operation is initiated with the WSIFOperation interface executeRequestResponseAsync method. This method lets a Remote Procedure Call (RPC) method be invoked asynchronously. The method returns before the operation is completed, and the thread of execution continues.

The response to the asynchronous request is processed by the WSIFOperation interface fireAsyncResponse or processAsyncResponse methods.

To initiate the request, there are two forms of the executeRequestResponseAsync method:

```
public WSIFCorrelationId executeRequestResponseAsync
                         (WSIFMessage input, WSIFResponseHandler handler)
```

and

```
public WSIFCorrelationId executeRequestResponseAsync (WSIFMessage input)
```

executeRequestResponseAsync(WSIFMessage input, WSIFResponseHandler handler)

This method takes an input message and a WSIFResponseHandler handler. The handler is invoked on another thread when the operation completes. When using this method the client listener calls the fireAsyncResponse method, which then calls the WSIFResponseHandler interface executeAsyncResponse method.

For more information about the WSIFResponseHandler interface (/wsi/org/apache/wsif/WSIFResponseHandler.html), see the generated API information.

executeRequestResponseAsync(WSIFMessage input)

This method only takes an input message, and does not use a response handler. The client listener processes the response by calling the WSIFOperation interface processAsyncResponse method. This process updates the WSIFMessage output and fault messages with the result of the request.

WSIF supports correlation between the asynchronous request and response. When the request is sent, the WSIFOperation object is serialized into the WSIFCorrelationService object. The executeRequestResponseAsync methods return a WSIFCorrelationId object which identifies the serialized WSIFOperation object. The client listener can use this to match a response to a particular request.

The correlation service is located with the WSIFCorrelationServiceLocator class getCorrelationService() method in the org.apache.wsif.utils package.

In a managed container a default correlation service is defined in the default Java Naming and Directory Interface (JNDI) namespace using the name: java:comp/wsif/WSIFCorrelationService. If this correlation service is not available, then WSIF uses the WSIFDefaultCorrelationService.

For more information about the WSIFCorrelationService interface (/wsi/org/apache/wsif/WSIFCorrelationService.html) see the generated API information.

and this is the correlator ID:

```
public interface WSIFCorrelator extends Serializable {
   public String getCorrelationId();
}
```

The client must implement its own response message listener or message data base so that it can recognize the arrival of response messages. This client implementation manages the correlation of the response message to the request and call of one of the asynchronous response processing methods. As an example of the requirement for a client listener, the following code fragment shows what can be in the onMessage method of a Java Message Service (JMS) listener:

```
public void onMessage(Message msg) {
     WSIFCorrelationService cs = WSIFCorrelationServiceLocator.getCorrelationService();
           WSIFCorrelationId cid = new JmsCorrelationId( msg.getJMSCorrelationID() );
           WSIFOperation op = cs.get( cid );
           op.fireAsyncResponse( msg );
}
```

WSIFOperation - Synchronous and asynchronous timeouts reference

When you use the Web Services Invocation Framework (WSIF) with the Java Message Service (JMS) you can set timeouts for synchronous and asynchronous operations.

Default values for these timeouts are defined in the wsif.properties file:

```
# maximum number of milliseconds to wait for a response to a synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined or invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

If you use these default values, a synchronous request (such as a WSIFOperation interface executeRequestResponseOperation method call) times out after ten seconds, and an asynchronous request (such as a WSIFOperation interface executeRequestResponseAsync method call) times out after sixty seconds.

Note:

> The code that processes both of these timeout values uses milliseconds as its unit of time. The WSIFProperties class getAsyncTimeout method multiplies the wsif.asyncrequest.timeout value by 1000, to convert the value from seconds to milliseconds.

You can override these default values for a given request by setting a JMS property on the operation request with the <jms:property> and <jms:propertyValue> WSDL elements. Set the name of the property to be the name of the timeout from the WSIF properties file.

The following example sets synchronous requests to time out after two minutes (120 seconds):

```
<jms:propertyValue name="wsif.syncrequest.timeout" type="xsd:string" value="120000"/>
```

and the following example disables asynchronous timeouts (a value of zero means wait forever):

```
<jms:propertyValue name="wsif.asyncrequest.timeout" type="xsd:string" value="0"/>
```

When an asynchronous timeout expires, no listener or message data base waiting for the response is notified. The asynchronous timeout is only used to tell the correlation service that the stored WSIFOperation can be deleted.

Chapter 7. Service integration

Programming mediations

This topic is an overview of the tasks involved in programming a mediation. Typically, the mediation code is written by a programmer, and is then deployed and administered by an integrator.

Before you begin

Code examples for writing a mediation are provided at “Adding mediation function to handler code” on page 883.

The following application programming interfaces are provided for you to work with messages:

- SIMessage API allows you to manipulate the contents of the message.
- SIMediationSession API provides access to the service integration bus so that your mediation can send and retrieve messages.

Mediations are deployed using Rational Application Developer tools.

About this task

The tasks for programming a mediation are:

Developing
Writing a mediation by adding functional code to a mediation handler.

Deploying
Adding a mediation to a mediation handler list, and deploying it.

Administering
Associating a mediation handler with a destination (optional), and configuring the parameters to be used by the mediation handler at run time.

Take the following steps to program a mediation:

1. Create a mediation handler. For more information, see “Writing a mediation handler” on page 882.
2. Add mediation function code to your mediation handler. For more information, see “Adding mediation function to handler code” on page 883.
3. Working with the message payload, for example for logging messages within a mediation. For more information, see “Working with the message payload” on page 890.
4. Use the Rational Application Developer tools to create a handler list, add your mediation handler to the list, and deploy the handler list as an Enterprise Archive (EAR file). See the Rational Application Developer information center for information about how to do this.

Serializing the content of SIMessage

Use this task to convert an SIMessage object to a byte array.

About this task

If you want to save an SIMessage object in your local file system or in a database, you must first convert the object to a byte array and format string. You can reconstruct the message from the byte array and format string. To do this, complete the following steps.

1. In your application program, record the format string associated with the SIMessage instance. For example:

```
String savedFormat=message.getFormat();
```

2. Call the `getDataGraphAsBytes`. For example:

```
Bytes newDataGraph = message.getNewDataGraph(newFormat);
```

   This method returns a copy of the payload as a byte stream. You can store the bytes and the associated format string, as you require.
3. To reconstruct the message, call the method `createDataGraph` provided by the SIDataGraphFactory API. This method requires a byte array and a format string. For example:

```
DataGraph newDataGraph = SIDataGraphFactory.getInstance().createDataGraph(byteArray, newFormat);
```

   This method creates a new data graph by parsing the bytes according to the format passed to the method.

What to do next

You can use the newly created datagraph as the payload of an SIMessage instance by using the SIMessage setDataGraph() method. For example:

```
newMessage.setDataGraph(newDataGraph, savedFormat);
```

Writing a mediation handler

This topic outlines how to write a mediation handler, add mediation function to it, and prepare it for installation on an application server.

Before you begin

Before you start this task, you should have access to a Java programming environment, and an assembly tool such as IBM Rational Application Developer.

About this task

A mediation handler can be deployed. Each mediation handler executes some specific message processing at run time, for example transforming a message format, or routing a message to a particular destination. A mediation handler is a Java program framework, to which you add the code that performs the mediation function. For more information about handlers, see Mediation handlers and mediation handler lists.

Your mediation handler class can be defined either in a Java project or an EJB project (which is needed for the deployment artifact.) Your programming and deployment artifacts can be separated in different projects. The steps below are for an EJB project, but the steps are very similar if you want to create a Java project, since you simply define a target server for either a Java project or an EJB project and the server runtime plug-in sets the class path correctly.

1. Create a new EJB project:
   a. Switch to the Java EE perspective to work with Java EE projects. Click **Window > Open Perspective > Other > Java EE**.
   b. From the File menu, select **New > Project**.
   c. Expand the Java EE folder, and select Enterprise Application Project. Click **Next**.
   d. Optional: If you have created a Java project instead of an EJB project, right click on the Java project folder icon for the context menu and select Properties. When the Properties panel appears, select the Server properties and target the project to WebSphere Application Server Version 7.0, as in the next step.
   e. Enter a name for the project and target the project to WebSphere Application Server Version 7.0. (If this is the first time you target this server you will need to click the **New...** button.) Click **Next** to take you to the EAR Module Projects window.
   f. Click **New Module...**.

g. Create a new module project by selecting the check box against EJB project, and entering the name of your mediation handler.
h. Click **Finish**. You are returned to the EAR Module Projects window.
i. Click **Finish** to create the new project.

2. Create a mediation handler class by implementing the `com.ibm.websphere.sib.mediation.handler.MediationHandler` interface.
   a. From the File menu, select **New > Java Class**.
   b. Specify the source folder for your mediation EAR project.
   c. Specify a name for your mediation handler.
   d. Select Superclass `java.lang.Object`.
   e. Select Interface `com.ibm.websphere.sib.mediation.handler.MediationHandler`.
   f. Click in the check box to select Inherited abstract methods
   g. Click **Finish** to create the new mediation handler class.
3. Add functional code that transforms or routes messages to your mediation handler using the IBM Rational Application Developer. For more information, see "Adding mediation function to handler code." Beware that the default return value for the handle method created by the toolkit is `false`, which causes the message to be discarded. You need to change the return value to `true` to preserve the message.
4. Generate an EAR file from your mediation handler class. Follow the instructions in the IBM Rational Application Developer documentation.

What to do next

Next, you are ready to install the EAR file containing your mediation handler into the application server.

Adding mediation function to handler code

This topic directs you to different tasks that enable you to add mediation function to an existing mediation handler.

Before you begin

Before you start, you need a mediation handler in an EJB project. For more information, see "Writing a mediation handler" on page 882.

About this task

There are four ways in which you can change the behavior of the mediation handler code, or influence the routing of a message. You can change the values in the message context and message properties, and you can work with the contents of the message (called the message payload), or with the message header:

- "Working with the message context" on page 884
- "Working with the message properties" on page 885
- "Working with the message header" on page 886
- "Working with the message payload" on page 890

Example: Using mediations to trace, monitor and log messages

The most straightforward use of a mediation is for tracing, monitoring or logging messages that pass through a destination or topics spaces. This type of mediation does not modify the message; it simply extracts information from the message and saves or displays the information elsewhere.

For example, the following mediation handler displays the API message and correlation IDs for each message it is sent:

```
public boolean(MessageContext context)
{
  SIMessageContext   msgCtx  = (SIMessageContext)context;
  SIMediationSession session = msgCtx.getSession();
  SIMessage          msg     = msgCtx.getMessage();
  String             msgId   = msg.getApiMessageId();
  String             corrId  = msg.getCorrelationId();
  String             dest    = session.getDestinationName();

  System.out.println(msgId+" (correlation id="+corrid) is passing through "+dest+".");

  return true;
}
```

Working with the message context

This topic describes how to work with the message properties to affect the way a message is mediated.

Before you begin

Before you start this task, you should read about how information is carried in the mediation context in Mediation context information

About this task

Interface SIMessageContext has a superinterface MessageContext. Methods in MessageContext allow you to manage a set of message properties, which enable handlers in a handler chain to share processing-related state. Most importantly, you can get the value of a specific property from the MessageContext using the method getProperty, and you can set the name and value of a property associated with the MessageContext using the method setProperty. You can also view the names of the properties in this MessageContext and remove a property (that is, a name-value pair) from the MessageContext.

At mediation runtime, all of the user-defined properties that have been set during configuration for the current mediation (see Configuring mediation context properties) are applied to the MediationContext property set.

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. As you are working with the MessageContext methods that give you access to message properties, you do not need to cast the interface to SIMessageContext **unless** you are also interested in the methods provided by SIMessageContext.
2. Get the SIMessage from the MessageContext object. For example, `SIMessage message = ((SIMessageContext)context).getSIMessage();`
3. Retrieve or set properties, using the MessageContext methods. For instance, if a property has been defined during configuration with the name streetName, the type String, and the value ″Main Street″ your code to retrieve and print the street name may look like this:

Example

```
public boolean handle(MessageContext context) throws MessageContextException {

   ........
  {
    /* Retrieve the street name property */
    String myStreetName;
    myStreetName = (String) getProperty(streetName);

    /* Display property value */
```

```
    System.out.println(myStreetName);

  }
}
```

Working with the message properties

This topic describes how to work with the message properties to affect subsequent processing.

Before you begin

Before you start this task, you should read about the properties that are supported by the SIMessage interface in Message properties support for mediations.

About this task

There are two different types of message properties:

- System properties (including JMS headers, JMSX properties, and JMS_IBM_properties)
- User properties.

You can work with message properties to affect which messages a later mediation should process, or to affect processing by a downstream application or mediation. The rule set in the selector field during mediation configuration tests values in the message properties.

You can access, modify and clear properties using the SIMessage interface (see “SIMessage” on page 893.) There are three different sets of methods:

- These properties operate on system properties, plus user properties if the name is qualified with a prefix `user.`:
  - getMessageProperty
  - setMessageProperty
  - deleteMessageProperty
  - clearMessageProperties
- These properties operate on user properties only, without the need for the prefix `user.`:
  - getUserProperty
  - setUserProperty
  - deletUserProperty
  - clearUserProperties
- getUserPropertyNames returns a list of the names of the user properties in the message.

Typically, you can work with message properties in the following way, when programming a mediation:

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext **unless** you are only interested in the methods provided by MessageContext.
2. Get the SIMessage from the MessageContext object. For example, `SIMessage message = ((SIMessageContext)context).getSIMessage();`
3. Build your mediation header function in a similar way to these examples, using the reference information in Message properties support for mediations to help:
   a. Get a user property of the message. For instance, `String task = (String)msg1.getUserProperty("task");`. In this case, the task string may refer to an operation that the mediation should perform.
   b. Set a user property, where message Properties are stored as name-value pairs. The setUserProperty method may only be used to set user properties, so the name passed into the method should not include the ″user.″ prefix. For example, `msg1.setUserProperty("background","green");`

c. Delete a user property from the message. For instance, `msg1.deleteUserProperty("task");`

Example

Mediation function code to work with message properties may look similar to the code snippet in this example:

```
String task = (String)msg1.getUserProperty("task");
if (task != null) {
  if (task.equals("addColor")) {
    msg1.setMessageProperty(SIProperties.JMS_IBM_Format, "colorful");
    msg1.setUserProperty("background","green");
    msg1.setUserProperty("foreground","purple");
    msg1.setUserProperty("depth",new Integer(3));
    msg1.deleteUserProperty("task");
  }
  else {
    msg1.clearUserProperties();
  }
}
```

Working with the message header

This topic describes how to add function to a preexisting mediation handler to operate on the message header.

Before you begin

Before you start this task, you should have created the basic mediation handler in an EJB project (see “Writing a mediation handler” on page 882. It will be useful to have understood the elements of the task “Working with the message payload” on page 890, because some of those elements are used in this task

About this task

There are different types of field that you can set in message headers. Importantly, you can set the forward and return routing addresses for messages after they have been mediated at the current destination. In addition there are other fields that you can set, such as priority and reliability for the message and its reply, and the remaining time before the message (or the reply) expires.

1. To set routing addresses in the message header, see “Setting routing addresses in a message header.”
2. To set all other fields in the message header, see “Working with non-routing path fields in a message header” on page 889.

Setting routing addresses in a message header:

This topic describes how to add function to a pre-existing mediation handler to set routing addresses in the message header.

Before you begin

Before you start this task, you should have created the basic mediation handler in an EJB project (see “Writing a mediation handler” on page 882.

About this task

To work with routing addresses, you will use the SIDestinationAddress and SIDestinationAddressFactory APIs. The SIDestinationAddress is the public interface which represents an service integration bus, and gives your mediation access to the name of the destination and the bus name. SIDestinationAddressFactory enables you to create a new SIDestinationAddress to represent an service

integration bus destination. For reference information about these APIs, see “SIDestinationAddress” on page 888 and “SIDestinationAddressFactory” on page 888.

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext **unless** you are only interested in the methods provided by MessageContext.
2. Get the SIMessage from the MessageContext object. For example:
   ```
   SIMessage message = ((SIMessageContext)context).getSIMessage();
   ```
3. Build your mediation header function using these basic steps:
   a. Get a handle to the core runtime environment. For example:
      ```
      .... SIMediationSession mediationSession = mediationContext.getSession();
      ```
   b. Create a forward routing path to set on the cloned object. For example, use the Vector class to create a extendable array of objects.
   c. Get the SIDestinationAddressFactory which is to be used for creating SIDestinationAddress instances. For example:
      ```
      SIDestinationAddressFactory destFactory = SIDestinationAddressFactory.getInstance();
      ```
   d. Create a new SIDestinationAddress, representing a service integration bus destination. For example:
      ```
      SIDestinationAddress dest = destFactory.createSIDestinationAddress(remoteDestinationName(),false);
      ```
      In this case, the second parameter, the Boolean “false”, indicates that the destination should not be localized to the local messaging engine, but can be anywhere on the service integration bus.
   e. Use the add method of the Vector class to add another destination name to the array.
   f. Clone the message, and modify the forward routing path in the cloned message. For example:
      ```
      clonedMessage.setForwardRoutingPath(forwardRoutingPath);
      ```
   g. Send the cloned message using the send method in the SIMediationSession interface to send the message to the service integration bus. For example, if named ″clonedMessage″:
      ```
      mediationSession.send(clonedMessage, false);
      ```
4. Return *true* to ensure the message passed into the handle method of the MediationHandler interface continues along the handler chain.

Example

The complete mediation function code to change the forward routing path might look like this example:

```
/* A sample mediation that simply clones a message
 * and sends the clone off to another destination  */

public class RoutingMediationHandler implements MediationHandler {

 public String remoteDestinationName="newdest";

 public boolean handle(MessageContext context) throws MessageContextException {
  SIMessage clonedMessage = null;
  SIMessageContext mediationContext = (SIMessageContext) context;
  SIMessage message = mediationContext.getSIMessage();
  SIMediationSession mediationSession = mediationContext.getSession();

  // Create a forward routing path which will be set on the cloned message
  Vector forwardRoutingPath = new Vector();
  SIDestinationAddressFactory destFactory =
        SIDestinationAddressFactory.getInstance();
  SIDestinationAddress dest =
        destFactory.createSIDestinationAddress(remoteDestinationName,false);
  forwardRoutingPath.add(dest);
```

```
  try {
   // Clone the message
   clonedMessage = (SIMessage) message.clone();
   // Modify the forward routing path for the clone
   clonedMessage.setForwardRoutingPath(forwardRoutingPath);
   // Send the message to the next destination in the frp
   mediationSession.send(clonedMessage, false);
  } catch (SIMediationRoutingException e1) {
   e1.printStackTrace();
  } catch (SIDestinationNotFoundException e1) {
   e1.printStackTrace();
  } catch (SINotAuthorizedException e1) {
   e1.printStackTrace();
  } catch (CloneNotSupportedException e) {
   // SIMessage should clone OK so we shouldn't really enter this block
   e.printStackTrace();
  }
  // allow original message to continue on its path
  return true;
 }
```

SIDestinationAddress:

This topic describes the SIDestinationAddress, the public interface which represents a service integration bus destination.

The API has three methods:

- isTemporary: This method determines whether the SIDestinationAddress represents a temporary or permanent Destination, returning a Boolean value.
- getDestinationName: Method to retrieve the name of the Destination represented by this SIDestinationAddress.
- getBusName: Method to retrieve the bus name of the Destination represented by this SIDestinationAddress.

For more information about the SIDestinationAddress interface, see the SIDestinationAddress generated API information.

SIDestinationAddressFactory:

This topic describes the SIDestinationAddressFactory, the public interface which is used for the creation of each instance of an SIDestinationAddress.

public abstract class SIDestinationAddressFactory extends java.lang.Object

This class creates an SIDestinationAddressFactory at static initialization that is subsequently used for the creation of all instances of SIDestinationAddress

The API has three methods:

- getInstance: This method gets the singleton SIDestinationAddressFactory which is to be used for creating SIDestinationAddress instances.
- createSIDestinationAddress: These two methods are used to create a SIDestinationAddress to represent a service integration bus destination. The first will create a SIDestination that exists only on the local service integration bus (and maybe localized to the ″local″ messaging engine depending on the localOnly flag). The second method is used to create a SIDestination that exists on a remote service integration bus.
-

For more information about the SIDestinationAddressFactory interface, see the SIDestinationAddressFactory generated API information.

Working with non-routing path fields in a message header:

This topic describes how to work with fields in a message header that identify and affect the behavior of messages.

About this task

In addition to the routing fields (see "Setting routing addresses in a message header" on page 886), there are a number of fields in the message header that you can work with. These fields affect important qualities and characteristics of the message, like priority and reliability, identity, and so on. See "Message header information" for information about the equivalence of the header fields to JMS message header fields, and the methods available to work with them.

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext **unless** you are only interested in the methods provided by MessageContext.
2. Get the SIMessage from the MessageContext object. For example, `SIMessage message = ((SIMessageContext)context).getSIMessage();`
3. Build your mediation header function in a similar way to these examples, using the reference information in "Message header information" to help:
   a. Set the reliability of the message. For instance, `siMessage.setReliability(Reliability.ASSURED_PERSISTENT);`. In this case, the quality of service is set to the highest level.
   b. Set the time to live for a message - that is, the time, in milliseconds, that the message is allowed to remain on a queue before it is removed if it is not processed. For example, `siMessage.setRemainingTimeToLive(1000000);` will set the remaining time before the message should expire to 1000 seconds.

Message header information:

This topic describes the mapping of the non-routing message property fields to JMS header fields, and the methods available to work with them.

Header fields

SIMessage header field	Field description	Corresponding JMS message header field	SIMessage methods
Priority (ReplyPriority)	Integer value 0-9, higher value is higher message priority	JMSPriority (integer)	• getPriority • setPriority • getReplyPriority • setReplyPriority
Reliability (ReplyReliability)	Specifies the reliability of message delivery. See Message reliability levels for a description of the allowed values.	JMSDeliveryMode (string) supports two levels of reliability: PERSISTENT and NON_PERSISTENT	• getReliability • setReliability • getReplyReliebility • setReplyReliability
TimeToLive (ReplyTimeToLive, RemainingTimeToLive)	An integer that represents the time in milliseconds that a message can remain on the queue before it expires.	JMSExpiration (long) is the time of expiry, calculated as ″current time″ plus (+) ″time-to-live″.	• getTimeToLive • getReplyTimeToLive • getRemainingTimeToLive • setTimeToLive • setReplyTimeToLive • setRemainingTimeToLive

SIMessage header field	Field description	Corresponding JMS message header field	SIMessage methods
Discriminator (ReplyDiscriminator)	A string that contains a topic name that is tested by a selector rule to determine if the message should be mediated.	No corresponding JMS field	• getDiscriminator • setDiscriminator • getReplyDiscriminator • setReplyDiscriminator
RedeliveredCount	Read-only field (integer) that holds that counts each time a message is re-delivered.	JMSRedelivered (Boolean) indicates that it is likely, but not guaranteed, that the message was delivered but unacknowledged in the past.	getRedeliveredCount
ApiMessageId	A string that uniquely identifies each message sent.	JMSMessageId (string)	• getApiMessageId • setApiMessageId
CorrelationId	A string that links two messages, typically linking a request message with its response.	JMSCorrelationId (string)	• getCorrelationId • setCorrelationId
UserId	A string that represents the identity of the user sending the message.	JMSX Userid is a message property not used by WebSphere Application Server.	• getUserId • setUserId

Working with the message payload

This topic describes how to work with the message payload in a pre-existing mediation handler, and transcode the message payload from one message format to another.

Before you begin

This task requires a mediation handler in an EJB project. For more information, see “Writing a mediation handler” on page 882. You should also read the tips for successfully programming mediations in the topic Coding considerations for mediations.

About this task

You can use this task to perform some or all of the following actions on the message payload:

- Locate the data objects within the message payload
- Convert the payload into another format
- Convert the payload into a byte array, for example if you want your mediation to log messages.

To work with the contents of a message, use the SIMessage and SIMessageContext APIs. Additionally, use SIMediationSession to provide your mediation with access to the service integration bus, to send and receive messages. For more information, see:

- SI programming resources
- “MediationHandler” on page 893
- “SIMessageContext” on page 893

To work with specific fields within a message, use SDO data graphs. For more information, see SDO data graphs. For more information about the format of supported message types, and examples of how to work with them, see “Mapping of SDO data graphs for Web services messages” on page 899.

To work with the message payload, take the following steps:

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext unless you only want to work with the methods provided by MessageContext.

2. Retrieve the data graph of the message payload as follows:
   a. Get the SIMessage from the MessageContext object. For example:
      ```
      SIMessage message = ((SIMessageContext)context).getSIMessage();
      ```
   b. Get the message format string to determine its type. For example:
      ```
      String messageFormat = message.getFormat();
      ```
   c. Retrieve the DataGraph object from the message. For example:
      ```
      DataGraph dataGraph = message.getDataGraph();
      ```

      For more information, see SDO data graphs.
3. Optional: Locate data objects within the payload:
   a. Navigate within the graph to a named DataObject. For example, where DataObject has the name ″data″:
      ```
      DataObject dataObject = dataGraph.getRootObject().getDataObject("data");
      ```
   b. Retrieve information contained in the data object. For example, if the message is a text message:
      ```
      String textInfo = dataObject.getString("value");
      ```
4. Work with the fields within the message. For an example of how to do this, see “Example code for message fields.”
5. Optional: Transcode the payload into another format:
   a. Review the topic “Transcoding between message formats” on page 895 to understand the implications of transcoding the payload.
   b. Call the method getNewDataGraph, passing the new format as a parameter, which returns a copy of the payload in the new format. For example:
      ```
      DataGraph newDataGraph = message.getNewDataGraph(newFormat);
      ```
   c. Write the data graph in the new format back to the message using the setDataGraph method. For example:
      ```
      message.setDataGraph(newDataGraph, newFormat);
      ```
6. Optional: Convert the payload into a stream of bytes:
   a. Review the topics “Transcoding a message payload into a byte array” on page 897 and “Transcoding a byte array into a message payload” on page 898 to understand the implications of converting between message format and byte stream, and back again.
   b. Call the method getDataGraphAsBytes, which returns a copy of the payload as a byte stream. For example:
      ```
      byte[] newByteArray = message.getDataGraphAsBytes();
      ```
   c. Call the method createDataGraph provided by the SIDataGraphFactory API which creates a new data graph by parsing the bytes according to the format passed to the method. For example:
      ```
      DataGraph newDataGraph = SIDataGraphFactory.getInstance().createDataGraph( byteArray, format);
      ```
   d. Work with the message as a stream of bytes. For an example of how to do this, see “Example code for message fields”
7. Return *True* in your mediation code so that the MessageContext is passed to the next mediation handler in the handler list. If the return value is *False* the MessageContext will be discarded and will not be delivered to the destination.

   Note: If your mediation handler is the last handler in the handler list, and the forward routing path is empty, the message is made available to consuming applications on that destination. If the forward routing path not empty, the message is not made available to any consumers on that destination. Instead, the message is forwarded to the next destination in the routing path.

Example code for message fields

Below is an example of the code for a mediation for working with a field in a message:

```
public boolean handle(MessageContext context) throws MessageContextException {

 /* Get the SIMessage from the MessageContext object */
 SIMessage message = ((SIMessageContext)context).getSIMessage();

 /* Get the message format string */
 String messageFormat = message.getFormat();

 /* If we have a JMS TextMessage then extract the text contained in the message. */
 if(messageFormat.equals("JMS:text"))
 {
  /* Retrieve the DataGraph object from the message */
  DataGraph dataGraph = message.getDataGraph();

  /* Navigate down the DataGraph to the DataObject named 'data'. */
  DataObject dataObject = dataGraph.getRootObject().getDataObject("data");

  /* Retrieve the text information contained in the DataObject. */
  String textInfo = dataObject.get("value");

  /* Use the text information retrieved */
  System.out.println(textInfo);
 }


 /* Return true so the MessageContext is passed to any other mediation handlers
 * in the handler list */
 return true;

 }
```

The complete mediation function code for working with the message payload as a stream of bytes might look like this example:

```
  public boolean handle(MessageContext context)throws MessageContextException {

  /* Get the SIMessage from the MessageContext object */
  SIMessage message = ((SIMessageContext)context).getSIMessage();

    if (!SIApiConstants.JMS_FORMAT_MAP.equals(msg.getFormat()))
    {
      try
      {
        dumpBytes(msg.getDataGraphAsBytes());
      }
      catch(Exception e)
      {
        System.out.println("The message contents could not be retrieved due to a "+e);
      }
    }
    else
    {
      System.out.println("The bytes for a JMS:map format message cannot be shown.");
    }

    return true;
  }

  private static void dumpBytes(byte[] bytes)
  {
    // Subroutine to dump the bytes in a readable form to System.out
  }
}
```

MediationHandler:

This topic describes the interface which defines the method invoked by the mediation runtime environment.

public interface MediationHandler. This interface defines the method which will be invoked by the mediation runtime environment.

The method handle invokes a mediation. It is called by the runtime when a message is to be mediated. The method returns Boolean *True* if the message passed into this method should continue along the handler list, otherwise *False*.

At the end of the handler list, the message is sent to the next destination on the routing path, unless the forward routing path is empty, when the message is made available to consuming applications on the current destination.

The API has just one method:

- handle: Method used by the runtime to invoke a mediation.

For more information about the MediationHandler interface, see the MediationHandler generated API information.

SIMessageContext:

This topic describes the interface which abstracts the message context processed by a message handler.

Public interface SIMessageContext extends javax.xml.rpc.handler.MessageContext.

This is the object that is required on the interface of a mediation handler. In addition to the context information that may be passed from one handler to another, it can return a reference to an SIMessage and an SIMediationSession. The SIMessage is the service integration technologies representation of the message being processed by the MediationHandler. The SIMediationSession is a handle to the run time resources.

The interface MessageContext abstracts the message context that is processed by a handler in the handle method. The MessageContext interface provides methods to manage a property set. MessageContext properties enable handlers in a handler chain to share processing related state.

As well as defining the method which will be invoked by the mediation runtime environment, the interface may also specify properties following the Enterprise JavaBeans naming pattern, or by providing a BeanInfo class. Each property of the bean will be initialized from a single environment entry with the same name as the property. Bean properties of simple type are specified using Java Platform, Enterprise Edition (Java EE) **env-entry**. If the handler has properties that are of non-simple type, then other environment definitions may be used.

The API has two methods:

- getSIMessage: Method to get the service integration bus representation of the message being mediated. Read more about the SIMessage API in “SIMessage.”
- getSession: Method to get an SIMediationSession object which is a handle to the core runtime environment. Read more about the SIMediationSession API in “SIMediationSession” on page 894.

SIMessage:

This topic describes the SIMessage interface; the public interface to a service integration bus message for use by mediations and other service integration bus components.

The public interface SIMessage extends java.lang.Cloneable and java.lang.Serializable.

The SIMessage interface has many methods allowing you to work with message properties, header contents, routing path, metadata, and others:

- The method getDataGraph returns the SDO data graph. This contains the SIMessage payload content in a tree representation. Using the data graph, you can work directly with individual fields in the message payload. For more information about SDO data graphs, see SDO data graphs.
- You can transcode a message payload by calling the method getNewDataGraph(format). It returns a copy of the payload in the new format. You can write the new datagraph back to the message using setDataGraph(DataGraph, format). For more information, see “Transcoding between message formats” on page 895.
- If you want to log a message as a simple byte stream, you can retrieve the message payload as a byte array using the method getDataGraphAsBytes. For more information about converting from data graph to bytes, and back again, see “Transcoding a message payload into a byte array” on page 897 and “Transcoding a byte array into a message payload” on page 898.
- There are methods to get, set, delete and clear user properties and message properties. You can also retrieve a list of user property names. For more information about working with properties, see “Working with the message properties” on page 885.
- Forward and reverse routing paths define a sequential list of intermediate bus destinations through which messages pass to reach a target bus destination. You use a routing path to apply the mediations configured on several destinations to the messages sent along the path. The following methods allow you to get and set the contents of the ForwardRoutingPath and ReverseRoutingPath for an SIMessage:
  - getForwardRoutingPath()
  - setForwardRoutingPath()
  - getReverseRoutingPath()
  - setReverseRoutingPath()

  For more information about routing paths, see Destination routing paths. For information about how to work with routing addresses, see “Setting routing addresses in a message header” on page 886.
- If your mediation changes the content of the message, there is a risk that the message is no longer valid. If the data graph is not valid, the message cannot be sent through the service integration bus or stored in the message store. In this case, the message is not **well formed**. A message is well formed when all the values of the message properties may be serialized, and the data graph of the message conforms to the format of the message. You can test your message using the method isWellFormed. It returns *true* when the message contains a well formed data graph. This test has implications for performance. For more information, see Setting tuning properties for a mediation.
- You can work with the time for the message to live, measured in milliseconds from the time when the message was originally sent:
  - The methods getTimeToLive and setTimeToLive allow you to get and set the value of the TimeToLive field in the message header. A value of 0 indicates that the message will never expire.
  - The methods getRemainingTimeToLive and setRemainingTimeToLive allow you to get the remaining time in milliseconds before the message expires, and set the remaining time in milliseconds before the message should expire.

For more information about SIMessage, see the API documentation.

SIMediationSession:

This topic describes the SIMediationSession interface which defines the methods for querying and interacting with the service integration bus. It also includes methods that provide information on where the mediation is being invoked from.

public interface SIMediationSession

As well as defining the methods for working with the service integration bus, this API also includes methods that provide information on where the mediation is invoked from, and the criteria that are applied before the message is mediated.

Both selector and discriminator control which messages are sent to the mediation, through a rule specified in a text string. The rule specified by the selector examines the header and properties of the message, while the discriminator examines the topic of the message. If a message contains both selector and discriminator, it must match both rules for the message to be mediated. If either the selector or the discriminator rule does not match, the message is not mediated.

The API has these methods:

- getBusName returns the name of the bus upon which the mediation is associated.
- getDestinationName returns the name of the destination with which the mediation is associated.
- getDiscriminator returns the discriminator that is defined in the mediation definition.
- getMediationName returns the name of the mediation that is being executed.
- getMessageSelector returns the message selector that is defined in the mediation definition.
- getMessagingEngineName returns the name of the messaging engine from which the mediation was invoked
- getSIDestinationConfiguration returns the SIDestinationConfiguration object associated with the destination, specified by destinationName or destinationAddress.
- receive receives an SIMessage from the service integration bus. There are four variants.
- resetIdentity changes the identity of the given message to the current run-as identity.
- send sends a copy of an SIMessage to the service integration bus, in addition to the message returned by the message interface.

See also the generated API information for SIMessageContext.

Transcoding between message formats:

A mediation can convert a message from one format to another without changing the semantic meaning of the message. This operation is referred to as transcoding a message.

The following code is an example mediation handler that transcodes a message into a new message format, providing that the message can be transcoded:

```
private static final String NEW_FORMAT = "JMS:text";

public boolean(MessageContext context) throws MessageContextException
{
  try
  {
    SIMessageContext msgCtx  = (SIMessageContext)context;
    SIMessage        msg     = msgCtx.getMessage();
    DataGraph        newDg   = msg.getNewDataGraph(NEW_FORMAT);

    msg.setDataGraph(newDg,NEW_FORMAT);
    return true;
  }
  catch(Exception e)
  {
    // Reroute the original message to the exception destination
    MessageContextException mce =
      new MessageContextException("Unable to transcode to "+NEW_FORMAT",e);
    throw mce;
  }
}
```

The table below describes which messages can be transcoded, and gives the outcome for each format pairing. Note that the abbreviation DG represents ″data graph″. The numbers within brackets in the table are explained as follows:

- (1) A message with format JMS: cannot have a payload. It does not carry any message data other than the message properties. If a mediation calls `getDataGraph()` on a message with format JMS:, `null` is always returned. All other message formats must have a message payload. This means that a message with format JMS: cannot be transcoded into another format. If a mediation needs to change a message with format JMS: into a message with any other format, the mediation needs to call the methods `SIDataGraphFactory.getInstance().createDataGraph(newFormat)` and `setDataGraph` on the SIMessage object to change the message contents.
- (2) `null` is always returned if a mediation calls `getDataGraph()` on a message with format JMS:
- (3) A mediation can call the method getNewDataGraph() on a message to return a copy datagraph with the same format as the message. The copy can be edited, leaving the original message unchanged. For SOAP and Beans, you can change the message model by editing the format string to change the value that follows the ":".

	To JMS:	**To JMS:text**	**To JMS:bytes**	**To JMS:stream**	**To JMS:object**	**To SOAP:**	**To Bean:**
From JMS:	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)
From JMS:text	DG=null (2)	Yes (3)	Yes, bytes contain UTF-8	Yes, if text contains XML that conforms to the correct schema.	No	Yes, if message content is valid SOAP.	Yes, if message content is valid SOAP.
From JMS:bytes	DG=null (2)	Yes, but only when the bytes can correctly be interpreted as a UTF-8 string.	Yes (3)	Yes, if bytes contain XML that conforms to the correct schema.	Yes, assume that bytes are a serialized object.	Yes, if message content is valid SOAP.	Yes, if message content is valid SOAP.
From JMS:stream	DG=null (2)	Yes, text is XML transcoding.	Yes, bytes contain XML transcoding.	Yes (3)	No	No	No
From JMS:object	DG=null (2)	No	Yes, bytes contain the object serialization.	No	Yes (3)	No	No
From SOAP:	DG=null (2)	Yes	Yes	No	No	Yes (3) - if message content matches the new WSDL.	Yes
From Bean:	DG=null (2)	Yes	Yes	No	No	Yes	Yes (3) - if message content matches the new WSDL.

XML schema definition for JMS stream messages:

This is the XML schema definition for transcoding JMS stream messages to message types.

The following XML schema definition uses the target namespace http://www.ibm.com/xmlns/prod/websphere/messaging/jms/ to express JMS stream messages in XML. Use this definition to transcode between a byte array and a message payload.

```
<xsd:schema elementFormDefault="qualified" xml:lang="EN"
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/messaging/jms"
  xmlns="http://www.ibm.com/xmlns/prod/websphere/messaging/jms"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="data" type="StreamBody"/>

  <xsd:complexType name="StreamBody">
    <xsd:sequence>
      <xsd:element name="value"
        type="streamTypes"
        minOccurs="0"
        maxOccurs="unbounded"
        nillable="true"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="character">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/>
      <xsd:maxLength value="1"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="streamTypes">
   <xsd:union memberTypes="xsd:long xsd:int xsd:short xsd:byte xsd:boolean
            xsd:float xsd:double xsd:string xsd:hexBinary character"/>
  </xsd:simpleType>
</xsd:schema>
```

Transcoding a message payload into a byte array:

You can transcode the message payload into a byte array.

For example, you might want to write a mediation handler that logs a message as a simple byte stream. You can retrieve the message payload as a byte array using the method getDataGraphAsBytes. The table below describes the rules for transcoding an SIMessage data graph into a byte array.

Datagraph format	Pre-conditions	Outcome	Character set encoding
JMS:	None	Returns null.	Not applicable.
JMS:text	None	Returns the result of java.lang.String:getBytes(String charSetName) when applied to the `data/value` element of the graph, where charSetName = “UTF-8″	UTF-8
JMS:bytes	None	Returns a copy of the value of the `data/value` element of the data graph for the message.	Not applicable.
JMS:stream	None	Returns a byte buffer containing an XML serialization of the stream message according to the XML schema for stream messages.	UTF-8
JMS:object	None	Returns a copy of the value of the `data/value` element of the data graph for the message.	Not applicable.

Datagraph format	Pre-conditions	Outcome	Character set encoding
SOAP:	If the byte array must be generated by this operation (instead of using an existing byte array available through lazy parsing) then the data graph must be valid with respect to the WSDL model.	Returns a byte buffer containing a SOAP serialization of the data graph. If the SOAP message contains an attachment, the buffer has the multipart MIME format.	Either UTF-8, or that of the source message for the graph, where logically equivalent to the graph state.
Bean:	The data graph must be valid with respect to the WSDL model. In the absence of a SOAP binding the serialization will be performed using RPC/literal encoding.	Returns a byte buffer containing a SOAP serialization of the data graph. If the Bean contains attachments then the buffer will be in multipart MIME format.	UTF-8

Transcoding a byte array into a message payload:

A mediation can transcode a byte array into a message payload without changing the meaning of the message.

A mediation can reconstruct the message payload from a byte array, for example after logging a message. The table below describes the rules for transcoding a byte array into an SIMessage data graph.

Format argument	Pre-conditions	Outcome
JMS:	None	Returns null
JMS:text	`java.lang.String(inputBytes, "UTF-8")` does not result in an exception.	Returns new data graph instance of format JMS:text. Value of graph at path `data/value` has value equal to `java.lang.String(inputBytes, "UTF-8")`.
JMS:bytes	`inputBytes` is not null.	Returns new data graph instance of format JMS:bytes. Value of graph at path `data/value` is a copy of the `inputBytes` byte array.
JMS:stream	Byte array is XML, and is valid with respect to the `JmsStreamBody` type of the XML schema definition.	Returns new data graph instance of format JMS:stream. Value of graph at path `data/value` has type List, containing a sequence of simple typed values according to the types and values of each of the elements in the XML document.
JMS:object	Not null **Note:** You must ensure that the byte array is a valid serialized object.	Returns new data graph instance of format JMS:object. Value of graph at path `data/value` is a copy of the `inputBytes` byte array.
SOAP:	The byte buffer contains valid SOAP with respect to the associated WSDL model.	Returns new data graph with type system defined by the WSDL referenced by the byte buffer, and values of the graph defined by the SOAP payload.
Bean:	The byte buffer contains valid Bean with respect to the associated WSDL model.	Returns new data graph with type system defined by the WSDL referenced by the byte buffer, and values of the graph defined by the Bean payload.

Web services overview:

This topic describes how to work with the abstract data graph form of Web services messages.

The format of Web services messages

WebSphere Application Server (base) supports two formats for Web services messages: SOAP and enterprise beans (similar to Java APIs for XML based RPC, or JAX-RPC).

The information you need to work with Web services messages is in three parts:

- The structure of the SDO data graphs for Web services messages. See “Mapping of SDO data graphs for Web services messages” for more information about the data elements and the shape of the data graph.
- Reference information to help you develop code to navigate the data graphs of the messages that your program mediates. See “Mapping XML schema definitions to the SDO type system” on page 903.
- For XML representations of the shape of each part of Web services messages, sample code snippets and further information about the data graph format, see:“Web Services code example” on page 906.

Format types

The Web services message type is defined by a message format string within the message. The format string is prefixed with a domain identifier, either SOAP or Bean, followed by four comma separated fields as shown below:

```
SOAP:<wsdlLocation>,<serviceNameSpace>,<serviceName>,<portName>
Bean:<wsdlLocation>,<serviceNameSpace>,<serviceName>,<portName>
```

The fields are described in the following table:

Field name	Message format string	Field description
WSDL location	`<wsdlLocation>`	The URI where the WSDL for this message is located. The WSDL is deployed to the SDO repository using this location as the key.
Service namespace	`<serviceNamespace>`	Service namespace and Service name uniquely identify the Service definition within the WSDL.
Service name	`<serviceName>`	Service name and Service namespace uniquely identify the correct Service definition within the WSDL.
Port name	`<portName>`	Locates the Port definition within the Service, giving the PortType and Binding information required for message processing.

Mapping of SDO data graphs for Web services messages:

This topic describes the layouts for the different parts of Web services messages

Overall Web service message layout

The Info node is the top of the graph for all Web services messages. It has these properties and associated types:

Property name	Property type	Property description

operationName	`java.lang.String`	Identifies the WSDL operation the message is associated with. If the data access service cannot identify the message this field may be null. See “Identifying Web services messages”
messageName	`java.lang.String`	Identifies the WSDL message this message is associated with. If the data access service cannot identify the message this field may be null. See “Identifying Web services messages”
messageType	`java.lang.String`	Identifies WebService type of message instance. The field can have the values *input*, *output*, *fault*, *ambiguous*. If the data access service cannot identify the message this field may be null. See “Identifying Web services messages”
headers	`java.util.List` of data objects.	Contains a list of header entry data objects. Each SOAP header in the message results in a header entry in this list. See “Message header layout ” on page 901
attachments	`java.util.List` of data objects.	Contains a list of attachment entry data objects. In SOAP messages with attachments, each MIME part in the message (except the MIME part containing the SOAP envelope) is mapped to an entry in this list. See “Message attachment layout” on page 902
body	`commonj.sdo.DataObject`	A nested data object, which represents the body of the SOAP Envelope. See “Message body layout” on page 903

In addition to the format string, the message is described by the three metadata fields, operationName, messageName, and messageType. The payload of the message is split across the three other sections: headers, attachments and the body. These follow a section on the identification of messages.

Identifying Web services messages

Processing of messages depend on whether or not they have WSDL definitions. The minimum amount of information required for processing without WSDL is ″SOAP:″ The minimum amount of information required for processing with WSDL is: ″SOAP:location,namespace,service,port″. If the format string does not include all five of these fields, the SOAP data access service will attempt to process the message without WSDL.

- **Processing messages without WSDL definitions:** If the format string does not include full WSDL information, the SOAP data access service processes the message without attempting to match the message against definitions in WSDL. As a result, operationName and messageName are set to null, and the messageType field is only set when processing a fault message.
- **Processing messages with WSDL definitions:** If the format string includes <WSDL location>,<Service namespace>,<Service name>, and <Port name> then the SOAP and Beans data access services process the message using the WSDL definitions of the service.

Note: SOAP message processing will fail after supplying all the required WSDL information,
- if the SOAP data access service fails to locate the WSDL
- if the WSDL fails to corroborate the message.

When the SOAP data access service processes a SOAP request or reply message, it tries to match it against the message definitions in the WSDL. Normally there is unique match, and the operationName, messageName, and messageType are filled in appropriately. If there is more than one possible match the data access service picks a message definition, and fills in the operationName and messageName. In this case the messageType is set to *ambiguous*.

When processing fault messages, identification is slightly different. In all cases the messageType will be set to *fault*. If the message matches a unique fault definition in the WSDL then the operationName and messageName properties will also be set.

Message header layout

The list of headers can have two types of entry, depending on whether the header is based on part of the message or not.

The first type is used to handle headers that are not parts of the message:
- either not modeled in WSDL,
- or modeled in WSDL but not based on a part of the message.

For a model of this header, see “Header entry”

The second type of entry is used when the SOAP binding for the message has bound a part of the body into a MIME attachment. (This occurs when you use a <MIME:content> element to bind a part of the message to an attachment.) For consistent mediation programming, all of the body data is stored in the body node in the graph. In place of the normal attachment entry a bound attachment entry is placed into the attachments list. The bound attachment entry contains the MIME meta-data for the attachment, and for completeness also contains the name of the message part that contains the data taken from this attachment. This allows mediations designed to process attachments to locate the data in the body part of the graph. For a model of this attachment see “Bound header entry.”

Header entry

Property name	Property type	Property description
mustUnderstand	`java.lang.Boolean`	Carries the value from the mustUnderstand attribute on the SOAP header, if present.
actor	`java.lang.String`	Carries the value from the actor attribute on the SOAP header, if present.
any	`commonj.sdo.Sequence`	Container for the contents of the SOAP Header.

Bound header entry

Property name	Property type	Property description
mustUnderstand	`java.lang.Boolean`	Carries the value from the mustUnderstand attribute on the SOAP header, if present.

actor	`java.lang.String`	Carries the value from the actor attribute on the SOAP header, if present.
messagePart	`java.lang.String`	Contains the name of the message part which carries the data from this message header.

Message attachment layout

Message attachments are handled in a similar way to headers, and instances of them populate the attachments list in the Info node.

There are two types of attachment entry to handle MIME attachments. The first is for general attachments: see “Attachment entry”

The second type of attachment entry includes <MIME:content> elements that bind a part of the body into a MIME attachment. If you are programming a mediation, you need to know how to locate the data within the graph. For consistent mediation programming, the attachment data is placed in the message body, referred to by the part name in the header entry, which includes the other MIME metadata. For a model of this attachment, see “Bound attachment entry.”

Attachment entry

Property name	Property type	Property description
contentType	`java.lang.String`	Carries the contentType from the MIME part that is represented by the attachment entry.
contentTransferEncoding	`java.lang.String`	Carries the contentTransferEncoding from the MIME part that is represented by the attachment entry.
contentId	`java.lang.String`	Carries the contentId from the MIME part that is represented by the attachment entry.
data	`byte[]`	Carries the content of the MIME element, as a byte array.

Bound attachment entry

Property name	Property type	Property description
contentType	`java.lang.String`	Carries the contentType from the MIME part that is represented by the attachment entry.
contentTransferEncoding	`java.lang.String`	Carries the contentTransferEncoding from the MIME part that is represented by the attachment entry.
contentId	`java.lang.String`	Carries the contentId from the MIME part that is represented by the attachment entry.
messagePart	`java.lang.String`	Contains the name of the message part which carries the data from this attachment.

Message body layout

The layout of the data object in the body is defined by the service WSDL. The type of the data object is derived from the message definition in the WSDL. The data object will have one property for each part in the message definition. The layout of each message part follows the convention for mapping XML Schema into SDO, see “Web Services code example” on page 906 for more information.

Web services fault message

If the message is a fault message, the messageType field (in the Info node of the graph) will be set to ″fault″, and the message body will have the following properties:

Property name	Property type	Property description
faultcode	`javax.xml.namespace.QName`	Carries the faultcode value from the SOAP Fault element
faultstring	`java.lang.String`	Carries the faultstring value from the SOAP Fault element
faultactor	`java.lang.String`	Carries the faultactor value from the SOAP Fault element
detail	`commonj.sdo.DataObject`	Carries the content within the detail child of the SOAP Fault Element

Note: As the detail element definition uses element and attribute wildcards, the content of the detail data object will contain a Sequence. See “Web Services code example” on page 906 for more information.

Mapping XML schema definitions to the SDO type system:

Use this reference information to help you develop code to navigate the data graphs of the messages that your program mediates.

XML schemas might be embedded in the WSDL sections that describe the message parts and SOAP headers. However the SOAP header description is more likely to be available as a separate schema, in which case you should load it into the SDO repository where it can be used to process any message with a matching header at run time.

Schema to Java class mapping

Each XML schema complex type is mapped to an SDO type. This means that an element with a complex type will be represented by an instance of an SDO data object. The type has a property for each element, attribute, or wildcard that is contained in the schema type definition.

In turn, the instance will contain a value for each property that has been set. If the property is mapped from a schema complex type then the value will be another SDO data object. If the property is mapped from a schema simple type then the value will be an instance of a Java class, as shown in the following table.

Schema type	Java class	Notes
anyURI	java.lang.String	
base64Binary	byte[]	See note 2
boolean	java.lang.Boolean/ boolean	See note 1
byte	java.lang.Byte / byte	See note 1
date	java.lang.String	

dateTime	java.lang.String	
decimal	java.math.BigDecimal	
double	java.lang.Double / double	See note 1
duration	java.lang.String	
ENTITIES	java.util.List	
ENTITY	java.lang.String	
float	ava.lang.Float / float	See note 1
gDay	java.lang.String	
gMonth	java.lang.String	
gMonthDay	java.lang.String	
gYear	java.lang.String	
gYearMonth	java.lang.String	
hexBinary	byte[]	See note 2
ID	java.lang.String	
IDREF	java.lang.String	
IDREFS	java.util.List	
int	java.lang.Integer / int	See note 1
integer	java.math.BigInteger	
language	java.lang.String	
long	java.lang.Long / long	See note 1
Name	java.lang.String	
NCName	java.lang.String	
negativeInteger	java.math.BigInteger	
NKTOKENS	java.util.List	
NMTOKEN	java.lang.String	
nonNegativeInteger	java.math.BigInteger	
nonPositiveInteger	java.math.BigInteger	
normalisedString	java.lang.String	
NOTATION	javax.xml.namespace.QName	
positiveInteger	java.math.BigInteger	
QName	javax.xml.namespace.QName	
short	java.lang.Short / short	See note 1
string	java.lang.String	
time	java.lang.String	
token	java.lang.String	
unsignedByte	java.lang.Short / short	See note 1
unsignedInt	java.lang.Long / long	See note 1
unsignedLong	java.math.BigInteger	
unsignedShort	java.lang.Integer / int	See note 1

Note:

1. SDO automatically converts primitives (int, long and so on) into objects as needed. This means that you can use a mixture of the specialized methods (getInt, setInt, getLong, setLong) as well as the generic get and set methods.
2. As byte arrays are mutable, it is possible to update the value without setting it back onto the data object. However when this occurs the data object may not be aware of implicit update. When working with byte array values you should always use the setBytes() method to explicitly update the data object.

Working with global elements and attributes

When a schema is mapped to SDO we also define a special SDO type, typically called 'DocumentRoot'. This type is a container for all the global elements and attributes in the schema. Whenever you need to locate an SDO property for a global element or attribute you should locate the 'DocumentRoot' type and then locate the appropriate property within it.

The following schema defines the layout of Web services messages. By comparing this schema with the information in "Mapping of SDO data graphs for Web services messages" on page 899 you can see the schema to SDO mapping in action.

```
<?xml version="1.0"?>
<xsd:schema
    targetNamespace="http://www.ibm.com/ns/2004/05/webservices/messagemodel"
    xmlns:tns="http://www.ibm.com/ns/2004/05/webservices/messagemodel"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

 <xsd:import namespace="http://schemas.xmlsoap.org/soap/envelope/"/>

    <xsd:element name="Info">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="operationName" nillable="true" type="xsd:string"/>
          <xsd:element name="messageName"   nillable="true" type="xsd:string"/>
          <xsd:element name="messageType"   nillable="true" type="xsd:string"/>
          <xsd:element name="headers"       type="tns:HeaderEntryType"     maxOccurs="unbounded"/>
          <xsd:element name="attachments"   type="tns:AttachmentEntryType" maxOccurs="unbounded"/>
          <xsd:element name="body"          type="tns:BodyType"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="BodyType" abstract="true"/>

    <xsd:complexType name="HeaderEntryType" abstract="true"/>

    <xsd:complexType name="AttachmentEntryType" abstract="true"/>

    <xsd:complexType name="SOAPFaultBody">
      <xsd:complexContent>
        <xsd:extension base="tns:BodyType">
          <xsd:sequence>
            <xsd:element name="faultcode" type="xsd:QName"/>
            <xsd:element name="faultstring" type="xsd:string"/>
            <xsd:element name="faultactor" type="xsd:anyURI" minOccurs="0"/>
            <xsd:element name="detail" type="soap:detail" minOccurs="0"/>
    </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="SOAP_1_1_HeaderEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:HeaderEntryType">
          <xsd:sequence>
            <xsd:element name="mustUnderstand" nillable="true" type="xsd:boolean"/>
            <xsd:element name="actor"          nillable="true" type="xsd:anyURI"/>
            <xsd:any/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
```

```
    <xsd:complexType name="SOAP_1_1_BoundHeaderEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:HeaderEntryType">
          <xsd:sequence>
            <xsd:element name="mustUnderstand" nillable="true" type="xsd:boolean"/>
            <xsd:element name="actor"          nillable="true" type="xsd:anyURI"/>
            <xsd:element name="messagePart"    type="xsd:string"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="MIMEAttachmentEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:AttachmentEntryType">
          <xsd:sequence>
            <xsd:element name="contentType"             type="xsd:string"/>
            <xsd:element name="contentTransferEncoding" type="xsd:string"/>
            <xsd:element name="contentId"               type="xsd:string"/>
            <xsd:element name="data"                    type="xsd:base64Binary"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="BoundMIMEAttachmentEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:AttachmentEntryType">
          <xsd:sequence>
            <xsd:element name="contentType"             type="xsd:string"/>
            <xsd:element name="contentTransferEncoding" type="xsd:string"/>
            <xsd:element name="contentId"               type="xsd:string"/>
            <xsd:element name="messagePart"             type="xsd:string"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="UnknownBodyType">
      <xsd:complexContent>
        <xsd:extension base="tns:BodyType">
          <xsd:sequence>
            <xsd:any/>
          </xsd:sequence>
          <xsd:attribute name="encodingStyle" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

Web Services code example:

This topic contains example WSDL and code snippets to show how to access fields within a Web services message for programming a mediation.

Web services message definition

This topic contains an example of a Web services message. It is characterized in Web Services Description Language (WSDL), an XML-based language used to describe the services a business offers and how those services may be accessed.

Based upon this Web service, the rest of the topic shows how to program mediations to work with different parts of the message (described with the SDO representation in “Mapping of SDO data graphs for Web services messages” on page 899.) For each part of the message, you will see an XML description of the message, representing its SDO data graph. To accompany each XML description, you will see some snippets of code that illustrate how to work with that part of the message.

Note that in the following example the SOAP header schema is included in the WSDL. It could alternatively have been included as a separate schema in the SDO repository.

Here is the WSDL description of the message that is used as an illustration for the subsequent code snippets:

companyInfo Web service message description

```
<wsdl:definitions targetNamespace="http://example.companyInfo"
 xmlns:tns="http://example.companyInfo"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:wsdlmime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
  <xsd:schema elementFormDefault="qualified"
   targetNamespace="http://example.header">

   <xsd:element name="sampleHeader">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name="priority" type="xsd:int"/>
     </xsd:all>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>


  <xsd:schema elementFormDefault="qualified"
   targetNamespace="http://example.companyInfo">

   <xsd:element name="getCompanyInfo">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name="tickerSymbol" type="xsd:string"/>
     </xsd:all>
    </xsd:complexType>
   </xsd:element>

   <xsd:element name="getCompanyInfoResult">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name="result" type="xsd:float"/>
     </xsd:all>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>


 </wsdl:types>

 <wsdl:message name="getCompanyInfoRequest">
   <wsdl:part name="part1" element="tns:getCompanyInfo"/>
 </wsdl:message>

 <wsdl:message name="getCompanyInfoResponse">
  <wsdl:part name="part1" element="tns:getCompanyInfoResult"/>
  <wsdl:part name="part2" type="xsd:string"/>
  <wsdl:part name="part3" type="xsd:base64Binary"/>
 </wsdl:message>

 <wsdl:portType name="CompanyInfo">
  <wsdl:operation name="GetCompanyInfo">
   <wsdl:input message="tns:getCompanyInfoRequest"
               name="getCompanyInfoRequest"/>
```

```
   <wsdl:output message="tns:getCompanyInfoResponse"
                name="getCompanyInfoResponse"/>
  </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="CompanyInfoBinding" type="tns:CompanyInfo">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="GetCompanyInfo">
   <wsdlsoap:operation soapAction=""/>
   <wsdl:input name="getCompanyInfoRequest">
    <wsdlsoap:body use="literal"/>
   </wsdl:input>
   <wsdl:output name="getCompanyInfoResponse">
    <wsdlsoap:body use="literal"/>
   </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="CompanyInfoService">
  <wsdl:port binding="tns:CompanyInfoBinding" name="SOAPPort">
   <wsdlsoap:address location="http://somewhere/services/CompanyInfoService"/>
  </wsdl:port>
 </wsdl:service>

</wsdl:definitions>
```

Working with the info node

This is an example of a simple SOAP request:

```
<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
```

You can access the properties of the info node (see “Overall Web service message layout” on page 899) using code snippets like this:

```
 // Get the info node (a child of the graph's root object)
 DataObject rootNode = graph.getRootObject();
 DataObject infoNode = rootNode.getDataObject("Info");

  // Query the operationName, and messageType.
  String opName = infoNode.getString("operationName");
  String type   = infoNode.getString("messageType");
```

Working with a header

This is an example of a SOAP request including a header:

```
<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Header>
        <example:sampleHeader
          env:mustUnderstand='1'
          xmlns:example='http://example.header'>
            <example:priority>4</example:priority>
        </example:sampleHeader>
```

```
    </env:Header>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
```

You can see the properties of the header entry with a list of headers in “Header entry” on page 901. You can work with a header entry and its properties using code like this:

```
   // Get the info node (a child of the graph's root object)
    DataObject rootNode = graph.getRootObject();
    DataObject infoNode = rootNode.getDataObject("Info");

    // Access the list of headers
    List headerEntries = infoNode.getList("headers");

    // Get the first entry from the list
    DataObject headerEntry = (DataObject) headerEntries.get(0);

    // Query the mustUnderstand property of the header entry
    boolean mustUnderstand = headerEntry.getBoolean("mustUnderstand");

    // Get the Sequence which holds the content of the header entry
    Sequence headerContent = headerEntry.getSequence("any");

    // Get the first piece of content from the Sequence
    DataObject header = (DataObject) headerContent.getValue(0);

    // Read the priority from the header
    int priority = header.getInt("priority");

    // Shorthand for the above, using SDO path expressions that start
    // from the info node.
    mustUnderstand = infoNode.getBoolean("headers[1]/mustUnderstand");
    priority       = infoNode.getInt("headers[1]/any[1]/priority");
```

Working with an attachment

This is an example of a SOAP request including an XML attachment:

```
Content-Type: multipart/related; start="<start>"; boundary="boundary"


--boundary
Content-Type: text/xml
Content-Transfer-Encoding: 7bit
Content-ID: <start>

<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
--boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-ID: <myAttachment>

<info>Some attached information</info>
--boundary--
```

You can see the properties of the attachment entry with byte array in “Attachment entry” on page 902. You can work with a header entry and its properties using code like this:

```
// Get the info node (a child of the graph's root object)
  DataObject rootNode = graph.getRootObject();
  DataObject infoNode = rootNode.getDataObject("Info");

  // Access the list of attachments
  List attachmentEntries = infoNode.getList("attachments");

  // Get the first entry from the list
  DataObject attachmentEntry = (DataObject) attachmentEntries.get(0);

  // Query the contentId property of the header entry
  String contentId = attachmentEntry.getString("contentId");

  // Get the data contained in the attachment
  byte[] data = attachmentEntry.getBytes("data");
```

Working with the message body

This is an example of a simple SOAP request:

```
<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
```

You can see the properties of the body in “Message body layout” on page 903. You can work with the contents of the body using code like this:

```
    // Get the info node (a child of the graph's root object)
    DataObject rootNode = graph.getRootObject();
    DataObject infoNode = rootNode.getDataObject("Info");

    // Get hold of the body node
    DataObject bodyNode = infoNode.getDataObject("body");

    // Get hold of the data object for the first part of the body
    DataObject part1Node = bodyNode.getDataObject("part1");

    // Query the tickerSymbol
    String ticker = part1Node.getString("tickerSymbol");

    // Shorthand for the above, using a SDO path expression that
    // starts from the info node.
    ticker = infoNode.getString("body/part1/tickerSymbol");
```

JMS formats:

This topic directs you to the information you need to access the different types of JMS message.

Format types

Service integration technologies supports four different types of JMS message. Each message type is defined by a message format string within the message. You can retrieve the format string using the code snippet in the example below. The format string will be one of the following:

JMS Message type	Message format string	Mapping to SDO

JMS Bytes message	`JMS:bytes`	See “JMS Formats -- bytes”
JMS Text message	`JMS:text`	See “JMS Formats -- text”
JMS Stream message	`JMS:stream`	See “JMS formats -- Stream”
JMS Object message	`JMS:object`	See “JMS Formats -- object” on page 912

This code snippet is an example of how to retrieve the message format string from the message:

```
String format = siMsg.getFormat();
if (format.equals ....
```

JMS Formats -- bytes:

This topic contains reference information you can use to map from the body of a JMS bytes message to SDO:

Bytes body

You can retrieve the payload of a JMS bytes message as a Java byte array (`byte[]`). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named ″data″, and that data object in turn contains a property named ″value″. In JMS bytes messages, the value property may be accessed as a Java byte array.

You can access the data within the data graph with code like this:

```
SIMessage siMsg;
String format = siMsg.getFormat();
if (format.equals("JMS:bytes")) {
  DataGraph graph = siMsg.getDataGraph();
  byte[] payload = graph.getRootObject().getBytes("data/value");
}
```

JMS Formats -- text:

This topic contains reference information you can use to map from the body of a JMS text message to SDO:

Text body

You can retrieve the payload of a JMS text message as a Java string value (java.lang.String). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named ″data″, and that data object in turn contains a property named ″value″. In JMS text messages the value property may be accessed as a Java string value.

You can access the data within the data graph with code like this:

```
SIMessage siMsg;
String format = siMsg.getFormat();
if (format.equals("JMS:text")) {
  DataGraph graph = siMsg.getDataGraph();
  String payload = graph.getRootObject().getString("data/value");
}
```

JMS formats -- Stream:

This topic contains reference information you can use to map from the body of a JMS Stream message to SDO.

Stream body

You can retrieve the payload of a JMS Stream message as a Java list value (java.util.List). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named ″data″, and that data object in turn contains a property named ″value″. In the case of a JMS Stream message the value property may be accessed as a List value. The member functions of the List interface can be used to access the individual objects within the JMS Stream message instance. (Note that the JMS standard places constraints on the kinds of objects which may be placed in a Stream message.)

You can access the data within the data graph with code like this:

```
}SIMessage siMsg;
String format = siMessage.getFormat();
if (format.equals("JMS:stream")) {
  DataGraph graph = siMsg.getDataGraph();
  List payload = graph.getRootObject().getList("data/value");
  int streamLength = payload.size();
  if (streamLength > 0) {
    Object item1 = payload.get(0);
    // You can also access items directly, for example: (for the_same_ value)
    item1 = graph.getRootObject().get("data/value[1]");
  }
}
```

JMS Formats -- object:

This topic contains reference information you can use to map from the body of a JMS object message to SDO:

Object body

You can retrieve the payload of a JMS object message as a Java byte array (`byte[]`). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named ″data″, and that data object in turn contains a property named ″value″. In the case of a JMS object message the value property may be accessed as a Java byte array. The original Object instance which the payload represents may be reconstructed from the byte array.

You can access the data within the data graph with code like this:

```
SIMessage siMsg;
String format = siMsg.getFormat();
if (format.equals("JMS:object")) {
  DataGraph graph = siMsg.getDataGraph();
  byte[] payload = graph.getRootObject().getBytes("data/value");
  if(payload != null) {
    // Need to deserialize to recover original object
    ObjectInputStream in =
      new ObjectInputStream(new ByteArrayInputStream(payload));
    Object obj = in.readObject();
  }
}
```

Writing a routing mediation

Use this topic to create a mediation that chooses a particular forward route for a message.

Before you begin

For an introduction to using mediations with the service integration bus, see Learning about mediations. For details of how to install a mediation into WebSphere Application Server and associate it with a bus destination, see Working with mediations.

This topic assumes that you are familiar with using a Java Platform, Enterprise Edition (Java EE) session bean development environment such as the assembly tools or IBM Rational Application Developer.

About this task

A routing mediation is a mediation application that contains a routing handler. You associate a routing mediation with a service integration bus destination, and use the mediation to choose a particular route from a range of available routes. For example when you create a new outbound service configuration or modify an existing outbound service configuration you can apply a port selection mediation to choose a particular outbound port from the range of ports that are available to the outbound service.

To create a routing mediation, use a Java Platform, Enterprise Edition (Java EE) session bean development environment to complete the following steps:

1. Create an empty mediation handler project. This creates the project, and creates the handler class that implements the handler interface. For detailed instructions on how to do this, see Writing the mediation handler.
2. Use the mediation pane on the EJB descriptor to define the handler class as a mediation handler.

   Note: When you do this, you specify a name by which the mediation handler list is known. Make a note of this name, for later reference when you create the mediation in the bus.
3. Add the routing function to the handler. Before you begin, review Adding mediation function to handler code, in particular its subtopic Working with message context. Add import statements to your handler class, and modify the handle method by adding your routing code. Specify the routing destination by adding that destination to the front of the forward routing path list. The forward routing path list is available from the message context. For example:

```
import javax.xml.rpc.handler.MessageContext;
import com.ibm.websphere.sib.mediation.handler.MediationHandler;
import com.ibm.websphere.sib.mediation.handler.MessageContextException;
import com.ibm.websphere.sib.mediation.messagecontext.SIMessageContext;
import com.ibm.websphere.sib.SIMessage;
import com.ibm.websphere.sib.SIDestinationAddress;
import com.ibm.websphere.sib.SIDestinationAddressFactory;
import java.util.List;
public class RouteMediationHandler implements MediationHandler {

 public boolean handle(MessageContext ctx) throws MessageContextException {
  SIMessageContext siCtx = (SIMessageContext) ctx;
  SIMessage msg = siCtx.getSIMessage();
  List frp = msg.getForwardRoutingPath();
  try {
   SIDestinationAddress destination =
    SIDestinationAddressFactory
     .getInstance()
     .createSIDestinationAddress(
     "RoutingDestination", //this is the name of the target destination
     false);
   frp.add(0, destination);
  } catch (Exception e) {
   return false;
  }
  msg.setForwardRoutingPath(frp);
```

```
  return true;
 }

}
```

For more information on the service integration technologies classes, including the mediation handler and message context classes, see the Generated API documentation - Application programming interfaces .

4. Export the routing mediation enterprise application.

What to do next

You are now ready to install your mediation into WebSphere Application Server and associate it with a bus destination, as described in Working with mediations.

Writing a mediation that maps between attachment encoding styles

Use this topic to create a mediation that maps from SOAP Messages with Attachments encoding style to WS-I Attachments Profile Version 1.0 encoding style.

Before you begin

For an introduction to using mediations with the service integration bus, see Learning about mediations. For details of how to install a mediation into WebSphere Application Server and associate it with a bus destination, see Working with mediations.

This topic assumes that you are familiar with using a Java Platform, Enterprise Edition (Java EE) session bean development environment such as the assembly tools or IBM Rational Application Developer.

The example mediation given in this topic is based upon the WSDL examples that are given in Supporting bound attachments: WSDL examples

About this task

You can use a mediation to map from a SOAP Messages with Attachments encoding of a message to WS-I Attachments Profile Version 1.0 encoding. The WSDL definition is the same in both cases, so if you create a mediation that rewrites the Content ID values to match the Version 1.0 conventions then the message is encoded by service integration technologies according to Version 1.0 rules.

To create a mapping mediation, use a Java Platform, Enterprise Edition (Java EE) session bean development environment to complete the following steps:

1. Create an empty mediation handler project. This creates the project, and creates the handler class that implements the handler interface. For detailed instructions on how to do this, see Writing the mediation handler.
2. Use the mediation pane on the EJB descriptor to define the handler class as a mediation handler.

   Note: When you do this, you specify a name by which the mediation handler list is known. Make a note of this name, for later reference when you create the mediation in the bus.
3. Add the mapping function to the handler. Before you begin, review Adding mediation function to handler code. Here is an example of mediation handler code that rewrites the Content ID values to match the Version 1.0 conventions:

```
int uuidBase = 0;
DataObject root = SIMessage.getDataGraph().getRootObject();
List attachments = root.getList("info/attachments");
Iterator entries = attachments.iterator();
while(entries.hasNext()) {
  DataObject entry = (DataObject) entries.next();
```

```
  if(entry.getType().equals("BoundMIMEAttachmentEntryType")) {
    String newContentId = entry.getString("messagePart") + "=" +
                          Integer.toString(uuidBase++) +
                          "@some.domain";
  }
}
```

Note: For messages that use a SOAP with attachments reference (swaref) or some other URI mechanism to refer to the attachments, the URI values might also need to be updated to match the new Content ID values. However such mechanisms are usually used to refer to unbound attachments.

For more information on the service integration technologies classes, including the mediation handler classes, see the Generated API documentation - Application programming interfaces .

4. Export the mapping mediation enterprise application.

What to do next

You are now ready to install your mediation into WebSphere Application Server and associate it with a bus destination, as described in Working with mediations.

Choosing a target service and port through a routing mediation

Write a routing mediation and configure it to select a target service and port. If you have migrated a Web services gateway that included a routing filter, use this task to recreate the equivalent functionality using a mediation.

About this task

One gateway service can map to one or more target services, and (for outbound target services) each target service can have one or more ports as defined in the outbound service WSDL. Without a routing mediation there is no point in mapping multiple targets because the gateway always picks the default destination. If you want to map multiple targets, you must write a routing mediation then configure it, for each gateway service, to select the target service and target port.

You associate routing mediations with service integration bus destinations. To pick a target service, you associate the mediation with the gateway service destination. To pick an outbound service port, you associate the mediation with the outbound service destination.

Use the steps described in Writing a routing mediation to help you write a mediation application that contains a routing handler, install it into WebSphere Application Server and associate it with a destination.

Programming for interoperation with WebSphere MQ

This topic covers programming considerations for messaging applications that interoperate with WebSphere MQ applications.

About this task

There are some differences between the WebSphere Application Server environment and the WebSphere MQ environment. If you are writing messaging programs that interoperate between these two environments, you should be aware of these differences and take them into account when designing, coding and deploying your programs.

1. Learn more about the environment differences and other relevant concepts in “Learning about programming for interoperation with WebSphere MQ” on page 916.
2. Read about design considerations for programs that interoperate with WebSphere MQ in “Designing an application for interoperation with WebSphere MQ” on page 918.

Learning about programming for interoperation with WebSphere MQ

This topic describes what you need to know to write programs that interoperate with a WebSphere MQ network.

About this task

WebSphere Application Server can be connected to other messaging systems based upon WebSphere MQ, including:

- WebSphere Application Server Version 5.1 systems that include the MQ-based JMS provider
- WebSphere Application Server Version 5.1 Java EE application clients
- WebSphere MQ queue managers or queue-sharing groups.

Interoperation with other JMS systems and clients is straightforward if your messaging application connections are built using a connection factory and stored in a JNDI namespace. The JNDI namespace insulates your application from provider-specific information, and there are no differences that are significant for programming messaging applications.

If your messaging application has to interoperate with queue managers or queue-sharing groups on WebSphere MQ systems, there are a few significant differences that you must account for in your application.

- Learn about how JNDI simplifies the programming task in "JNDI namespaces and connecting to different JMS provider environments."
- When JMS messages are sent to WebSphere MQ, a configuration setting on the destination determines whether the messages are forwarded to WebSphere MQ as MQ JMS messages (which include an MQRFH2 header) or as non-JMS MQ messages.
  1. Learn about how the two sets of formats are mapped to each other in "How service integration converts the message body to and from WebSphere MQ format" on page 921.
  2. Learn about how the different delivery options for the two message formats map to each other in "Mapping of message delivery options flowing through the WebSphere MQ link" on page 920.
- There are three main differences between service integration and WebSphere MQ messaging. These differences are described in the following topics:
  1. "How to address bus destinations across the WebSphere MQ link" on page 917
  2. "How WebSphere MQ link handles reply-to queues" on page 917
  3. "How WebSphere MQ link handles reply-to topics" on page 918
- Learn about designing an application in "Designing an application for interoperation with WebSphere MQ" on page 918.

JNDI namespaces and connecting to different JMS provider environments

This topic discusses how applications can use a JNDI namespace to connect to different JMS-provider environments to access the resources hosted by those environments.

The Java Naming and Directory Interface (JNDI) API enables JMS clients to look up configured JMS objects. By delegating all the provider-specific work to administration tasks for creating and configuring these objects, the clients can be completely portable between environments. In addition, the applications are easier to administer because they have no need for embedded administrative values in their code.

There are two types of JMS administered objects:

- ConnectionFactory - the object a client uses to create a connection with a provider.
- Destination - the object a client uses to specify the destination of messages it is sending and the source of messages it receives.

The messaging environment to which the application connects will depend upon the implementation type of the ConnectionFactory object that is obtained from JNDI. For example, if the object is a WebSphere Application Server 7.0 default messaging ConnectionFactory, then a connection will be made to the same service integration bus.

How to address bus destinations across the WebSphere MQ link

This topic describes how to handle the issues relating to addressing a service integration bus destination from WebSphere MQ, and a WebSphere MQ queue from a service integration bus.

The name characteristics and naming structure of the service integration bus and WebSphere MQ are not the same. WebSphere MQ essentially has a two-level addressing structure:

- Queue Manager name.
- Queue Name.

Each of these names is limited to 48 characters. The equivalents for the service integration bus are the messaging engine name and destination name, but the scope of a destination is not limited to a particular messaging engine. Both destination and messaging engine have a scope that spans the bus, so the service integration bus uses bus name and destination to uniquely address a particular target destination.

The service integration bus does not enforce the 48 character limit for names that is imposed by WebSphere MQ. Messages from a WebSphere MQ application sent to a bus destination with a name greater than 48 characters must have some means of using the shorter name (used in WebSphere MQ) to address the long name (used in the service integration bus). The service integration bus uses an alias destination to map between the shorter name and the long name. For more information about alias destinations, see Alias destinations.

An alias can also be used to send a message from a WebSphere Application Server application using a long name (greater than 48 characters) and route it to a WebSphere MQ queue. however, there is still an issue addressing a queue on an arbitrary queue manager in an MQ network. In the service integration bus environment, you specify the bus name (the WebSphere MQ link bus name) and the destination name, so there has to be a special format for a destination which allows you to specify the queue manager name: <queue>@<queue manager>. These destination names will only be parsed by the WebSphere MQ link, which uses the value to determine what values to place in the target queue and queue manager fields of the message header.

For example, you could define an alias on the local bus called ″MyLongQueueNameWithMoreThanFortyEightCharactersInTheName″, and set the target bus to the name of the foreign bus representing the MQ network and the target identifier to ″QUEUE1@QM2″ to address the WebSphere MQ queue called QUEUE1 on the queue manager QM2 in the WebSphere MQ network.

There are two fields in the JMS API that are used for sharing information about the destination to which a message is sent (JMSDestination) and the destination to which replies should be sent (JMSReplyTo). The JMSReplyTo field of a JMS message passing from a service integration bus to WebSphere MQ (or from WebSphere MQ to a service integration bus) is automatically mapped so that a consuming application in WebSphere MQ can reply to the original WebSphere Application Server application.

How WebSphere MQ link handles reply-to queues

This topic describes how reply-to queues, a feature of WebSphere MQ, are handled by the WebSphere MQ link.

WebSphere MQ allows the definition and usage of reply-to queues, which indicate to a receiving application where a reply should be sent. WebSphere MQ link emulates the reply-to queue functions in order to allow request/reply exchanges between applications on opposite sides of the WebSphere MQ link.

You can use reply-to queues for point-to-point request messages (queues) and for publish/subscribe request messages when exchanging information with a WebSphere MQ network.

WebSphere MQ reply-to queue names are limited to 48 characters or less. It is important when sending a message from WebSphere Application Server to WebSphere MQ that the reply queue name is less than 48 characters. This may require you to define an alias queue, as described in “How to address bus destinations across the WebSphere MQ link” on page 917.

Deciding to use reply-to queues is part of application design (see “Designing an application for interoperation with WebSphere MQ”). Your sending application must contain a definition of where replies are to be sent and attach this information to its messages. The replying application looks at this data in the received message to discover the name of the queue to which to reply.

There are two fields in the JMS API that are used for sharing information about the destination to which a message is sent (JMSDestination) and the destination to which replies should be sent (JMSReplyTo).

The JMSReplyTo field allows a response message to be returned if required. It contains enough detail for the receiving application to send a response message to the intended queue or topic so that it can be read by an application associated with the sender of the request.

How WebSphere MQ link handles reply-to topics

This text describes how reply-to topics, a feature of WebSphere MQ, are handled by WebSphere MQ link.

A WebSphere Application Server JMS application can publish a message to a topic space with a reply-to topic which is received by the publish/subscribe bridge on the WebSphere MQ link and passed to a message broker in a WebSphere MQ network. The WebSphere MQ JMS application receives the message from the message broker, obtains the reply destination and publishes a message to the message broker on the reply topic. In order for the reply message to be routed back to WebSphere Application Server subscribers, the administrator must have configured a topic mapping from WebSphere MQ to WebSphere Application Server (base) on the reply topic (unless it is a temporary reply topic, as described below.)

A WebSphere MQ JMS application can publish a message to a broker with a reply-to topic which is received by the WebSphere Application Server application. An example is a WebSphere MQ JMS application publishing to a message broker in the WebSphere MQ network, on ″myTopic″ with a reply topic of ″myReplyTopic″. A mapping must have already been specified on the publish/subscribe bridge so that the publish/subscribe bridge is subscribing to ″myTopic″ on the message broker. When the message is sent over the WebSphere MQ link it is translated into the correct format, and delivered to the publish/subscribe bridge subscriber queue where it is processed and then sent on to the topic space as specified in the publish/subscribe topic mapping. It is then received by the WebSphere Application Server JMS application, which sends a reply message back to the message broker on the WebSphere MQ network by way of the publish/subscribe bridge. You must already have created a mapping to forward messages published on ″myReplyTopic″ from WebSphere Application Server (base) to the WebSphere MQ broker.

The exceptions to this rule are temporary topic reply destinations which are automatically routed back across the WebSphere MQ link by the publish/subscribe bridge, without any intervention from the administrator. See Request-reply across the WebSphere MQ link for more information.

Note: for temporary topic reply messages to be routed from the service integration bus back to WebSphere MQ through the publish/subscribe bridge, you must configure the broker stream queue of the mapping on which the request message is sent. This field will already be specified for bi-directional topic mappings. While it is not a mandatory field for ″From MQ″ topic mappings, it must be completed if you want temporary topic reply messages to be routed.

Designing an application for interoperation with WebSphere MQ

This topic outlines the steps to consider when designing an application that will interoperate with queue managers in a WebSphere MQ network.

Before you begin

Identify the WebSphere MQ queues with which your applications will interoperate. The exact names and locations can be left to the installation.

1. Familiarize yourself with important reference information for the two interoperating environments, WebSphere MQ and the service integration bus.

   There are three types of reference material:

   - For mapping that is unique to service integration bus messaging, see “Mapping of additional MQRFH2 header fields in service integration.”
   - For mapping between WebSphere Application Server service integration bus messaging and WebSphere MQ, see “Mapping between a service integration bus and WebSphere MQ” on page 920 and How message properties and reliability levels are mapped between service integration and WebSphere MQ.
   - For the differences between the WebSphere MQ functions and the service integration bus, see “WebSphere MQ functions not supported by service integration” on page 928.

2. Design your JMS client based on the typical Java EE pattern:
   a. Use JNDI to find a ConnectionFactory object.
   b. Use JNDI to find one or more Destination objects.
   c. Use the connection factory to create a JMS Connection object.
   d. Use the JMS connection to create one or more JMS Session objects.
   e. Use a JMS session and the destinations to create the MessageProducer and MessageConsumer objects.
   f. Start delivery of messages by starting the JMS connection.

   At this point a client has the basic JMS setup needed to produce and consume messages.
3. Identify any name-handling incompatibilities between the service integration bus and WebSphere MQ environments. If necessary, identify alias requirements, so that the WebSphere MQ application can handle service integration bus destination names of greater than 48 characters. For more information, see “How to address bus destinations across the WebSphere MQ link” on page 917.
4. Identify any reply destinations that are used by your application and check them for name-handling incompatibilities. For more information, see “Reply-to queue constraints when interoperating with WebSphere MQ” on page 930.
5. If your application publishes messages that you want to be forwarded to WebSphere MQ brokers, work with your administrator to define appropriate topic mappings on a publish/subscribe broker profile. You will also need to define topic mappings for any permanent reply topics. See “How WebSphere MQ link handles reply-to topics” on page 918 and Request-reply across the WebSphere MQ link for more information.

Mapping of additional MQRFH2 header fields in service integration

This topic describes additional fields added to the MQRFH2 header to allow for functions that can be used by service integration but that are unavailable in WebSphere MQ.

A set of message fields specific to the service integration bus convey extra information not used in WebSphere MQ.

- These properties are inserted in the MQRFH2 header of application messages in the `<sib>` and `<jms>` folders.

They can be used to influence the routing of messages within the service integration bus, but do not appear as JMS message fields or properties.

MQRFH2 header and field (`<jms>` folder)	SIBusMessage field or property
Frp (appended to Dst field)	Forward routing path header field

MQRFH2 header and field (<jms> folder)	SIBusMessage field or property
Rrp (appended to Rto field)	Reverse routing path header field

MQRFH2 header and field (<sib> folder)	SIBusMessage field or property
RTopic	Reply topic
RPri	Reply priority
RPer	Reply persistence
RTTL	Reply time to live

When a message is sent to WebSphere MQ across a WebSphere MQ link, a <sib> folder is included in the MQRFH2 header of the message if both of the following are true:

- The setting on the destination definition is configured to propagate the MQRFH2 header.
- Fields corresponding to the <sib> folder content are set in the message.

Mapping between a service integration bus and WebSphere MQ

To help you program applications that interoperate with WebSphere MQ, here are a set of links to tables that describe how the message formats are mapped between service integration messages and WebSphere MQ messages.

The specifics of delivery options, message types, and fields can differ between the two environments as messages flow to a WebSphere MQ environment, and back. Attributes are mapped between the two, according to the reference information in these tables:

- Delivery options (also known as qualities of service). See “Mapping of message delivery options flowing through the WebSphere MQ link.”
- Inbound message conversion to JMS, and outbound message conversion to WebSphere MQ JMS or non-JMS messages. See “How service integration converts the message body to and from WebSphere MQ format” on page 921.
- Inbound message fields and properties conversion to JMS. See “How service integration converts the message header fields and properties to and from WebSphere MQ format” on page 923.
- Inbound MQ Message D Report fields conversion to JMS provider-specific properties. See “Mapping of MQMD Report fields to JMS provider-specific properties” on page 927.
- Data conversion. See “Conversion of data to and from WebSphere MQ” on page 928.

Mapping of message delivery options flowing through the WebSphere MQ link:

This topic shows the mapping of delivery options (qualities of service), when messages flow through the WebSphere MQ link, between WebSphere Application Server service integration and a WebSphere MQ network.

Delivery options (also called qualities of service) differ between service integration and WebSphere MQ: Service integration offers a wider range of quality of service options.

When messages are received from a WebSphere MQ network, the default mapping of delivery options, carried out by the WebSphere MQ link receiver, is shown in the table below.

For details of delivery options definitions see Message reliability levels.

WebSphere MQ delivery option	WebSphere Application Server service-integration delivery option (default)
Persistent	Assured persistent

WebSphere MQ delivery option	WebSphere Application Server service-integration delivery option (default)
Nonpersistent	Express nonpersistent

Your administrator can select the attributes of the WebSphere MQ link receiver to alter the delivery option of inbound messages:

- Inbound persistent messages can be altered, with the advanced attribute ″inbound persistent message reliability,″ to *reliable* or *assured*.
- Inbound nonpersistent messages can be altered, with the advanced attribute ″inbound nonpersistent message reliability,″ to *best effort* or *express* or *reliable*.

When messages are sent to WebSphere MQ the mapping of the delivery options, carried out by the MQLinkSender, is shown in the table below.

WebSphere Application Server service-integration delivery option	WebSphere MQ delivery option
Reliable persistent	Persistent
Assured persistent	Persistent
Reliable nonpersistent	Nonpersistent
Express nonpersistent	Nonpersistent
Best effort nonpersistent	Nonpersistent

How service integration converts the message body to and from WebSphere MQ format:

The WebSphere MQ message header (MQRFH2) and descriptor (MQMD) can contain information about the format of the WebSphere MQ message body. Service integration uses information contained in the MQRFH2 and MQMD when converting a message from WebSphere MQ format, and posts information into the MQRFH2 and MQMD when converting a message to WebSphere MQ format.

Exchanging messages between JMS programs through service integration and WebSphere MQ

In general, you do not need to be aware of conversion between message formats to exchange JMS messages between service integration and WebSphere MQ. However, you might need to learn about message conversion if your JMS applications do not behave as expected or if your service integration configuration includes JMS programs or mediations that process messages to or from non-JMS WebSphere MQ programs.

If your service integration applications exchange MapMessage objects with WebSphere MQ applications, you might need to specify a non-default map message encoding format. For more information, see Setting the WebSphere MQ encoding format for JMS Map entries.

WebSphere MQ message payload: format indications

The WebSphere MQ format message contains the following two indications of the payload format:

MQRFH2 `<mcd>` folder, `Msd` field

This field can contain information about the payload format. This is the “JMS format” information.

- When service integration converts a message to WebSphere MQ format, it automatically sets the appropriate value for the JMS message class.
- When service integration converts a message from WebSphere MQ format, it uses the value in this field (if there is an MQRFH2 that contains the field) to set the JMS message class.

JMS message class	MQRFH2 <mcd> folder, Msd field ("JMS format")
TextMessage	jms_text
BytesMessage	jms_bytes
StreamMessage	jms_stream
MapMessage	jms_map
ObjectMessage	jms_object
Message	jms_none

If the "JMS format" information is not available, for example if there is no MQRFH2, service integration sets the JMS message class based on the "MQ format" (see below).

For more information about the MQRFH2 <mcd> folder, see the WebSphere MQ Using Java documentation.

MQRFH2 (or MQMD) format field

The MQRFH2 (or the MQMD if there is no MQRFH2) format field contains information about the payload format. This is the "MQ format" information. Typically it contains MQFMT_STRING, which indicates that the payload is character data (and can be translated to a different codepage by WebSphere MQ), or MQFMT_NONE, which indicates that the payload is not character data. These values are suitable for most JMS messages, and when service integration converts a message to WebSphere MQ format it automatically sets this field to one of the following values:

JMS message class	MQRFH2 (or MQMD) format field ("MQ format")
TextMessage	MQFMT_STRING
BytesMessage	MQFMT_NONE
StreamMessage	MQFMT_STRING
MapMessage	MQFMT_STRING
ObjectMessage	MQFMT_NONE
Message	MQFMT_NONE

If your application constructs messages for a WebSphere MQ application that requires a different format value, you can override the value from the previous table by setting the JMS_IBM_Format property to the required value. A particular example is when the WebSphere MQ application requires an additional header (for example, the MQCIH header for a CICS® bridge application). Your application constructs a BytesMessage object that contains the header followed by any other message data, then replaces the default "MQ format" (MQFMT_NONE) by setting the JMS_IBM_Format property to the appropriate value for the header (for example, MQFMT_CICS for an MQCIH header).

When service integration converts a message from WebSphere MQ format, it sets the JMS_IBM_Format property to the value in the "MQ format" field. If the "JMS format" is not available, for example if there is no MQRFH2, service integration sets the JMS message class to TextMessage if the "MQ format" is MQFMT_STRING and to BytesMessage otherwise.

For more information about the MQRFH2 (or MQMD) format field, see the WebSphere MQ Application Programming Reference.

WebSphere MQ message payload: character and numeric encoding

In addition to the format field, the MQRFH2 (or the MQMD if there is no MQRFH2) contains fields that identify the character encoding and numeric encoding for the message payload.

When service integration converts a message to WebSphere MQ format, it automatically selects default values (UTF-8 character encoding and big-endian numeric encoding) which are suitable for most

JMS messages. If your application constructs messages for a WebSphere MQ application that requires a different character or numeric encoding, you can override the character encoding value by setting the **JMS_IBM_Character_Set** property to the required coded character set ID (CCSID), or the **JMS_IBM_Encoding** property to the required numeric format, or both. For information about the values you can use for **JMS_IBM_Character_Set** and **JMS_IBM_Encoding**, see the documentation in the WebSphere MQ library.

When the JMS message has a body that is encoded as character data in WebSphere MQ (TextMessage, StreamMessage, or MapMessage), setting **JMS_IBM_Character_Set** causes service integration to convert the text to that coded character set in the WebSphere MQ message body.

When the JMS message has a body that is not character data (BytesMessage or ObjectMessage), setting **JMS_IBM_Character_Set** does not cause service integration to convert the bytes; it indicates to WebSphere MQ that any character data in the message body is already encoded using the specified coded character set. If the value of the **JMS_IBM_Format** is a format that WebSphere MQ recognises, it can convert that character data to the coded character set which the receiving application requires.

When service integration converts a message from WebSphere MQ format, it sets the **JMS_IBM_Character_Set** and **JMS_IBM_Encoding** properties from the fields in the MQRFH2 (or the MQMD if there is no MQRFH2). If the JMS message is a TextMessage, StreamMessage, MapMessage, or ObjectMessage, your application makes no use of the values of the **JMS_IBM_Character_Set** and **JMS_IBM_Encoding** properties. If the JMS message is a BytesMessage, then the body of the JMS message is binary data. In this case, your application does need to be aware of the values of the **JMS_IBM_Character_Set** and **JMS_IBM_Encoding** properties, because they indicate the encoding of any character data or numeric data that is embedded within the binary data of the message.

How service integration converts the message header fields and properties to and from WebSphere MQ format:

This topic describes how service integration processes the message header fields and properties when converting messages between WebSphere MQ format and service integration format.

Exchanging messages between JMS programs through service integration and WebSphere MQ

In general, you do not need to be aware of conversion between message formats to exchange JMS messages between service integration and WebSphere MQ. However, you might need to learn about message conversion if your JMS applications do not behave as expected or if your service integration configuration includes JMS programs or mediations that process messages to or from non-JMS WebSphere MQ programs.

If your service integration applications exchange MapMessage objects with WebSphere MQ applications, you might need to specify a non-default map message encoding format. For more information, see Setting the WebSphere MQ encoding format for JMS Map entries.

WebSphere MQ message properties: the MQMD and the MQRFH2

WebSphere MQ messages contain message properties in the message descriptor (MQMD) and in the rules and formatting header 2 (MQRFH2).

When service integration converts a message to WebSphere MQ format, it sets fields in the MQMD and the MQRFH2 based on the service integration message header fields and properties; these include JMS message header fields and properties applicable to the message. When service integration converts a message from WebSphere MQ format, it sets the service integration message header fields and properties from the MQMD and the MQRFH2 in the WebSphere MQ message.

The WebSphere MQ message always includes an MQMD, but the MQRFH2 is optional because some WebSphere MQ applications cannot process messages that contain an MQRFH2. To simplify interoperation, you can configure service integration to omit the MQRFH2 from messages for applications that cannot process the MQRFH2. When service integration omits the MQRFH2, it discards the corresponding service integration header fields and properties.

Note: A small amount of the MQRFH2 information is also stored in MQMD fields. However, these MQMD fields are not exact equivalents, tend to be less specific, and cannot be relied upon to provide an adequate substitute for the MQRFH2 information. Therefore if the receiving application can accept an MQRFH2 header, you should always provide one.

Similarly, service integration might receive messages from WebSphere MQ applications that generate messages with no MQRFH2. When service integration receives a message with no MQRFH2, it creates a "best guess" service integration header, by getting as much information as it can from the MQMD, and using default values for the other fields.

For detailed information about the contents of the message descriptor and the message headers, see the WebSphere MQ Application Programming Reference. For details of WebSphere MQ JMS support, including details of how WebSphere MQ stores JMS message properties and header fields in the MQMD and the MQRFH2, see WebSphere MQ Using Java.

WebSphere MQ message properties: JMS header fields

The following table shows how service integration maps JMS header fields to and from MQMD and MQRFH2 fields when converting messages to and from WebSphere MQ format.

The table shows the MQRFH2 field as folder.field, where folder is the name of the MQRFH2 folder that contains the field, and field is the name of the field within the MQRFH2 folder.

For several JMS header fields, there is both an MQMD field and an MQRFH2 field. When service integration is converting messages to WebSphere MQ format, it sets both the MQMD and the MQRFH2 fields. When service integration is converting messages from WebSphere MQ format, it sets the JMS header field from the MQRFH2 field if it is available, otherwise from the MQMD field.

JMS header field	MQMD field	MQRFH2 field	Notes
JMSCorrelationID	CorrelId	jms.Cid	See Note 1.
JMSDeliveryMode	Persistence	jms.Dlv	
JMSDestination		jms.Dst	
JMSExpiration	Expiry	jms.Exp	
JMSMessageID	MsgId		
JMSPriority	Priority		See Note 2.
JMSRedelivered	BackoutCount		See Note 3.
JMSReplyTo	ReplyToQ and ReplyToQMgr	jms.Rto	
JMSTimestamp	PutDate and PutTime	jms.Tms	
JMSType		mcd.Type	

Note: The MQMD **CorrelId** field can hold a standard WebSphere MQ Correlation ID of 48 hexadecimal digits (24 bytes). The **JMSCorrelationID** can be a `byte[]` value, a string value containing hexadecimal characters and prefixed with "ID:", or an arbitrary string value not beginning "ID:". The first two of these represent a standard WebSphere MQ Correlation ID and map directly to or from the MQMD **CorrelId** field (truncated or padded with zeros as applicable); they do not use the

MQRFH2 **jms.Cid** field. The third (arbitrary string) uses the MQRFH2 **jms.Cid** field; the first 24 bytes of the string, in UTF-8 format, are written into the MQMD **CorrelID**.

Note: WebSphere MQ stores the **JMSPriority** value in the MQRFH2 **jms.Pri** field but does not use any value already in that field. Service integration does not check or set the MQRFH2 **jms.Pri** field.

Note: Service integration sets the **JMSRedelivered** indicator for a message it receives form WebSphere MQ based on the **BackoutCount** field of the MQMD; a non-zero **BackoutCount** value indicates that a previous receive for the message was rolled back.

WebSphere MQ message properties: JMS defined properties

The following table shows how service integration maps JMS defined properties to and from MQMD and MQRFH2 fields when converting messages to and from WebSphere MQ format.

The table shows the MQRFH2 field as folder.field, where folder is the name of the MQRFH2 folder that contains the field, and field is the name of the field within the MQRFH2 folder.

For several JMS-defined properties, there is both an MQMD field and an MQRFH2 field. When service integration is converting messages to WebSphere MQ format, it sets both the MQMD and the MQRFH2 fields. When service integration is converting messages from WebSphere MQ format, it sets the JMS defined property from the MQRFH2 field if it is available, otherwise from the MQMD field.

JMS defined property	MQMD field	MQRFH2 field	Notes
JMSXAppID	PutApplName		
JMSXDeliveryCount	BackoutCount		
JMSXGroupID	GroupId	jms.Gid	See Note 4 and Note 5.
JMSXGroupSeq	MsgSeqNumber	jms.Seq	
JMSXUserID	UserIdentifier		

Note: The MQMD **GroupId** field can hold a standard WebSphere MQ **GroupId** of 48 hexadecimal digits (24 bytes). The **JMSXGroupID** is a string value containing hexadecimal characters and prefixed with “ID:” or an arbitrary string value not beginning “ID:”. The first of these represents a standard WebSphere MQ **GroupId** and map directly to or from the MQMD **GroupId** field (truncated or padded with zeros as applicable). The third (arbitrary string) uses the MQRFH2 **jms.Gid** field; the first 24 bytes of the string, in UTF-8 format, are written into the MQMD **GroupId**.

Note: When service integration is converting messages to WebSphere MQ format, if the **JMSXGroupID** has been set then service integration also sets the **MQMF_MSG_IN_GROUP** flag in the **MsgFlags** field of the MQMD. Note that when sending group messages, the sending JMS application must ensure that the **MQMF_LAST_MSG_IN_GROUP** flag is set as required (see “WebSphere MQ message properties: JMS provider-specific properties” on page 926).

WebSphere MQ message properties: JMS provider-specific properties

The following table shows how service integration maps JMS provider-specific properties to and from MQMD and MQRFH2 fields when converting messages to and from WebSphere MQ format. Typically you use these properties to satisfy special requirements in the receiving application, so you should consult the developer or administrator of the receiving application for details of the required property values.

The table shows the MQRFH2 field as folder.field, where folder is the name of the MQRFH2 folder that contains the field, and field is the name of the field within the MQRFH2 folder.

JMS provider-specific property	MQMD field	MQRFH2 field	Notes
JMS_IBM_ArmCorrelator		mqext.Arm	See Note 6.
JMS_IBM_Character_Set	CodedCharacterSetId	CodedCharacterSetId	See Note 7.
JMS_IBM_Encoding	Encoding	Encoding	See Note 7.
JMS_IBM_Feedback	Feedback		
JMS_IBM_Format	Format	Format	See Note 7.
JMS_IBM_Last_Msg_In_Group	MQMF_LAST_MSG_IN_GROUP		See Note 8.
JMS_IBM_MQMD_CorrelId	CorrelId		See Note 9 and Note 10.
JMS_IBM_MQMD_MsgId	MsgId		See Note 9 and Note 11.
JMS_IBM_MQMD_Persistence	Persistence		See Note 9 and Note 12.
JMS_IBM_MQMD_ReplyToQ	ReplyToQ		See Note 9 and Note 13.
JMS_IBM_MQMD_ReplyToQMgr	ReplyToQMgr		See Note 9 and Note 13.
JMS_IBM_MsgType	MsgType		
JMS_IBM_PutDate	PutDate		
JMS_IBM_PutTime	PutTime		
JMS_IBM_Report_*	Report		See Note 14.
JMS_IBM_RMCorrelator		mqext.Wrm	
JMS_TOG_ARM_Correlator		mqext.Arm	See Note 6.

Note: You should use the name **JMS_TOG_ARM_Correlator** for the ARM correlator. The name **JMS_IBM_ArmCorrelator** is available for compatibility with some existing JMS programs.

Note: The **JMS_IBM_Character_Set**, **JMS_IBM_Encoding**, and **JMS_IBM_Format** properties contain information about the WebSphere MQ message payload; that is, the part of the WebSphere MQ message that follows the MQRFH2 (if there is one) or the whole WebSphere MQ message, excluding the MQMD, if there is no MQRFH2. For more information about these properties and how to use them, see “How service integration converts the message body to and from WebSphere MQ format” on page 921.

Note: **MQMF_LAST_MSG_IN_GROUP** is one of the flags in the **MsgFlags** field of the MQMD.

Note: The **JMS_IBM_MQMD_CorrelId**, **JMS_IBM_MQMD_MsgId**, **JMS_IBM_MQMD_Persistence**, **JMS_IBM_MQMD_ReplyToQ**, and **JMS_IBM_MQMD_ReplyToQMgr** properties allow JMS applications to override the service integration default processing of WebSphere MQ MQMD fields. When service integration converts messages to WebSphere MQ format, service integration sets the corresponding MQMD field for each of these properties if, and only if, that property has been explicitly set by the application (using `setObjectProperty()` or `setNonNullProperty()`).

Service integration sets each of these properties from the corresponding MQMD field when it converts a message from WebSphere MQ format.

Note: The **JMS_IBM_MQMD_CorrelId** property overrides the default processing of the **JMSCorrelationID** property. When service integration converts messages to WebSphere MQ format, service

integration sets the MQMD **CorrelId** field to the value (`byte[]`) if explicitly set of the **JMS_IBM_MQMD_CorrelId** property, regardless of the value (if any) of the **JMSCorrelationID** property. Setting the **JMS_IBM_MQMD_CorrelId** property does not affect the value of the MQRFH2 **jms.Cid** field.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_CorrelId** property to the value (`byte[]`) of the **MQMD CorrelId** field, regardless of the value (if any) of the MQRFH2 **jms.Cid** field.

Note: The **JMS_IBM_MQMD_MsgId** property overrides the JMS default processing of the **JMSMessageID** property. When service integration converts messages to WebSphere MQ format, service integration checks whether the **JMS_IBM_MQMD_MsgId** property has been explicitly set. If so, service integration sets the MQMD **MsgId** field to this value (byte[]) , and replaces the unique value of the **JMSMessageID** that JMS allocates to the message.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_MsgId** property to the value (`byte[]`) of the MQMD **MsgId** field.

Note: The **JMS_IBM_MQMD_Persistence** property overrides the default processing of the **JMSDeliveryMode** property. When service integration converts messages to WebSphere MQ format, service integration sets the MQMD **Persistence** field to the value (integer) if explicitly set of the **JMS_IBM_MQMD_Persistence** property, regardless of the value (if any) of the **JMSDeliveryMode** property. Setting the **JMS_IBM_MQMD_Persistence** property does not affect the value of the MQRFH2 **jms.Dlv** field.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_Persistence** property to the value (integer) of the MQMD **Persistence** field, regardless of the value (if any) of the MQRFH2 **jms.Dlv** field.

Note: The **JMS_IBM_MQMD_ReplyToQ** and **JMS_IBM_MQMD_ReplyToQMgr** properties override the default processing of the **JMSReplyTo** property. When service integration converts messages to WebSphere MQ format, service integration sets the MQMD **ReplyToQ** field to the value (string) if explicitly set of the **JMS_IBM_MQMD_ReplyToQ** property and the MQMD **ReplyToQMgr** field to the value (string) if explicitly set of the **JMS_IBM_MQMD_ReplyToQMgr** property, regardless of the value (if any) of the **JMSReplyTo** property. Setting the **JMS_IBM_MQMD_ReplyToQ** or **JMS_IBM_MQMD_ReplyToQMgr** field does not affect the value of the MQRFH2 **jms.Rto** field.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_ReplyToQ** and **JMS_IBM_MQMD_ReplyToQMgr** property to the values (string) of the MQMD **ReplyToQ** and **ReplyToQMgr** fields, regardless of the value (if any) of the MQRFH2 **jms.Rto** field.

Note: For a list of the **JMS_IBM_Report_*** properties, see “Mapping of MQMD Report fields to JMS provider-specific properties.”

Mapping of MQMD Report fields to JMS provider-specific properties:

This topic describes, in a table, how WebSphere MQ MQMD Report fields map to JMS provider-specific fields and properties.

Table 23. MQMD Report fields mapped to JMS provider-specific properties.

WebSphere MQ MQMD Report field	JMS provider-specific field or property name
MQRO_NONE MQRO_COA MQRO_COA_WITH_DATA MQRO_COA_WITH_FULL_DATA	JMS_IBM_Report_COA

Table 23. MQMD Report fields mapped to JMS provider-specific properties. (continued)

WebSphere MQ MQMD Report field	JMS provider-specific field or property name
MQRO_NONE MQRO_COD MQRO_COD_WITH_DATA MQRO_COD_WITH_FULL_DATA	JMS_IBM_Report_COD
MQRO_NONE MQRO_EXPIRATION MQRO_EXPIRATION_WITH_DATA MQRO_EXPIRATION_WITH_FULL_DATA	JMS_IBM_Report_Expiration
MQRO_NONE MQRO_EXCEPTION MQRO_EXCEPTION_WITH_DATA MQRO_EXCEPTION_WITH_FULL_DATA	JMS_IBM_Report_Exception
MQRO_NONE MQRO_PAN	JMS_IBM_Report_PAN
MQRO_NONE MQRO_NAN	JMS_IBM_Report_NAN
MQRO_NONE MQRO_PASS_MSG_ID	JMS_IBM_Report_Pass_Msg_ID
MQRO_NONE MQRO_PASS_CORREL_ID	JMS_IBM_Report_Pass_Correl_ID
MQRO_NONE MQRO_DISCARD_MSG	JMS_IBM_Report_Discard_Msg

Conversion of data to and from WebSphere MQ:

This topic describes how data and headers received from WebSphere MQ are converted by the WebSphere MQ link into service integration format, and how data is sent to WebSphere MQ.

Conversion of data sent from a WebSphere MQ network

The WebSphere MQ link is capable of automatically handling data in both Big and Little-Endian encoding and character sets, converting header and message content according to the WebSphere MQ formats and protocols (MQFAP) definition. Known message formats such as JMS messages and MQ string format are also automatically converted.

User defined formats are not converted and are treated solely as bytes messages. Any necessary data conversion must be performed in the receiving application.

Conversion of data sent to a WebSphere MQ network

Data and headers sent to the WebSphere MQ network are in Big Endian encoding and UTF8 format. User defined formats are not converted and are treated solely as bytes messages. Any necessary data conversion must be performed in the WebSphere MQ network.

WebSphere MQ functions not supported by service integration

This topic describes WebSphere MQ functions that are not supported by a service integration bus

- Use this topic for functions not supported by service integration.
- Use this topic for differences in delivery options and qualities of service.

WebSphere MQ functions not available

There are various functions available in a WebSphere MQ network that are not available on a service integration bus that has a WebSphere MQ link or a WebSphere MQ client link. The following list helps you identify those functions but it is given as guidance rather than a complete definition. Functions not supported include:

1. Native MQ client (this includes client applications that make use of the base MQ classes for Java) attach.
2. Message segmentation.
3. Message grouping.
4. The MQMD Offset. Original length, MsgFlags, MsgSeqNumber, and GroupId fields are not supported because Message grouping and message segmentation are not supported.
5. Distribution lists.
6. Reference messages.
7. Triggering.
8. Alternate user authority.
9. Pass/set identity context.
10. In a program, setting the attributes of a queue (that is, the equivalent function of MQSET).
11. Confirmation of arrival/delivery.
12. Cluster sender/receiver channels (and cluster workload exits), because a messaging engine cannot participate in a WebSphere MQ cluster.
13. Server and requestor channels.
14. API crossing exits.
15. Data conversion exits.
16. Channel exits.
17. The equivalent to the MCAUSER and PUTAUTH fields of a channel.
18. Networks based on NetBIOS, SPX or SNA.
19. Message based command server.
20. PCF (Programmable Canonical Form messages).
21. Model queues. Service integration does not allow you to define model queues of a given name. Service integration technology supports only one model queue called the SYSTEM.DEFAULT.MODEL.QUEUE.
22. Dynamic queue name prefix length. Service integration suffixes all dynamic queue names with "_Q" and a unique id. This restricts the name specified in the dynamic queue name field of the Object Descriptor to up to 12 chars. If this name is greater than 12 characters, then it is truncated to 12 characters. In service integration, it is not possible to create a dynamic queue with the full name specified in the dynamic queue name field of the Object Descriptor.
23. Mark skip backout option.
24. Signal option on a get request.
25. Version 3 get message options structures.
26. All queue properties (the properties of a service integration destination do not map, one for one, to the properties of a WebSphere MQ local queue, for example).
27. Poisoned messages. Service integration bus local destination definitions have a maximum failed deliveries count (that is, the equivalent to the WebSphere MQ BackoutThreshold value) but there is no equivalent of the WebSphere MQ backout requeue queue name. In service integration technology, poisoned messages are instead backed out to an exception destination. Additionally, in service integration technology, when the number of times an application backs out a poisoned message is equal to the maximum failed deliveries count, the message is automatically backed out to an ExceptionDestination. If there is more than one message in the current unit of recovery, only the

poisoned message is backed out to the ExceptionDestination. The remainder of the messages in the unit of recovery are backed out to the destination from which they were read.

28. A strict limitation of 48 bytes on the name of a queue. Service integration bus destination names can be greater than 48 bytes in length. If a destination name is to be returned to a WebSphere MQ JMS application, then it is important to use 48 byte destination lengths. Though, in some cases, it may be feasible to define an alias destination with a name length of up to 48 bytes) to map to a local destination with a name of length greater than 48 bytes.

Differences from WebSphere MQ delivery options

While WebSphere MQ supports persistent and nonpersistent messages, service integration supports five delivery options (also known as qualities of service (QoS):

- BEST_EFFORT_NONPERSISTENT
- RELIABLE_NONPERSISTENT
- EXPRESS_NONPERSISTENT
- RELIABLE_PERSISTENT
- ASSURED_PERSISTENT

Outbound BEST_EFFORT_NONPERSISTENT, RELIABLE_NONPERSISTENT, and EXPRESS_NONPERSISTENT messages sent to a WebSphere MQ network, are sent as nonpersistent messages in the WebSphere MQ network. Outbound RELIABLE_PERSISTENT and ASSURED_PERSISTENT messages, when sent to a WebSphere MQ network, are sent as persistent messages.

For inbound messages from a WebSphere MQ network, the inbound *nonpersistent* reliability (with possible values of BEST_EFFORT_NONPERSISTENT, RELIABLE_NONPERSISTENT, and EXPRESS_NONPERSISTENT) and the inbound *persistent* reliability (with possible values of RELIABLE_PERSISTENT and ASSURED_PERSISTENT), fields of the MQLinkReceiver channel can be set to specify the service integration delivery options to be used for nonpersistent and persistent messages.

Similarly, for inbound messages from JMS clients developed for WebSphere Application Server Version 5.1, the inbound nonpersistent reliability and the inbound persistent reliability fields of the MQ client link can be set to control the message persistence.

Reply-to queue constraints when interoperating with WebSphere MQ

A WebSphere MQ server provides a direct client connection between a service integration bus and queues on a WebSphere MQ queue manager or (for WebSphere MQ for z/OS) queue-sharing group. This topic describes the behavior and restrictions of the reply-to queue message property when using a WebSphere MQ server. These restrictions apply when replying to a message using a WebSphere MQ application, such as an application developed using WebSphere MQ JMS.

When a message with a reply-to destination is sent to a destination assigned to a WebSphere MQ server bus member, this is represented using the following WebSphere MQ MQMD fields:

- MQMD reply-to queue name is set to the name of the service integration destination specified as a reply-to queue.
- MQMD reply-to queue manager name is set to the name of the service integration bus from which the message was sent.

In both cases, names which are not recognized by WebSphere MQ are truncated at the first character which is not a valid WebSphere MQ character, or at the WebSphere MQ limit on the field length.

This places the following restrictions on the situations in which a WebSphere MQ JMS application can successfully reply to a message sent from service integration.

- The reply-to destination name must be a valid WebSphere MQ queue name
- The bus in which the reply-to destination resides must have a name that is a valid WebSphere MQ queue manager name.
- The reply-to destination must reside in the same bus as the bus that originates the message.
- There must be an MQ Link between the service integration bus and the WebSphere MQ network, over which the reply can flow.
- The “virtual queue manager name” given to the MQ Link must match the service integration bus name to which the MQ Link leads.

Using durable subscriptions

This topic describes things to consider when using *durable subscriptions* for publish/subscribe messaging. A durable subscription can be used to preserve messages published on a topic while the subscriber is not active.

About this task

If there is no active subscriber for a durable subscription, JMS retains the subscription’s messages until they are received by the subscriber, or until they expire, or until the durable subscription is deleted. This enables subscriber applications to operate disconnected from the JMS provider for periods of time, and then reconnect to the provider and process messages that were published during their absence.

Each JMS durable subscription is identified by a subscription name (*subName*), which is defined when the durable subscription is created. A JMS connection also has an associated client identifier (*clientID*), which is used to associate a connection and its objects with the list of messages (on the durable subscription) that is maintained by the JMS provider for the client. The subName assigned to a durable subscription must be unique within a given client ID.

If an application needs to receive messages published on a topic while the subscriber is inactive, it uses a *durable subscriber*.

In normal operation there can be at most one active (connected) subscriber for a durable subscription at a time. However, when running inside an application server it is possible to clone the application server for failover and load-balancing purposes. In this case, a cloned durable subscription can have multiple simultaneous consumers.

For information about durable subscriptions, see the JMS 1.1 Specification (for example, section 9.3.3 “Using Durable Subscriptions”).

The following operations for durable subscriptions are in addition to the usual JMS operations, such as to first look up a connection factory and a JMS destination, and to create a connection and session.

The following are the main operations for using durable subscriptions:

- Creating a new durable subscription
- Reconnecting to an existing durable subscription
- Unsubscribing (deleting) a durable subscription
- Define the Durable Subscription Home This property must be set on the JMS connection factory if durable subscriptions are to be created using connections created from this connection factory. The value is the name of the messaging engine where all durable subscriptions accessed through this connection are managed.

  You can also set the Durable Subscription Home on the JMS topic destination, which enables a single connection to access durable subscriptions on more than one messaging engine.

To be able to create durable subscriptions, the property on the connection factory must not be null (the default). Setting a value of null or empty string on the property of a destination indicates that the value specified on the connection factory should be inherited.

- Creating a new durable subscription A durable TopicSubscriber can be created by a Session or by a TopicSession.

  Having performed the normal setup, an application can create a durable subscriber to a destination. To do this, the client program creates a durable TopicSubscriber, using session.createDurableSubscriber. The name *subName* is used as an identifier of the durable subscription.

  ```
  session.createDurableSubscriber( Topic topic,
          java.lang.String subName,
          java.lang.String messageSelector,
          boolean noLocal);
  ```

  Alternatively, you can use the two-argument form of this operation, which takes only a topic and name (*subName*) as parameters. This alternative form invokes the four-argument operation with null as the messageSelector and false as the noLocal parameters.

  ```
  session.createDurableSubscriber( Topic topic, java.lang.String subName);
  ```

  A JMS durable subscription is created with a unique identifier of the form *clientID*+"##"+*subName*. The characters ## should not be used in the clientID or subName if the JMS connection is to use a durable subscription.

  Handling exceptions. The following JMS exceptions can be thrown for the reasons listed in the exception messages:

  – InvalidDestination - The name of this durable subscription (*clientID*+"##"+*subName*) clashes with an existing destination.
  – IllegalState - The method was invoked on a closed connection.
  – IllegalState - This destination is not accepting consumers. This probably means that there is already an active subscriber for this durable subscription.
  – InvalidDestination - The mediation named in the parameters cannot be found.
  – InvalidDestination - The destination cannot be found.
  – JMSSecurity - The user does not have authorization to perform this operation.
  – JMSException - Errors occurred in the MsgStore, Comms or Core layers.

- Reconnecting to an existing durable subscription To reconnect to a topic that has an existing durable subscription, the subscriber application calls `session.CreateDurableSubscriber` again, using the same parameters that it used to originally create the durable subscription. However, consider the following important restrictions:

  – The subscriber must be attached to the same connection.
  – The destination and subscription name must be the same as in the original method call.
  – If a message selector was specified, it must be the same as in the original method call.

  By calling createDurableSubscriber again, the subscriber application reconnects to the topic, and receives any messages that arrived while the subscriber was disconnected.

- Unsubscribing (deleting) a durable subscription To unsubscribe (delete) a durable subscription to a topic, the subscriber application calls `session.unsubscribe(java.lang.String name)`.

  Do not call the unsubscribe method to delete a durable subscription if there is a TopicConsumer currently consuming messages from the topic.

Sending Web service messages directly over the bus from a JAX-RPC client

About this task

Java API for XML-based remote procedure calls (JAX-RPC) client applications send and receive Web service request and response messages. JAX-RPC client applications using the IBM JAX-RPC run-time

environment can do this in a number of different ways, depending on the bindings in the WSDL document that they are developed against, and the configuration data that is used at run time.

For an introduction to basic JAX-RPC programming concepts, including the JAX-RPC client and server programming models, see Getting Started with JAX-RPC.

If you want to use a JAX-RPC client to send messages over the service integration bus, you have two choices:

- Use a SOAP binding (SOAP over HTTP or SOAP over JMS), and pass messages indirectly through an endpoint listener to an inbound service. You would do this if you had SOAP-specific JAX-RPC handlers that must run in the client application context.
- Pass messages directly into the service integration bus at a destination by "retargeting" the JAX-RPC client application as described in this topic.

  Note: There are currently limitations regarding the Java types used by services that are retargeted through a JAX-RPC client application.

Retargeting involves setting the following two values into the client application deployment descriptor, or specifying them dynamically at run time from within the client application:

- The *binding namespace* is set to indicate that the client uses the messaging bus directly.
- The *endpoint address* is set to include the particular destination and (optionally) the format of messages that the client uses.

The destination also needs to be configured so that it knows the port type of messages that the JAX-RPC client is using. There are two ways to achieve this:

- Create an outbound service. An outbound service represents an externally-provided Web service. In this case, requests from the JAX-RPC client pass through the service destination and are then sent on to the service provider defined by the outbound service configuration.
- Create an inbound service. An inbound service represents a service provided somewhere within or beyond the messaging bus. You can create an inbound service on any existing destination. The creation of an inbound service associates a WSDL port type with the destination. When retargeting to a destination with an inbound service, the client application needs to specify both the destination name and inbound service name, because it is possible to configure more than one inbound service against a single destination. In this case, requests from the JAX-RPC client pass through the destination and then onwards through the service integration bus depending on routing that is done at the initial destination.

To have Web service messages sent directly to a destination using a JAX-RPC client, complete the following steps:

1. Create the JAX-RPC client application.
2. Create the outbound service or inbound service with which you want the JAX-RPC client application to exchange messages.
3. Use the administrative console to access the port information for your JAX-RPC client application, as described in Configuring Web service client bindings and Web services client port information.
4. Override the default SOAP binding for your JAX-RPC client application. Change the binding namespace to `http://www.ibm.com/ns/2004/02/wsdl/mp/sib`
5. Override the endpoint that your JAX-RPC client application uses to send Web service requests. The new endpoint should use the sib: URL syntax and include either the outbound service destination name, or both the inbound service name and its corresponding destination name.

What to do next

After you change the *binding namespace*, any JAX-RPC handler lists that were configured for the retargeted port are ignored. For clients that are developed against WSDL with a SOAP binding, retargeting

directly to the bus causes the handlers to be ignored. However if the client is developed against the non-bound WSDL for the service, retargeting to the bus is not considered to be changing the binding namespace, and so the handler information is retained. In this case the JAX-RPC handlers are called with the SDOMessageContext subclass.

Associated reference information:

- “sib: URL syntax”

sib: URL syntax

The `sib:` URL has the following syntax:

`sib:/[destination|path]?property_1=value_1&property_2=value_2&...`

where:

- Square brackets (“[ ]”) indicate that a parameter is optional.
- Transport type is `sib:`, followed by either `/destination` to specify destination type or `/path` to specify a forward routing path, followed by a “query string” that contains one or more properties. The permitted properties are described in the subsequent sections of this topic.

Required properties

The following properties are required. They are used to specify the destination for the request.

Note: All destination names must be fully-qualified. That is, they must include the name of the service integration bus as well as the destination name itself. Use the syntax *bus*:*destination*. If a bus or destination name contains a colon or comma, wrap the name in double quotation marks (“”). If it contains a double quotation mark, repeat the quotation mark.

destinationName
The destination name.

path The forward routing path, in the form of a sequence of destination names separated by commas.

replyDestinationName
The name of the destination to be used for the reply.

inboundService
The name of the inbound service that identifies the specific attachment that the requester application uses. You can omit this value if the destination is a service destination with an associated outbound service configuration, because in that case the requester is attaching to the outbound service through the service destination.

timeout
The time the requester waits for a response. The default value is 60 seconds. A zero value indicates an unlimited wait.

Service integration technologies-related properties

The following properties are optional. If you do not specify a value for a property, then the default value is used. For more information regarding the permitted values for these properties, see the generated API information for the SIMessage interface.

reliability
The reliability of the request message.

timeToLive
The amount of time (in milliseconds) before the request times out. A zero value indicates that the request never times out.

Note: The **timeout** property (see the required properties) is the time delay after which the requester application blocks the application thread that is waiting for a response to a request and response operation. The **time to live** and **replyTimeToLive** optional properties indicate how long the request and reply messages should be processed by the messaging engines. This does not include the processing time at the service implementation. You would therefore usually set the timeout to be the sum of the request and response times to live, plus some amount for the service processing time.

priority
The priority of the request message.

user
The user ID required to access the request destination.

password
The password required to access the request destination.

replyReliability
The reliability of the reply message.

replyTimeToLive
The amount of time (in milliseconds) before the reply times out. A zero value indicates that the reply never times out.

replyPriority
The priority of the reply message.

Other properties

You can also include user-defined properties in the URL. These properties must be named with a `user.` prefix. For example:

```
sib:/destination?destinationName=myBus:myDestination & reliability=assured & user.customData=XYZ
```

After the request is sent, the URL itself is available within the message properties, named `inbound.url`.

Developing applications that use WS-Notification

Example code for common tasks that your WS-Notification applications can perform.

Before you begin

Most of these examples use the Java API for XML-based remote procedure call (JAX-RPC) APIs and WebSphere Application Server APIs and SPIs. These JAX-RPC examples can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

If you have an existing Version 6.1 WS-Notification configuration, and you want to use WS-Notification with policy sets, work through the following tasks:

1. Migrating a Version 6.1 WS-Notification configuration from WebSphere Application Server Version 6.1 to Version 7.0.
2. Preparing a migrated Version 6.1 WS-Notification configuration for reliable notification.

3. Configuring WS-Notification for reliable notification.

About this task

A single application can be coded to perform several WS-Notification tasks. Use examples to help you code these tasks into your WS-Notification applications.

For an overview of how applications can use the notification broker, see WS-Notification: How client applications interact at runtime.

WS-Notification applications divide into two broad types: those that expose a Web service endpoint (for example a WS-Notification consumer application that receives notifications of stock valuation changes), and those that do not expose a Web service endpoint (for example applications that generate notifications of stock valuation changes). For broad guidance on the steps you take to develop each of these application types, see the following topics:

- "Writing a WS-Notification application that exposes a Web service endpoint" on page 937.
- "Writing a WS-Notification application that does not expose a Web service endpoint" on page 937.

Rather than receiving all messages on a topic to which you have subscribed, your consuming application can use XML Path (XPath) selectors to filter the messages based upon the contents of each message as described in the following topic:

- "Filtering the message content of publications" on page 938.

The code examples listed in this topic use the following WebSphere Application Server APIs and SPIs:

```
com.ibm.websphere.sib.wsn.AbsoluteOrRelativeTime;
com.ibm.websphere.sib.wsn.CreatePullPoint;
com.ibm.websphere.sib.wsn.CreatePullPointResponse;
com.ibm.websphere.sib.wsn.Filter;
com.ibm.websphere.sib.wsn.GetMessages;
com.ibm.websphere.sib.wsn.GetMessagesResponse;
com.ibm.websphere.sib.wsn.NotificationMessage;
com.ibm.websphere.sib.wsn.TopicExpression;
com.ibm.websphere.webservices.soap.IBMSOAPFactory;
com.ibm.websphere.wsaddressing.EndpointReference;
com.ibm.websphere.wsaddressing.WSAConstants;
com.ibm.wsspi.wsaddressing.EndpointReferenceManager;
```

- The following examples describe a JAX-RPC client application:
  1. "Example: Subscribing a WS-Notification consumer" on page 939.
  2. "Example: Pausing a WS-Notification subscription" on page 941.
  3. "Example: Publishing a WS-Notification message" on page 942.
  4. "Example: Creating a WS-Notification pull point" on page 943.
  5. "Example: Getting messages from a WS-Notification pull point" on page 944.
  6. "Example: Registering a WS-Notification publisher" on page 944.
- The following example XML code illustrates message content filtering using XPath selectors:
  1. "Example: Matching a WS-Notification publication and subscription through a message content filter" on page 945.
- The following example WSDL document describes a Web service that implements the NotificationConsumer portType:
  1. "Example: Notification consumer Web service skeleton" on page 946.

What to do next

Your applications can also use WS-Notification to receive event notifications generated by other clients of the service integration bus such as JMS clients. This is described in Topology for WS-Notification as an

entry or exit point to the service integration bus and Providing access for WS-Notification applications to an existing bus topic space. For information about developing applications for a mixed clients solution, including cross-streaming from a JMS client, see “Sharing event notification messages with other bus client applications” on page 947.

Writing a WS-Notification application that exposes a Web service endpoint

Write a Java EE application, containing a Web service definition, that can be deployed to the application server and act as a NotificationProducer, NotificationConsumer or demand-based publisher.

Before you begin

This task assumes that you have the following resources:

- An installed and functioning copy of IBM Rational Application Developer, Rational Software Architect or equivalent tooling.
- The WSDL file for the endpoint that is to be exposed.

About this task

To write a WS-Notification application that exposes a Web service endpoint, follow the method provided by your tooling for creating a Web service implementation from a WSDL file. For example in Rational Software Architect there is a wizard in the Tutorials Gallery under “Create and deploy a WS-I compliant Web service and an enterprise bean skeleton from a WSDL document using the WebSphere Application Server runtime environment”. This wizard guides you through the following steps for writing a JAX-RPC application. The steps are very similar for writing a JAX-WS application.

1. Create a Dynamic Web Project.
2. Import and validate the WSDL file.
3. Create the Web service. Select **File** → **New** → **Other** → **Web services** → **Web service wizard** → **Skeleton EJB Web service**.
4. Implement the business methods in the generated EJB class. The methods you choose depend upon the type of endpoint that you are exposing (NotificationProducer, NotificationConsumer or demand based publisher).
5. Export the application.

What to do next

You are now ready to deploy the application to WebSphere Application Server as described in Installing application files with the console. In the Select installation options panel, ensure that the **Deploy Web services** option is enabled.

Writing a WS-Notification application that does not expose a Web service endpoint

Write a Java EE application that can be run outside of the application server to make Web service invocations against an external Web service. This application acts as a lightweight publisher, or a pull type consumer by invoking Web service operations against another Web service such as the NotificationBroker provided by WebSphere Application Server.

Before you begin

This task assumes that you have the following resources:

- An installed and functioning copy of IBM Rational Application Developer, Rational Software Architect or equivalent tooling.

- Knowledge of where to find the WSDL file for the service that is to be invoked.

About this task

To write a WS-Notification application that does not expose a Web service endpoint, follow the method provided by your tooling for creating a Web service implementation from a WSDL file. As an illustration, the following steps describe the method provided by Rational Software Architect for writing a JAX-RPC application. The steps are very similar for writing a JAX-WS application.

1. Get the WSDL files for the service that you want to invoke. If the target service is the notification broker service that was generated by WebSphere Application Server, use the administrative console to publish the WSDL files for the service to a ZIP file.
2. Create a Dynamic Web Project with a name of your choice.
3. Choose **File > New > Other > Web services > Web services Client**.
4. Select **Java Proxy**.
5. Enter or select the WSDL you obtained earlier.
6. Choose a Client Type of “Application Client” or “Java” depending upon your requirements.
7. Select your required security configuration.
8. Click **Finish**.
9. Use the generated proxy and stubs to make calls against the remote Web service. For detailed coding examples, see “Developing applications that use WS-Notification” on page 935.

What to do next

You are now ready to deploy the application for use in the Java EE application client container as described in Running application clients.

Filtering the message content of publications

Rather than receiving all messages on a topic to which you have subscribed, your consuming application can use XML Path (XPath) selectors to filter the messages based upon the contents of each message. This content-based subscription gives greater flexibility in defining the type of information that you want to receive, it allows your applications to avoid responsibility for their own filtering, and it improves performance by not flowing messages unnecessarily from the server to the application.

About this task

The core WS-Notification publish and subscribe messaging model is topic-based. Each publication is classified as belonging to one of a fixed set of topics. Publishers label each publication with a topic name and consumers subscribe to all publications on a particular topic. For example a stock trading notification system might define a topic for each issue: Publishers post information labeled with the appropriate issue as the topic name, and subscribers subscribe to information regarding some issue.

You can use XPath selectors to filter messages for a given topic, by using a Boolean expression that is evaluated over the XML message content of the message body. For example, a subscriber to a topic-based publish and subscribe system for stock trading could use XPath selectors to specify constraints against three message attributes at the same time:

- issue name
- price
- volume of shares

The resultant Boolean statement might be as follows:

```
(issue="IBM") and (price<120) and (volume>1000)
```

You code your XPath message content filters in the subscribing applications, using XML Path (XPath) language, Version 1.0..

Note: If your subscriber applications use message content filtering, and are coded to specify the XPath Version 1.0 SelectorDomain, they can also filter publications from other WS-Notification providers that are of type JMS TextMessage or BytesMessage. For more information about these JMS message types, see “Sharing event notification messages with other bus client applications” on page 947.

To filter the message content of publications using XPath selectors, complete the following steps:

1. Create a new application that subscribes a WS-Notification consumer.
2. Code an XPath message content filter in the subscribing application. For example code for doing this, see “Example: Subscribing a WS-Notification consumer.” For an example of message content filter usage, see “Example: Matching a WS-Notification publication and subscription through a message content filter” on page 945.
3. Code error handling for cases where the filter is not valid.
4. Invoke the application.

Example: Subscribing a WS-Notification consumer

This example code describes a JAX-RPC client acting in the subscriber role, subscribing a consumer application with a broker.

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs in conjunction with code generated using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

Note:

In this example, the first optional code block shows you how to create a *raw subscription*. This concept is defined in section 4.2 of the Web Services Base Notification specification.

In the normal case, a wrapped subscription causes the Notify operation of the NotificationConsumer to be driven when matching event notifications become available. If the Subscriber instead creates a raw subscription, then only the application specific content of the event notification (that is, the contents of the NotificationMessage element) are sent to the target consumer endpoint. Note that the Web service endpoint specified in the ConsumerReference of the Subscribe request that also specifies **UseRaw** (that is, a raw subscription request) does not have to implement the NotificationConsumer port type because the Notify operation will not be used to deliver event notifications.

This means that the consumer must be able to accept operations for each of the types of application content that will be published on the specified topic. This pattern allows WS-Notification to invoke a group of existing Web service applications that are not WS-Notification aware, but that wish to receive the same information.

```
// Look up the JAX-RPC service. The JNDI name is specific to your Web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create the ConsumerReference. This contains the address of the consumer Web service which is being
// subscribed, or alternatively is a reference to a pull point (see alternative below). Specifying a
// pull point EPR indicates that the consumer wishes to use pull-based notifications.
EndpointReference consumerEPR =
    EndpointReferenceManager.createEndpointReference(new URI("http://myserver.mycom.com:9080/Consumer"));

/*
// Alternative ConsumerReference for pull-based notifications:

EndpointReference consumerEPR = pullPointEPR;

*/

// Create the Filter. This provides the name of the topic to which you want to subscribe the consumer
Filter filter = new Filter();

// Create a topic expression and add it to the filter. The prefixMappings are mappings between namespace
// prefixes and their corresponding namespaces for prefixes used in the expression
Map prefixMappings = new HashMap();
prefixMappings.put("abc", "uri:example");
TopicExpression exp =
    new TopicExpression(TopicExpression.SIMPLE_TOPIC_EXPRESSION, "abc:ExampleTopic", prefixMappings);
filter.addTopicExpression(exp);


//Create an XPath message content filter
//This example selects a subset of the available messages in the topic, based upon salary level
String filterExpression = "/company/department/employee/salary > 10000";
URI xpathURI = new URI(http://www.w3.org/TR/1999/REC-xpath-19991116);

QueryExpression qexp =
    new QueryExpression(xpathURI, filterExpression);

filter.addMessageContentExpression(qexp);

// Create the InitialTerminationTime. This is the time when you want the subscription to terminate.
// For this example we set a time of 1 year in the future.
Calendar cal = Calendar.getInstance();
cal.add(Calendar.YEAR, 1);
AbsoluteOrRelativeTime initialTerminationTime = new AbsoluteOrRelativeTime(cal);

// Create the Policy information
SOAPElement[] policyElements = null;

/*
Optional
--------
The following lines show how to construct a policy indicating that the consumer
wants to receive raw style notifications:

    javax.xml.soap.SOAPFactory soapFactory = javax.xml.soap.SOAPFactory.newInstance();
    SOAPElement useRawElement = null;

    if (soapFactory instanceof IBMSOAPFactory) {
```

```
        // We can use the value add methods provided by the IBMSOAPFactory API to create the SOAPElement
        // from an XML string.
       String useRawElementXML = "<mno:UseRaw xmlns:mno=\"http://docs.oasis-open.org/wsn/b-2\"/>";
        useRawElement = ((IBMSOAPFactory) soapFactory).createElementFromXMLString(useRawElementXML);
    } else {
        useRawElement = soapFactory.createElement("UseRaw", "mno", "http://docs.oasis-open.org/wsn/b-2");
    }

    policyElements =  new SOAPElement[] { useRawElement };
*/

// Create holders to hold the multiple values returned from the broker:
// The subscription reference
EndpointReferenceTypeHolder subscriptionRefHolder = new EndpointReferenceTypeHolder();

// The current time at the broker
CalendarHolder currentTimeHolder = new CalendarHolder();

// The termination time for the subscription
CalendarHolder terminationTimeHolder = new CalendarHolder();

// Any additional elements
AnyArrayHolder anyOtherElements = new AnyArrayHolder();

// Invoke the Subscribe operation by calling the associated method on the stub
stub.subscribe(consumerEPR,
                       filter,
                       initialTerminationTime,
                       policyElements,
                       anyOtherElements,
                       subscriptionRefHolder,
                       currentTimeHolder,
                       terminationTimeHolder);

// Get the returned values:
// An endpoint reference for the subscription that has been created. It is required for
// subsequent lifetime management of the subscription, for example pausing the subscription
com.ibm.websphere.wsaddressing.EndpointReference subscriptionRef = subscriptionRefHolder.value;

// The current time at the broker
Calendar currentTime = currentTimeHolder.value;

// The termination time of the subscription
Calendar terminationTime = terminationTimeHolder.value;

// Any other information
SOAPElement[] otherElements = anyOtherElements.value;
```

Example: Pausing a WS-Notification subscription

This example code describes a JAX-RPC client acting in the subscriber role, pausing a subscription for a consumer application.

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs in conjunction with code generated using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

```
// Look up the JAX-RPC service. The JNDI name is specific to your Web services client implementation.
// The PauseSubscription operation belongs to the SubscriptionManager service
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup("java:comp/env/services/SubscriptionManager");

// Get a stub for the port on which you want to invoke operations
SubscriptionManager stub = (SubscriptionManager) service.getPort(SubscriptionManager.class);

// Associate the request with the subscription you want to pause. The subscriptionEPR is the
// EndpointReference returned by the invocation of the Subscribe operation
((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, subscriptionEPR);

// Create any optional information
SOAPElement[] optionalInformation =  new SOAPElement[] {};

// Invoke the PauseSubscription operation by calling the associated method on the stub
SOAPElement[] additionalReturnedInformation = stub.pauseSubscription(optionalInformation);
```

Example: Publishing a WS-Notification message

This example code describes a JAX-RPC client acting in the producer role, publishing a message to a broker.

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs in conjunction with code generated using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

```
// Look up the JAX-RPC service. The JNDI name is specific to your Web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create the message contents for a notification message
SOAPElement messageContents = null;
javax.xml.soap.SOAPFactory soapFactory = javax.xml.soap.SOAPFactory.newInstance();
if (soapFactory instanceof IBMSOAPFactory) {
    // You can use the value add methods provided by the IBMSOAPFactory API to create the SOAPElement
    // from an XML string.
    String messageContentsXML = "<xyz:MyData xmlns:xyz=\"uri:mynamespace\">Some data</xyz:MyData>";
    messageContents = ((IBMSOAPFactory) soapFactory).createElementFromXMLString(messageContentsXML);
} else {
    // Build up the SOAPElement using the standard javax.xml.soap APIs
    messageContents = soapFactory.createElement("MyData", "xyz", "uri:mynamespace");
    messageContents.addTextNode("Some data");
}

// Create a notification message from the contents
```

```
NotificationMessage message = new NotificationMessage(messageContents);

// Add a topic expression to the notification message indicating to which topic or topics the
// message corresponds
Map prefixMappings = new HashMap();
prefixMappings.put("abc", "uri:example");
TopicExpression exp =
    new TopicExpression(TopicExpression.SIMPLE_TOPIC_EXPRESSION, "abc:ExampleTopic", prefixMappings);
message.setTopic(exp);

// Create any optional information
SOAPElement[] optionalInformation =  new SOAPElement[] {};

/*
Optional
--------
The following line will cause the request to be associated with a particular publisher registration.
You must do this if the broker requires publishers to register. The registrationEPR is the
ConsumerReference EndpointReference returned by the broker in relation to an invocation of the
RegisterPublisher operation.

    ((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, consumerReferenceEPR);
*/

// Invoke the Notify operation by calling the associated method on the stub
stub.notify(new NotificationMessage[] { message }, optionalInformation);
```

Example: Creating a WS-Notification pull point

This example code describes a JAX-RPC client acting in the subscriber role, creating a pull point for use by a consumer application that wants to use pull style notifications.

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs in conjunction with code generated using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

```
// Look up the JAX-RPC service. The JNDI name is specific to your Web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create the request information.
SOAPElement[] optionalInformation = null;
CreatePullPoint cpp = new CreatePullPoint(optionalInformation);

// Invoke the CreatePullPoint operation by calling the associated method on the stub
CreatePullPointResponse response = stub.createPullPoint(cpp);

// Retrieve the reference to the pull point from the response
```

```
EndpointReference pullPointEPR = response.getPullPoint();

// Retrieve any additional information from the response
SOAPElement[] additionalInformation = response.getElements();
```

Example: Getting messages from a WS-Notification pull point

This example code describes a JAX-RPC client acting in the pull style consumer role, requesting messages from a pull point.

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs in conjunction with code generated using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

```
// Look up the JAX-RPC service. The JNDI name is specific to your Web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Associate the request with a pull point. The pullPointEPR is the EndpointReference returned
// from invoking the CreatePullPoint operation
((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, pullPointEPR);

// Specify the number of messages you want to retrieve
Integer numberOfMessages = new Integer(2);

// Create any optional information
SOAPElement[] optionalInformation =  new SOAPElement[] {};

// Create the request information
GetMessages request = new GetMessages(numberOfMessages, optionalInformation);

// Invoke the GetMessages operation by calling the associated method on the stub
GetMessagesResponse response = stub.getMessages(request);

// Get the messages returned from the response
NotificationMessage[] messages = response.getMessages();
```

Example: Registering a WS-Notification publisher

This example code describes a JAX-RPC client acting in the publisher registration role, registering a publisher (producer) application with a broker.

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs in conjunction with code generated using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Developing and deploying JAX-WS Web services clients

```
// Look up the JAX-RPC service. The JNDI name is specific to your Web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create a reference for the publisher (producer) being registered. This contains the address of the
// producer Web service.
EndpointReference publisherEPR =
    EndpointReferenceManager.createEndpointReference(new URI("http://myserver.mysom.com:9080/Producer"));

// Create a list (array) of topic expressions to describe the topics to which the producer publishes
// messages. For this example you simply add one topic
Map prefixMappings = new HashMap();
prefixMappings.put("abc", "uri:mytopicns");
TopicExpression topic =
    new TopicExpression(TopicExpression.SIMPLE_TOPIC_EXPRESSION, "abc:xyz", prefixMappings);
TopicExpression[] topics = new TopicExpression[] {topic};

// Indicate that you do not want the publisher to use demand based publishing
Boolean demand = Boolean.FALSE;

// Set a value for the initial termination time of the registration. For this example we use 1 year in
// the future
Calendar initialTerminationTime = Calendar.getInstance();
initialTerminationTime.add(Calendar.YEAR, 1);

// Create holders to hold the multiple values returned from the broker:
// PublisherRegistrationReference: An endpoint reference for use in lifetime management of
// the registration
EndpointReferenceTypeHolder pubRegMgrEPR = new EndpointReferenceTypeHolder();

// ConsumerReference: An endpoint reference for use in subsequent publishing of messages
EndpointReferenceTypeHolder consEPR = new EndpointReferenceTypeHolder();

// Invoke the RegisterPublisher operation by calling the associated method on the stub
stub.registerPublisher(publisherEPR, topics, demand, initialTerminationTime, null, pubRegMgrEPR, consEPR);

// Retrieve the PublisherRegistrationReference
EndpointReference registrationEPR = pubRegMgrEPR.value;

// Retrieve the ConsumerReference
EndpointReference consumerReferenceEPR = consEPR.value;
```

Example: Matching a WS-Notification publication and subscription through a message content filter

This example XML code illustrates message content filtering using XPath selectors.

In this example a business, represented by a NotificationConsumer application, wants to be notified of bank transfers of over $1,000,000. The monitoring application subscribes on behalf of the NotificationConsumer specifying a valid XPath Version 1.0 message content filter, in the following WS-Notification subscribe message:

```
<wsnt:Subscribe>
  <wsnt:ConsumerReference>
    wsa:EndpointReference
  </wsnt:ConsumerReference>
  <wsnt:Filter>
    [ <wsnt:TopicExpression Dialect="xsd:anyURI">
        {any} ?
      </wsnt:TopicExpression> |
      <wsnt:ProducerProperties Dialect="xsd:anyURI">
        {any} ?
      </wsnt:ProducerProperties> |
      <wsnt:MessageContent Dialect="xsd:anyURI">
        /bankTransfer[value %gt; 1,000,000]
      </wsnt:MessageContent> |
      {any} *
    ] *
  </wsnt:Filter> ?
  <wsnt:InitialTerminationTime>
    [xsd:dateTime | xsd:duration]
  </wsnt:InitialTerminationTime> ?
  <wsnt:SubscriptionPolicy>
    [ <wsnt:UseRaw/> |
      {any}
    ] *
  </wsnt:SubscriptionPolicy> ?
  {any}*
</wsnt:Subscribe>
```

The WS-Notification service stores the subscription and its filter.

Another WS-Notification application then publishes a notification in which the message body contains the following information:

```
<bankTransfer origin="123456 87654321" target="224466 88664422">
  <originName>IBM Corporation</originName>
  <targetName>Matt Roberts</targetName>
  <date>02/02/2006</date>
  <value currency="USD">100,000,000</value>
</bankTransfer>
```

The WS-Notification service in the application server matches this publication to the earlier subscription and delivers the notification to the consumer specified in the subscription.

Example: Notification consumer Web service skeleton

This example WSDL document describes a Web service that implements the NotificationConsumer portType defined by the Web Services Base Notification specification.

```
<?xml version="1.0" encoding="utf-8"?>

<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:wsn-bw="http://docs.oasis-open.org/wsn/bw-2"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="uri:example.wsn/consumer"
    targetNamespace="uri:example.wsn/consumer">

    <wsdl:import namespace="http://docs.oasis-open.org/wsn/bw-2"
                          location="http://docs.oasis-open.org/wsn/bw-2.wsdl" />
```

```
    <wsdl:binding name="NotificationConsumerBinding" type="wsn-bw:NotificationConsumer">
        <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="Notify">
            <wsdlsoap:operation soapAction="" />
            <wsdl:input>
                <wsdlsoap:body use="literal" />
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="NotificationConsumerService">
        <wsdl:port name="NotificationConsumerPort" binding="tns:NotificationConsumerBinding">
            <wsdlsoap:address location="http://myserver.mycom.com:9080/Consumer" />
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

The following example shows a basic implementation of the Service Endpoint Interface (SEI) generated from the preceding WSDL document using the WSDL2Java tool:

```
public class ConsumerExample implements NotificationConsumer {

    public void notify(NotificationMessage[] notificationMessage, SOAPElement[] any)
                                                                    throws RemoteException {
        // Process each NotificationMessage
        for (int i=0; i<notificationMessage.length; i++) {
            NotificationMessage message = notificationMessage[i];

            // Get the contents of the message
            SOAPElement messageContent = message.getMessageContents();

            // Get the expression indicating which topic the message is associated with
            TopicExpression topic = message.getTopic();

            // Get a reference to the producer (this value is optional and so may be null)
            EndpointReference producerRef = message.getProducerReference();

            // Get a reference to the subscription (this value is optional and so may be null)
            EndpointReference subscriptionRef = message.getSubscriptionReference();

            // User defined processing ...

        }
    }
}
```

Sharing event notification messages with other bus client applications

How to create the JMS side of a mixed WS-Notification and JMS (bus) clients configuration, to enable cross-streaming of messages between WS-Notification applications and other clients of the service integration bus.

You can configure WS-Notification so that Web service applications receive event notifications generated by other clients of the service integration bus such as JMS clients. Similarly Web service applications can generate notifications to be received by other client types. This configuration is described in the Topology for WS-Notification as an entry or exit point to the service integration bus. You achieve this configuration by creating a permanent topic namespace that allows messages to be shared between Web service and non Web service clients of the bus, as described in Providing access for WS-Notification applications to an existing bus topic space.

Interacting with JMS message types

The WS-Notification service is responsible for both inserting messages into the service integration bus (in response to Notify operations received from Web services) and receiving messages from the bus (in order to pass messages to a Web service as a result of a Subscribe operation).

Messages inserted by the WS-Notification service are of the JMS BytesMessage type, so when a Web service invokes the Notify operation against a WS-Notification service point the application content of the message is inserted into the body of a JMS BytesMessage using the UTF-8 encoding.

For messages received by the WS-Notification service in response to a subscription the reverse conversion is applied. The received message is converted to the appropriate JMS message type. If the appropriate type is determined to be a BytesMessage type, then the body of the message is converted to a string using the UTF-8 encoding and proceeds through the code for checking before being sent to the requesting Web service.

If the converted BytesMessage string does not contain an XML element when converted to a string then this message is ignored as having been originated by a non WS-Notification aware (JMS) application.

If the received message is determined to be a TextMessage then the body content of the message is extracted and processing proceeds in the same way as for the converted BytesMessage content. This means that JMS applications that want to provide event notifications to a WS-Notification application can choose to send the content as either a BytesMessage or a TextMessage depending upon which is more convenient to the application.

If the received message is neither a BytesMessage nor a TextMessage then it is discarded as having been originated by a non WS-Notification aware (JMS) application.

Note: If your subscriber applications use message content filtering, and are coded to specify the XPath Version 1.0 SelectorDomain, they can filter the message content of publications that are of type JMS TextMessage or BytesMessage.

Chapter 8. Data access resources

Task overview: Accessing data from applications

Various enterprise information systems (EIS) use different methods for storing data. These *backend* data stores might be relational databases, procedural transaction programs, or object-oriented databases.

About this task

IBM WebSphere Application Server provides several options for accessing an information system's backend data store:

- Programming directly to the database through the JDBC 4.0 API, JDBC 3.0 API, or JDBC 2.0 optional package API.
- Programming to the procedural backend transaction through various Jave EE Connector Architecture (JCA) 1.0 or 1.5 compliant connectors.
- Programming in the bean-managed persistence (BMP) bean or servlets indirectly accessing the backend store through either the JDBC API or JCA compliant connectors.
- Using container-managed persistence (CMP) beans.
- Using embedded Structured Query Language in Java (SQLJ) support with applications that use DB2 as a backend database.
- Using the IBM data access beans, which also use the JDBC API, but give you a rich set of features and function that hide much of the complexity associated with accessing relational databases.

For all of these options, except for using the JCA 1.0 or 1.5 compliant connectors, the prerequisite Web site details which databases and drivers are currently supported.

1. Develop data access applications. Develop your application to access data using the various ways available through the application server. You can access data through APIs, container-managed persistence beans, bean-managed persistence beans, session beans, or Web components.
2. Assemble data access applications using the assembly tool. Assemble your application by creating and mapping resource references.
3. Prepare for deployment: Ensure that the appropriate database objects are available. Create or configure any databases or tables required, set necessary configuration parameters to handle expected load, and configure any necessary JDBC providers and data source objects for servlets, enterprise beans, and client applications to use.
4. Install the application on your application server.

Resource adapters

A resource adapter is a system-level software driver that a Java application uses to connect to an enterprise information system (EIS). A resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application.

WebSphere Application Server supports JCA versions 1.0 and 1.5 including additional, configurable features for JCA 1.5 resource adapters with activation specifications that handle inbound requests.

Data access for container-managed persistence (CMP) beans is indirectly managed by the WebSphere Persistence Manager. The JCA specification supports persistence manager delegation of the data access to the JCA resource adapter without knowing the specific backend store. For the relational database access, the persistence manager uses the relational resource adapter to access the data from the database.

You can find the supported database platforms for the JDBC API at the WebSphere Application Server prerequisite Web site.

Java EE Connector Architecture resource adapters

An application server vendor extends its system once to support the Java Platform, Enteprise Edition Connector Architecture (JCA) and is then assured of seamless connectivity to multiple EISs. Likewise, an EIS vendor provides one standard resource adapter with the capability to plug into any application server that supports the connector architecture.

The product supports any resource adapter that implements version 1.0 or 1.5 of this specification. IBM supplies resource adapters for many enterprise systems separately from the WebSphere Application Server package, including (but not limited to): the Customer Information Control System (CICS), Host On-Demand (HOD), Information Management System (IMS™), and Systems, Applications, and Products (SAP) R/3 .

The general approach to writing an application that uses a JCA resource adapter is to develop EJB session beans or services with tools such as Rational Application Developer. The session bean uses the *javax.resource.cci* interfaces to communicate with an enterprise information system through the resource adapter.

WebSphere Relational Resource Adapter

WebSphere Application Server provides the WebSphere Relational Resource Adapter implementation. This resource adapter provides data access through JDBC calls to access the database dynamically. The connection management is based on the JCA connection management architecture and provides connection pooling, transaction, and security support. The WebSphere RRA is installed and runs as part of WebSphere Application Server, and needs no further administration.

The RRA supports both the configuration and use of JDBC data sources and Java EE Connection Architecture (JCA) connection factories. The RRA supports the configuration and use of data sources implemented as either JDBC data sources or Java EE Connector Architecture connection factories. Data sources can be used directly by applications, or they can be configured for use by container-managed persistence (CMP) entity beans.

For more information about the WebSphere Relational Resource Adapter, see the following topics:

- For information about resource adapters and data access, see “Data access portability features” on page 951
- For RRA settings, see “WebSphere relational resource adapter settings”
- For information about enterprise beans, see EJB applications

WebSphere relational resource adapter settings

Use this page to view the settings of the WebSphere relational resource adapter. This adapter is preinstalled in the product to provide access to relational databases.

Note: Although the default relational resource adapter settings are viewable, you cannot make changes to them.

To view this administrative console page, click **Resources** > **Resource adapters** > **Resource adapters**. Expand the **Preferences** section at the top of the page. Select **Show built-in resources**. The table of configured resource adapters now displays the **WebSphere Relational Resource Adapter**.

Name:

Specifies the name of the resource provider.

Data type String

Description:

Specifies a description of the relational resource adapter.

Data type String

Archive path:

Specifies the path to the Resource Adapter Archive (RAR) file containing the module for this resource adapter.

Data type String

Class path:

Specifies a list of paths or Java Archive (JAR) file names, which together form the location for the resource provider classes.

Data type String

Native path:

Specifies a list of paths that forms the location for the resource provider native libraries.

Data type String

Data access portability features

These interfaces work with the relational resource adapter (RRA) to make database-specific functions operable on connections between the application server and that database.

In other words, your applications can access data from different databases, and use functions that are specific to the database, without any code changes. Additionally, WebSphere Application Server enables you to plug in a data source that is not supported by WebSphere persistence. However, the data source *must* be implemented as either the *XADataSource* type or the *ConnectionPoolDataSource* type, and it must be in compliance with the JDBC 2.x specification.

You can achieve application portability through the following:

DataStoreHelper interface

With this interface, each data store platform can plug in its own private datastore specific functions that the relational resource adapter runtime uses. WebSphere Application Server provides an implementation for each supported JDBC provider.

The interface also provides a GenericDataStoreHelper class for unsupported data sources to use. You can subclass the GenericDataStoreHelper class or other WebSphere provided helpers to support any new data source.

Note: If you are configuring data access through a user-defined JDBC provider, do not implement the DataStoreHelper interface directly. Either subclass the GenericDataStoreHelper class or subclass one of the DataStoreHelper implementation classes provided by IBM (if your database behavior or SQL syntax is similar to one of these provided classes).

For more information, see the API documentation **DataStoreHelper** topic (as listed in the API documentation index).

The following code segment shows how a new data store helper is created to add new error mappings for an unsupported data source.

```
public class NewDSHelper extends GenericDataStoreHelper
{
  public NewDSHelper(java.util.Properties dataStoreHelperProperties)
  {
    super(dataStoreHelperProperties);
    java.util.Hashtable myErrorMap = null;
    myErrorMap = new java.util.Hashtable();
    myErrorMap.put(new Integer(-803), myDuplicateKeyException.class);
    myErrorMap.put(new Integer(-1015), myStaleConnectionException.class);
    myErrorMap.put("S1000", MyTableNotFoundException.class);
    setUserDefinedMap(myErrorMap);
    ...
  }
}
```

WSCallHelper class

This class provides two methods that enable you to use vendor-specific methods and classes that do not conform to the standard JDBC APIs (and are not part of WebSphere Application Server extension packages).

- **jdbcCall() method**

  By using the static jdbcCall() method, you can invoke vendor-specific, nonstandard JDBC methods on your JDBC objects. (For more information, see the API documentation **WSCallHelper** topic.) The following code segment illustrates using this method with a DB2 data source:

  ```
  Connection conn = ds.getConnection();
  // get connection attribute
  String connectionAttribute =(String) WSCallHelper.jdbcCall(DataSource.class, ds,
   "getConnectionAttribute", null, null);
  // setAutoClose to false
  WSCallHelper.jdbcCall(java.sql.Connection.class,
  conn, "setAutoClose",
  new Object[] { new Boolean(false)},
  new Class[] { boolean.class });
  // get data store helper
  DataStoreHelper dshelper = WSCallHelper.getDataStoreHelper(ds);
  ```

- **jdbcPass() method**

  Use this method to exploit the nonstandard JDBC classes that some database vendors provide. These classes contain methods that require vendors' proprietary JDBC objects to be passed as parameters.

  In particular, implementations of Oracle can involve use of nonstandard classes furnished by the vendor. Methods contained within these classes include:

  ```
  oracle.sql.ArrayDescriptor ArrayDescriptor.createDescriptor(java.lang.String, java.sql.Connection)
  oracle.sql.ARRAY new ARRAY(oracle.sql.ArrayDescriptor, java.sql.Connection, java.lang.Object)
  oracle.xml.sql.query.OracleXMLQuery(java.sql.Connection, java.lang.String)
  oracle.sql.BLOB.createTemporary(java.sql.Connection, boolean, int)
  oracle.sql.CLOB.createTemporary(java.sql.Connection, boolean, int)
  oracle.xdb.XMLType.createXML(java.sql.Connection, java.lang.String)
  ```

  The following code sample demonstrates how to use jdbcPass to call the Oracle method `XMLType.createXML` on a connection. This Oracle function creates an XML type object out of the XML data that the database passes to your application.

  ```
  XMLType poXML = (XMLType)(WSCallHelper.jdbcPass(XMLType.class,
  "createXML", new Object[]{conn,poString},
   new Class[]{java.sql.Connection.class, java.lang.String.class},
   new int[]{WSCallHelper.CONNECTION,WSCallHelper.IGNORE}));
  ```

  For more examples of using jdbcPass and a complete list of method parameters, see the API documentation for the WSCallHelper class. In this information center, access the API documentation with the following steps:

1. Click **Reference** > **Developer API documentation** > **Application programming interfaces**
2. Click **com.ibm.websphere.rsadapter**
3. Under the Class Summary heading, click **WSCallHelper**

The first section on jdbcPass discusses using the method to call database static methods. The second section on jdbcPass addresses database non-static methods.

Note: Use of the jdbcPass() method causes the JDBC object to be used outside of the protective mechanisms of WebSphere Application Server. Performing certain operations (such as setting autoCommit, or transaction isolation settings, etc.) outside of these protective mechanisms will cause problems with the future use of these pooled connections. IBM does not guarantee stability of the object after invocation of this method; it is the user's responsibility to ensure that invocation of this method does not perform operations that harm the object. Use at your own risk.

Because of these potential problems, WebSphere Application Server strictly controls which methods are allowed to be invoked using the jdbcPass() method support. If you require support for a method that is not listed previously in this document, contact WebSphere Application Server Support with information on the method you require.

Resource Recovery Services (RRS)

WebSphere Application Server for z/OS supports resource adapters that use Resource Recovery Services (RRS) to support global transaction processing. RRS is an z/OS extension to the JCA resource adapter specifications.

WebSphere Application Server for z/OS supports the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) 1.0., and because of this, any resource adapter that is designed to use the 1.0 level of the Java EE Connector Architecture (JCA) is supported.

In addition to the 3 types of transaction support defined by JCA, WebSphere Application Server for z/OS supports a fourth type, **RRSTransactional** support, which is a z/OS only extension to the architecture. Resource adapters that are capable of using RRS and that properly indicate to WAS z/OS they are RRSTransactional will be supported as RRS compliant resource adapters.

z/OS resource adapters that are capable of using RRS are:
- IMS Connector for Java
- CICS CTG ECI Java EE Connector
- IMS JDBC Connector
- DB2 for z/OS Local JDBC connector when used as aJDBC Provider under the WebSphere Relational Resource Adapter (RRA)

All RRS Compliant resource adapters are required to support the property **RRSTransactional** in their ManagedConnectionFactory and must support a getter method for the property.

```
java.lang.Boolean.RRSTransactional=true;

java.lang.Boolean getRRSTransactional(){
    // Determine if the adapter can run RRSTransactional based
   // on it's configuration, and set the RRSTransactional property
   // appropriately to true or false.
 return RRSTransactional;
}
```

RRS support is only applicable in a local environment, where the backend must reside on the system. CICS and IMS resources adapters may use **RRSTransactional** support only when these adapters are configured to use local interfaces to their backend resource manager, which as stated above must reside on the same system as the IBM WebSphere Application Server for z/OS. These adapters are also capable of being configured to a remote instance of their backend resource manager. In this case, the adapters will

respond ″false″ when the getRRSTransactional() method is invoked and instead of running as RRSTransactional will use whichever one of the three types of Java EE Transaction support they have chosen to support.

JDBC providers

Installed applications use JDBC providers to interact with relational databases.

The JDBC provider object supplies the specific JDBC driver implementation class for access to a specific vendor database. To create a pool of connections to that database, you associate a data source with the JDBC provider. Together, the JDBC provider and the data source objects are functionally equivalent to the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) connection factory, which provides connectivity with a non-relational database.

For a current list of supported providers, see the WebSphere Application Server prerequisite Web site. For detailed descriptions of the providers, including the supported data source classes and their required properties, refer to the topics on data source required minimum required settings, by vendor.

Note: WebSphere Application Server Version 7.0 no longer supports the DB2 for zOS Local JDBC Provider (RRS). Version 6.1 and later of the product requires that you migrate your JDBC configurations to the DB2 Using IBM JCC Driver or DB2 Universal JDBC Driver. For instructions, consult the topic on migrating from the JDBC/SQLJ Driver for OS/390® and z/OS to the DB2 Universal JDBC Driver in the Information Management Software for z/OS Solutions Information Center, which is located at http://publib.boulder.ibm.com/infocenter/dzichelp.

DB2 Universal JDBC Driver Support

This article lists the support details for using the DB2 Universal JDBC Driver with WebSphere Application Server for z/OS.

WebSphere Application Server for z/OS supports the DB2 Universal JDBC Driver. The capabilities available depend on the DB2 Universal JDBC Driver that you installed as follows:

- The z/OS Application Connectivity to DB2 for z/OS feature that provides DB2 Universal JDBC Driver Type 4 connectivity. This DB2 Universal JDBC Drive can be invoked only as a type 4 driver for z/OS. As a type 4 driver, it uses a communication protocol to communicate requests from a z/OS application to a remote DB2 database.

  When you install and configure this driver for WebSphere Application Server for z/OS, it permits your applications to use JDBC or Container Managed Persistence (CMP) support to access backend DB2 databases (DB2 V7 and up) residing on z/OS at any location. All global transactions are handled as Java Platform Enterprise Edition (Java EE) XA transactions.
- The DB2 Universal JDBC Driver in DB2 UDB for z/OS Version 8. This driver provides both Type 2 and Type 4 support.

  Type 4 driver support uses a communication protocol to communicate requests from a z/OS application to a remote DB2 database. This driver supports using Java EE XA transaction processing to process global transactions.

  Type 2 driver support uses local API protocol to communicate requests from a z/OS application to a target DB2 running on the same z/OS system image as the application. When the Type 2 driver is used under z/OS, the driver supports the use of z/OS Resource Recovery Services (RRS) to coordinate global transactions across multiple resource manages using 2-phase commit processing.

  When you install and configure this version of the driver, your applications can use JDBC or CMP support to access backend DB2 databases (DB2 V7 and up). These databases can reside on the same z/OS system image, or on a different z/OS system image. depending on the driver type used. Type 2 driver handles all global transactions as RRS-coordinated global transactions.
- The DB2 Universal JDBC Driver Provider by APAR PQ80841 on DB2 UDB for OS/390 and z/OS Version 7. This version provides both driver Type 2 and driver Type 4 support.

Type 4 driver support uses a communication protocol to communicate requests from a z/OS application to a remote DB2 database. This driver supports using Java EE XA transaction processing to process global transactions.

Type 2 driver support uses local API protocol to communicate requests from a z/OS application to a target DB2 running on the same z/OS system image as the application. When the Type 2 driver is used under z/OS, the driver supports the use of z/OS Resource Recovery Services (RRS) to coordinate global transactions across multiple resource manages using 2-phase commit processing.

When you install and configure this version of the driver, your applications can use JDBC or CMP support to access backend DB2 databases (DB2 V7 and up). These databases can reside on the same z/OS system image, or on a different z/OS system image, depending on the driver type used.

Data sources

Installed applications use a *data source* to obtain connections to a relational database. A data source is analogous to the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) connection factory, which provides connectivity to other types of enterprise information systems (EIS).

A data source is associated with a JDBC provider, which supplies the driver implementation classes that are required for JDBC connectivity with your specific vendor database. Application components transact directly with the data source to obtain connection instances to your database. The connection pool that corresponds to each data source provides connection management.

You can create multiple data sources with different settings, and associate them with the same JDBC provider. For example, you might use multiple data sources to access different databases within the same vendor database application. WebSphere Application Server requires JDBC providers to implement one or both of the following data source interfaces, which are defined by Sun Microsystems. These interfaces enable the application to run in a single-phase or two-phase transaction protocol.

- *ConnectionPoolDataSource* - a data source that supports application participation in local and global transactions, excepting two-phase commit transactions. When a connection pool data source is involved in a global transaction, transaction recovery is not provided by the transaction manager. The application is responsible for providing the backup recovery process if multiple resource managers are involved.

  Note: A connection pool data source does support two-phase commit transactions in these cases:
  – the JDBC provider is DB2 for z/OS Local JDBC provider (RRS).

  For more information, consult the article “Using one-phase and two-phase commit resources in the same transaction” on page 1489.
- *XADataSource* - a data source that supports application participation in any single-phase or two-phase transaction environment. When this data source is involved in a global transaction, the product transaction manager provides transaction recovery.

Prior to version 5.0 of the application server, the function of data access was provided by a single connection manager (CM) architecture. This connection manager architecture remains available to support Java 2 Platform, Enterprise Edition (J2EE) 1.2 applications, but another connection manager architecture is provided, based on the JCA architecture supporting the J2EE 1.3 application style, J2EE 1.4 and Java EE applications.

These architectures are represented by two types of data sources. To choose the right data source, administrators must understand the nature of their applications, EJB modules, and enterprise beans.

- Data source (WebSphere Application Server V4) - This data source runs under the original CM architecture. Applications using this data source behave as if they were running in Version 4.0.
- Data source - This data source uses the JCA standard architecture to provide support for J2EE version 1.3 and 1.4, as well as Java EE applications. It runs under the JCA connection manager and the relational resource adapter.

Choice of data source

- J2EE 1.2 application - all EJB 1.1 enterprise beans, JDBC applications, or Servlet 2.2 components must use the **4.0** data source.
- J2EE 1.3 (and subsequent releases) application -
  - EJB 1.1 module - all EJB 1.x beans must use the **4.0** data source.
  - EJB 2.0 (and subsequent releases) module - enterprise beans that include container-managed persistence (CMP) Version 1.x, 2.0, and beyond must use the **new** data source.
  - JDBC applications and Servlet 2.3+ components - must use the **new** data source.

Data access beans

Data access beans provide a rich set of features and function, while hiding much of the complexity associated with accessing relational databases.

They are Java classes written to the Enterprise JavaBeans specification.

You can use the data access beans in JavaBeans-compliant tools, such as the IBM *Rational Application Developer*. Because the data access beans are also Java classes, you can use them like ordinary classes.

The data access beans (in the package *com.ibm.db*) offer the following capabilities:

Feature
: **Details**

Caching query results
: You can retrieve SQL query results all at once and place them in a cache. Programs using the result set can move forward and backward through the cache or jump directly to any result row in the cache.

  For large result sets, the data access beans provide ways to retrieve and manage *packets*, subsets of the complete result set.

Updating through result cache
: Programs can use standard Java statements (rather than SQL statements) to change, add, or delete rows in the result cache. You can propagate changes to the cache in the underlying relational table.

Querying parameter support
: The base SQL query is defined as a Java String, with parameters replacing some of the actual values. When the query runs, the data access beans provide a way to replace the parameters with values made available at run time. Default mappings for common data types are provided, but you can specify whatever your Java program and database require.

Supporting metadata
: A *StatementMetaData* object contains the base SQL query. Information about the query (*metadata*) enables the object to pass parameters into the query as Java data types.

  Metadata in the object maps Java data types to SQL data types (as well as the reverse). When the query runs, the Java-datatyped parameters are automatically converted to SQL data types as specified in the metadata mapping.

  When results return, the metadata object automatically converts SQL data types back into the Java data types specified in the metadata mapping.

Connection management architecture

The connection management architecture for both relational and procedural access to enterprise information systems (EIS) is based on the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) specification. The Connection Manager (CM), which pools and manages connections within an application server, is capable of managing connections obtained through both resource adapters (RAs) defined by the JCA specification, and data sources defined by the Java Database Connectivity (JDBC) 2.0 (and later) Extensions specification.

To make data source connections manageable by the CM, the WebSphere Application Server provides a resource adapter (the WebSphere Relational Resource Adapter) that enables JDBC data sources to be managed by the same CM that manages JCA connections. From the CM point of view, JDBC data sources and JCA connection factories look the same. Users of data sources do not experience any programmatic or behavioral differences in their applications because of the underlying JCA architecture. JDBC users still configure and use data sources according to the JDBC programming model.

Applications migrating from previous versions of WebSphere Application Server might experience some behavioral differences because of the specification changes from various Java EE requirements levels. These differences are not related to the adoption of the JCA architecture.

If you have Java 2 Platform, Enterprise Edition (J2EE) 1.2 applications using the JDBC API that you wish to run in WebSphere Application Server 6.0 and later, the JDBC CM from Application Server version 4.0 is still provided as a configuration option. Using this configuration option enables J2EE 1.2 applications to run unaltered. If you migrate a Version 4.0 application to Version 6.0 or later, using the latest migration tools, the application automatically uses the Version 4.0 connection manager after migration. However, EJB 2.x modules in J2EE 1.3, J2EE 1.4 and Java Platform, Enterprise Edition (Java EE) applications cannot use the JDBC CM from WebSphere Application Server Version 4.0.

Connection pooling

Using connection pools helps to both alleviate connection management overhead and decrease development tasks for data access.

Each time an application attempts to access a backend store (such as a database), it requires resources to create, maintain, and release a connection to that datastore. To mitigate the strain this process can place on overall application resources, the Application Server enables administrators to establish a pool of backend connections that applications can share on an application server. Connection pooling spreads the connection overhead across several user requests, thereby conserving application resources for future requests.

The application server supports JDBC 4.0 APIs for connection pooling and connection reuse. The connection pool is used to direct JDBC calls within the application, as well as for enterprise beans using the database.

Benefits of connection pooling

Connection pooling can improve the response time of any application that requires connections, especially Web-based applications. When a user makes a request over the Web to a resource, the resource accesses a data source. Because users connect and disconnect frequently with applications on the Internet, the application requests for data access can surge to considerable volume. Consequently, the total datastore overhead quickly becomes high for Web-based applications, and performance deteriorates. When connection pooling capabilities are used, however, Web applications can realize performance improvements of up to 20 times the normal results.

With connection pooling, most user requests do not incur the overhead of creating a new connection because the data source can locate and use an existing connection from the pool of connections. When the request is satisfied and the response is returned to the user, the resource returns the connection to the connection pool for reuse. The overhead of a disconnection is avoided. Each user request incurs a fraction of the cost for connecting or disconnecting. After the initial resources are used to produce the connections in the pool, additional overhead is insignificant because the existing connections are reused.

When to use connection pooling

Use connection pooling in an application that meets any of the following criteria:
- It cannot tolerate the overhead of obtaining and releasing connections whenever a connection is used.
- It requires Java Transaction API (JTA) transactions within the Application Server.

- It needs to share connections among multiple users within the same transaction.
- It needs to take advantage of product features for managing local transactions within the application server.
- It does not manage the pooling of its own connections.
- It does not manage the specifics of creating a connection, such as the database name, user name, or password

Note: Connection pooling is not supported in an application client. The application client calls the database directly and does not go through a data source. If you want to use the getConnection() request from the application client, configure the JDBC provider in the application client deployment descriptors, using Rational Application Developer or an assembly tool. The connection is established between application client and the database. Application clients do not have a connection pool, but you can configure JDBC provider settings in the client deployment descriptors.

How connections are pooled together

When you configure a unique data source or connection factory, you must give it a unique Java Naming and Directory Interface (JNDI) name. This JNDI name, along with its configuration information, is used to create the connection pool. A separate connection pool exists for each configured data source or connection factory.

Furthermore, the application server creates a separate instance of the connection pool in each application server that uses the data source or connection factory. For example:

- Within a cluster, three z/OS controllers each contain three servants that use *myDataSource*, and for the connection pool that the Application Server creates for every instance of *myDataSource*, you can set a Maximum Connections value of 10. Therefore, you can generate up to 90 connections (9 servants times 10 connections).

Consider how this behavior potentially impacts the number of connections that your backend resource can support. See Connection pool settings for more information.

It is also important to note that when using connection sharing, it is only possible to share connections obtained from the same connection pool.

Connection and connection pool statistics:

WebSphere Application Server supports use of PMI APIs to monitor the performance of data access applications.

Performance Monitoring Infrastructure (PMI) method calls that are supported in the two existing Connection Managers (JDBC and J2C) are supported in this version of WebSphere Application Server. The calls include:

- ManagedConnectionsCreated
- ManagedConnectionsAllocated
- ManagedConnectionFreed
- ManagedConnectionDestroyed
- BeginWaitForConnection
- EndWaitForConnection
- ConnectionFaults
- Average number of ManagedConnections in the pool
- Percentage of the time that the connection pool is using the maximum number of ManagedConnections
- Average number of threads waiting for a ManagedConnection
- Average percent of the pool that is in use
- Average time spent waiting on a request
- Number of ManagedConnections that are in use
- Number of Connection Handles

- FreePoolSize
- UseTime

Java Specification Request (JSR) 77 requires statistical data to be accessed through managed beans (Mbeans) to facilitate this. The Connection Manager passes the ObjectNames of the Mbeans created for this pool. In the case of Java Message Service (JMS) *null* is passed in. The interface used is:

```
PmiFactory.createJ2CPerf(
    String pmiName, // a unique Identifier for JCA /JDBC.  This is the
                    // ConnectionFactory name.

    ObjectName providerName,// the ObjectName of the J2CResourceAdapter
                            // or JDBCProvider Mbean

    ObjectName factoryName // the ObjectName of the J2CConnectionFactory
                           // or DataSourceMbean.
)
```

The following Unified Modeling Language (UML) diagram shows how JSR 77 requires statistics to be reported:

JCAConnectionPoolStats and JDBCConnectionPoolStats objects do not have a direct implementing Mbean; the statistics are gathered through a call to PMI. A J2C resource adapter, and JDBC provider each contain a list of ConnectionFactory or DataSource ObjectNames, respectively. The ObjectNames are used by PMI to find the appropriate connection pool in the list of PMI modules.

The JCA 1.5 Specification allows an exception from the matchManagedConnection() method that indicates that the resource adapter requests that the connection not be pooled. In that case, statistics for that connection are provided separately from the statistics for the connection pool.

Connection life cycle

A ManagedConnection object is always in one of three states: *DoesNotExist*, *InFreePool*, or *InUse*.

Before a connection is created, it must be in the DoesNotExist state. After a connection is created, it can be in either the InUse or the InFreePool state, depending on whether it is allocated to an application.

Between these three states are *transitions*. These transitions are controlled by *guarding conditions*. A guarding condition is one in which *true* indicates when you can take the transition into another legal state. For example, you can make the transition from the InFreePool state to InUse state only if:

- the application has called the data source or connection factory getConnection() method (the *getConnection* condition)
- a free connection is available in the pool with matching properties (the *freeConnectionAvailable* condition)
- and one of the two following conditions are true:
  - the getConnection() request is on behalf of a resource reference that is marked unsharable
  - the getConnection() request is on behalf of a resource reference that is marked shareable but no shareable connection in use has the same properties.

This transition description follows:

InFreePool > InUse:
getConnection AND
freeConnectionAvailable AND
NOT(shareableConnectionAvailable)

Here is a list of guarding conditions and descriptions.

Condition	Description
ageTimeoutExpired	Connection is older then its ageTimeout value.
close	Application calls close method on the Connection object.
fatalErrorNotification	A connection has just experienced a fatal error.
freeConnectionAvailable	A connection with matching properties is available in the free pool.
getConnection	Application calls getConnection method on a data source or connection factory object.
markedStale	Connection is marked as stale, typically in response to a fatal error notification.
noOtherReferences	There is only one connection handle to the managed connection, and the Transaction Service is not holding a reference to the managed connection.
noTx	No transaction is in force.
poolSizeGTMin	Connection pool size is greater than the minimum pool size (minimum number of connections)
poolSizeLTMax	Pool size is less than the maximum pool size (maximum number of connections)
shareableConnectionAvailable	The getConnection() request is for a shareable connection, and one with matching properties is in use and available to share.
TxEnds	The transaction has ended.
unshareableConnectionRequest	The getConnection() request is for an unshareable connection.
unusedTimeoutExpired	Connection is in the free pool and not in use past its unused timeout value.

Getting connections

The first set of transitions covered are those in which the application requests a connection from either a data source or a connection factory. In some of these scenarios, a new connection to the database results. In others, the connection might be retrieved from the connection pool or shared with another request for a connection.

DoesNotExist

Every connection begins its life cycle in the DoesNotExist state. When an application server starts, the connection pool does not exist. Therefore, there are no connections. The first connection is not created until an application requests its first connection. Additional connections are created as needed, according to the guarding condition.

```
getConnection AND
NOT(freeConnectionAvailable) AND
poolSizeLTMax AND
(NOT(shareableConnectionAvailable) OR
unshareableConnectionRequest)
```

This transition specifies that a connection object is not created unless the following conditions occur:
- The application calls the getConnection() method on the data source or connection factory
- No connections are available in the free pool (NOT(freeConnectionAvailable))
- The pool size is less than the maximum pool size (poolSizeLTMax)
- If the request is for a sharable connection and there is no sharable connection already in use with the same sharing properties (NOT(shareableConnectionAvailable)) OR the request is for an unsharable connection (unshareableConnectionRequest)

All connections begin in the DoesNotExist state and are only created when the application requests a connection. The pool grows from 0 to the maximum number of connections as applications request new connections. The pool is **not** created with the minimum number of connections when the server starts.

If the request is for a sharable connection and a connection with the same sharing properties is already in use by the application, the connection is shared by two or more requests for a connection. In this case, a new connection is not created. For users of the JDBC API these sharing properties are most often *userid/password* and *transaction context*; for users of the Resource Adapter Common Client Interface (CCI) they are typically *ConnectionSpec*, *Subject*, and *transaction context*.

InFreePool

The transition from the InFreePool state to the InUse state is the most common transition when the application requests a connection from the pool.

```
InFreePool>InUse:
getConnection AND
freeConnectionAvailable AND
(unshareableConnectionRequest OR
NOT(shareableConnectionAvailable))
```

This transition states that a connection is placed in use from the free pool if:
- the application has issued a getConnection() call
- a connection is available for use in the connection pool (freeConnectionAvailable),
- and one of the following is true:
  - the request is for an unsharable connection (unsharableConnectionRequest)
  - no connection with the same sharing properties is already in use in the transaction. (NOT(sharableConnectionAvailable)).

Any connection request that a connection from the free pool can fulfill does not result in a new connection to the database. Therefore, if there is never more than one connection used at a time from the pool by any

number of applications, the pool never grows beyond a size of one. This number can be less than the minimum number of connections specified for the pool. One way that a pool grows to the minimum number of connections is if the application has multiple concurrent requests for connections that must result in a newly created connection.

InUse

The idea of connection sharing is seen in the transition on the InUse state.

```
InUse>InUse:
getConnection AND
ShareableConnectionAvailable
```

This transition indicates that if an application requests a shareable connection (getConnection) with the **same** sharing properties as a connection that is already in use (ShareableConnectionAvailable), the existing connection is shared.

The same user (*user name* and *password*, or *subject*, depending on authentication choice) can share connections but only within the same transaction and only when all of the sharing properties match. For JDBC connections, these properties include the *isolation level*, which is configurable on the resource-reference (IBM WebSphere extension) to data source default. For a resource adapter factory connection, these properties include those specified on the ConnectionSpec object. Because a transaction is normally associated with a single thread, you should **never** share connections across threads.

Note: It is possible to see the same connection on multiple threads at the same time, but this situation is an error state usually caused by an application programming error.

Returning connections

All of the transitions discussed previously involve getting a connection for application use. With that goal, the transitions result in a connection closing, and either returning to the free pool or being destroyed. Applications should explicitly close connections (note: the connection that the user gets back is really a connection handle) by calling close() on the connection object. In most cases, this action results in the following transition:

```
InUse>InFreePool:
(close AND
noOtherReferences AND
NoTx AND
UnshareableConnection)
OR
(ShareableConnection AND
TxEnds)
```

Conditions that cause the transition from the InUse state are:

- If the application or the container calls close() (producing the close condition) and there are no references (the noOtherReferences condition) either by the application (in the application sharing condition) or by the transaction manager (in the NoTx condition, meaning that the transaction manager holds a reference when the connection is enlisted in a transaction), the connection object returns to the free pool.
- If the connection was enlisted in a transaction but the transaction manager ends the transaction (the txEnds condition), and the connection was a shareable connection (the ShareableConnection condition), the connection closes and returns to the pool.

When the application calls close() on a connection, it is returning the connection to the pool of free connections; it is **not** closing the connection to the data store. When the application calls close() on a currently shared connection, the connection is *not returned* to the free pool. Only after the application drops the last reference to the connection, and the transaction is over, is the connection returned to the

pool. Applications using unsharable connections must take care to close connections in a timely manner. Failure to do so can starve out the connection pool, making it impossible for any application running on the server to get a connection.

When the application calls close() on a connection enlisted in a transaction, the connection is not returned to the free pool. Because the transaction manager must also hold a reference to the connection object, the connection cannot return to the free pool until the transaction ends. Once a connection is enlisted in a transaction, you cannot use it in any other transaction by any other application until after the transaction is complete.

There is a case where an application calling close() can result in the connection to the data store closing and bypassing the connection return to the pool. This situation happens if one of the connections in the pool is considered stale. A connection is considered stale if you can no longer use it to contact the data store. For example, a connection is marked stale if the data store server is shut down. When a connection is marked as stale, the entire pool is cleaned out by default because it is very likely that all of the connections are stale for the same reason (or you can set your configuration to clean just the failing connection). This cleansing includes marking all of the currently InUse connections as stale so they are destroyed upon closing. The following transition states the behavior on a call to close() when the connection is marked as stale:

```
InUse>DoesNotExist:
close AND
markedStale AND
NoTx AND
noOtherReferences
```

This transition states that if the application calls close() on the connection and the connection is marked as stale during the pool cleansing step (markedStale), the connection object closes to the data store and is not returned to the pool.

Finally, you can close connections to the data store and remove them from the pool.

This transition states that there are three cases in which a connection is removed from the free pool and destroyed.

1. If a fatal error notification is received from the resource adapter (or data source). A fatal error notification (FatalErrorNotification) is received from the resource adaptor when something happens to the connection to make it unusable. All connections currently in the free pool are destroyed.
2. If the connection is in the free pool for longer than the unused timeout period (UnusedTimeoutExpired) and the pool size is greater than the minimum number of connections (poolSizeGTMin), the connection is removed from the free pool and destroyed. This mechanism enables the pool to shrink back to its minimum size when the demand for connections decreases.
3. If an age timeout is configured and a given connection is older than the timeout. This mechanism provides a way to recycle connections based on age.

Unshareable and shareable connections

The application server supports both unshareable and shareable connections. An unshareable connection is not shared with other components in the application. The component using this connection has full control of this connection.

Access to a resource marked as unshareable means that there is a one-to-one relationship between the connection handle a component is using and the physical connection with which the handle is associated. This access implies that every call to the getConnection method returns a connection handle solely for the requesting user. Typically, you must choose unshareable if you might do things to the connection that could result in unexpected behavior occurring in another application that is sharing the connection (for example, unexpectedly changing the isolation level).

Marking a resource as shareable allows for greater scalability. Instead of creating new physical connections on every getConnection() invocation, the physical connection (that is, managed connection) is

shared through multiple connection handles, as long as each getConnection request has the same connection properties. However, sharing a connection means that each user must not do anything to the connection that could change its behavior and disrupt a sharing partner (for example, changing the isolation level). The user also cannot code an application that assumes sharing to take place because it is up to the run time to decide whether or not to share a particular connection.

Connection property requirements

To permit sharing of connections used within the same transaction, the following data source properties must be the same:
- Java Naming and Directory Interface (JNDI) name. While not actually a connection property, this requirement simply means that you can only share connections from the same data source in the same server.
- Resource authentication
- In relational databases:
  - Isolation level (corresponds to access intent policies applied to CMP beans)
  - Readonly
  - Catalog
  - TypeMap

For more information on sharing a connection with a CMP bean, see “Sharing a connection with a CMP bean.”

To permit sharing of connections within the same transaction, the following properties must be the same for the connection factories:
- JNDI name. While not actually a connection property, this requirement simply means that you can only share connections from the same connection factory in the same server.
- Resource authentication

In addition, the ConnectionSpec object that is used to get the connection must also be the same.

Note: Java Message Service (JMS) connections cannot be shared with non-JMS connections.

Sharing a connection with a CMP bean

The application server allows you to share a physical connection among a CMP bean, a BMP bean, and a JDBC application to reduce the resource allocation or deadlock scenarios. There are several ways to ensure that all of these entity beans and the JDBC applications are sharing the same physical connection.

- **Sharing a connection between CMP beans or methods**

  When all CMP bean methods use the same access intent, they all share the same physical connection. A different access intent policy triggers the allocation of a different physical connection. For example, a CMP bean has two methods; method 1 is associated with `wsPessimisticUpdate` intent, whereas method 2 has `wsOptimisticUpdate` access intent. Method 1 and method 2 cannot share the same physical connection within a transaction. In other words, an XA data source is required to run in a global transaction.

  You can experience some deadlocks from a database if both methods try to access the same table. Therefore, sharing a connection is determined by the access intents that are defined in the CMP methods.

- **Sharing a connection between CMP and BMP beans**

  Remember to first verify that the **getConnection** methods of both the BMP bean and the CMP bean set the same connection properties. To match the authentication type of the CMP bean resource, set the authentication type of the BMP bean resource to container-managed, which is designated in the deployment descriptor as `res-auth = Container`.

  Additionally, use one of the following options to ensure connection-sharing between the bean types:

  - Define the same access intent on both CMP and BMP bean methods. Because both use the same access intent, they share the same physical connection. The advantage to using this option is that the backend is transparent to a BMP bean. However, this option also makes the BMP non-portable because it uses the WebSphere extended API to handle the isolation level. For more information, refer to the code example in Example: Accessing data using IBM extended APIs to share connections between container-managed and bean-managed persistence beans.
  - Determine the isolation level that the access intent uses on a CMP bean method, then use the corresponding isolation level that is specified on the resource reference to look up a data source and a connection. This option is more of a manual process, and the isolation level might be different from database to database. For more information refer to the isolation level and access intent mapping table: Access intent isolation levels and update locks and the Isolation level and resource reference section.
- **Sharing a connection between CMP and a JDBC application that is used by a servlet or a session bean**

  Determine the isolation level that the access intent uses on a CMP bean method, then use the corresponding isolation level specified on the resource reference to look up a data source and a connection. For more information refer to Access intent isolation levels and update locks and Isolation level and resource reference.

Factors that determine sharing

The listing here is not an exhaustive one. The product might or might not share connections under different circumstances.

- Only connections acquired with the same resource reference (resource-ref) that specifies the res-sharing-scope as shareable are candidates for sharing. The resource reference properties of res-sharing-scope and res-auth and the IBM extension isolationLevel help determine if it is possible to share a connection. IBM extension isolationLevel is stored in IBM deployment descriptor extension file; for example: ibm-ejb-jar-ext.xmi.
- You can only share connections that are requested with the same properties.
- Connection sharing only occurs between different component instances if they are within a transaction (container- or user-initiated transaction).
- Connection sharing only occurs within a sharing boundary. Current® sharing boundaries include *Transactions* and *LocalTransactionContainment* (LTC) boundaries.
- Connection sharing rules within an LTC Scope:
  - For shareable connections, only *Connection Reuse* is allowed within a single component instance. Connection reuse occurs when the following actions are taken with a connection: get, use, commit/rollback, close; get, use, commit/rollback, close. Note that if you use the LTC resolution-control of *ContainerAtBoundary* then no start/commit is needed because that action is handled by the container.

    The connection returned on the second *get* is the same connection as that returned on the first *get* (if the same properties are used). Because the connection use is serial, only one connection handle to the underlying physical connection is used at a time, so true connection sharing does not take place. The term *"reuse"* is more accurate.

    More importantly, the *LocalTransactionContainment* boundary enclosing both *get* actions is not complete; no cleanUp() method is invoked on the ManagedConnection object. Therefore the second *get* action inherits all of the connection properties set during the first getConnection() call.
- Shareable connections between transactions (either container-managed transactions (CMT), bean-managed transactions (BMT), or LTC transactions) follow these caching rules:
  - In general, setting properties on shareable connections is not allowed because a user of one connection handle might not anticipate a change made by another connection handle. This limitation is part of the Java Platform, Enterprise Edition (Java EE) standard.
  - General users of resource adapters can set the connection properties on the connection factory getConnection() call by passing them in a ConnectionSpec.

However, the properties set on the connection during one transaction are not guaranteed to be the same when used in the next transaction. Because it is not valid to share connections outside of a sharing scope, connection handles are moved off of the physical connection with which they are currently associated when a transaction ends. That physical connection is returned to the free connection pool. Connections are cleaned before going in the free pool. The next time the handle is used, it is automatically associated with an appropriate connection. The appropriateness is based on the security login information, connection properties, and (for the JDBC API) the isolation level specified in the extended resource reference, passed in on the original request that returned the current handle. Any properties set on the connection after it was retrieved are lost.

– For JDBC users, the application server provides an extension to enable passing the connection properties through the ConnectionSpec.

  Use caution when setting properties and sharing connections in a local transaction scope. Ensure that other components with which the connection is shared are expecting the behavior resulting from your settings.

- You cannot set the isolation level on a shareable connection for the JDBC API using a relational resource adapter in a global transaction. The product provides an extension to the resource reference to enable you to specify the isolation level. If your application requires the use of multiple isolation levels, create multiple resource references and map them to the same data source or connection factory.

Maximal connection sharing

To maximize connection sharing opportunities for an application, ensure that each component has the local transaction containment (LTC) Resolver attribute set to **ContainerAtBoundary**. This setting specifies that the component container, rather than the application code, resolves all resource manager local transactions (RMLTs) within the LTC scope. The container begins an RMLT when a connection is first used within the LTC scope, and completes it automatically at the end of the LTC scope.

Consult the topic “Configuring transactional deployment attributes” on page 1482 for instructions on setting the transaction resolution control and other attributes.

Connection sharing violations

There is a new exception, the sharing violation exception, that the resource adapter can issue whenever an operation violates sharing requirements. Possible violations include changing connection attributes, security settings, or isolation levels, among others. When such a mutable operation is performed against a managed connection, the sharing violation exception can occur when both of the following conditions are true:

- The number of connection handles associated with the managed connection is more than one.
- The managed connection is associated with a transaction, either local or XA.

Both the component and the J2C run time might need to detect this sharing violation exception, depending on when and how the managed connection becomes unshareable. If the managed connection becomes unshareable because of an operation through the connection handle (for example, you change the isolation level), then the component needs to process the exception. If the managed connection becomes unshareable without being recognized by the application server (due to some component interaction with the connection handle), then the resource adapter can reject the creation of a connection handle by issuing the sharing violation exception.

Connection handles

A connection handle is a representation of a physical connection. To use a backend resource (such as a relational database) in WebSphere Application Server you must get a connection to that resource. When you call the *getConnection()* method, you get a *connection handle* returned. The handle is not the physical connection. The physical connection is managed by the connection manager.

There are two significant configurations that affect how connection handles are used and how they behave. The first is the *res-sharing-scope*, which is defined by the resource-reference used to look up the DataSource or Connection Factory. This property tells the connection manager whether or not you can share this connection.

The second factor that affects connection handle behavior is the *usage pattern*. There are essentially two usage patterns. The first is called the *get/use/close* pattern. It is used within a single method and without calling another method that might get a connection from the same data source or connection factory. An application using this pattern does the following:
1. gets a connection
2. does its work
3. commits (if appropriate)
4. closes the connection.

The second usage pattern is called the *cached handle* pattern. This is where an application:
1. gets a connection
2. begins a global transaction
3. does work on the connection
4. commits a global transaction
5. does work on the connection again

A cached handle is a connection handle that is held across transaction and method boundaries by an application. Keep in mind the following considerations for using cached handles:

- Cached handle support requires some additional connection handle management across these boundaries, which can impact performance. For example, in a JDBC application, *Statements*, *PreparedStatements*, and *ResultSets* are closed implicitly after a transaction ends, but the connection remains valid.
- You are encouraged **not** to cache the connection across the transaction boundary for shareable connections; the get/use/close pattern is preferred.
- Caching of connection handles across servlet methods is limited to JDBC and Java Message Service (JMS) resources. Other non-relational resources, such as Customer Information Control System (CICS) or IMS objects, currently cannot have their connection handles cached in a servlet; you need to get, use, and close the connection handle within each method invocation. (This limitation only applies to single-threaded servlets because multithreaded servlets do not allow caching of connection handles.)
- You **cannot** pass a cached connection handle from one instance of a data access client to another client instance. Transferring between client instances creates the problematic contingency of one instance using a connection handle that is referenced by another. This relationship can only cause problems because connection handle management code processes tasks for each client instance *separately*. Hence, connection handle transfers result in run-time scenarios that trigger exceptions. For example:
  1. The application code of a client instance that receives a transferred handle closes the handle.
  2. If the client instance that retains the original reference to the handle tries to reclaim it, the application server issues an exception.

The following code segment shows the cached connection pattern.

```
Connection conn = ds.getConnection();
ut.begin();
conn.prepareStatement("....."); --> Connection runs in global transaction mode
...
ut.commit();
conn.prepareStatement(".....");   ---> Connection still valid but runs in autoCommit(True);
...
```

Unshareable connections

Some characteristics of connection handles retrieved with a *res-sharing-scope* of **unshareable** are described in the following sections.

- **The possible benefits of unshared connections**
  - Your application always maintains a direct link with a physical connection (managed connection).
  - The connection always has a one-to-one relationship between the connection handle and the managed connection.
  - In most cases, the connection does not close until the application closes it.
  - You can use a cached unshared connection handle across multiple transactions.
  - The connection can have a performance advantage in some cached handle situations. Because unshared connections do not have the overhead of moving connection handles off managed connections at the end of the transaction, there is less overhead in using a cached unshared connection.
- **The possible drawbacks of unshared connections**
  - Inefficient use of your connection resources. For example, if within a single transaction you get more than one connection (with the same properties) using the same data source or connection factory (same resource-ref) then you use multiple physical connections when you use unshareable connections.
  - Wasted connections. It is important not to keep the connection handle open (that is, your application does not call the *close()* method) any longer then it is needed. As long as an unshareable connection is open, the physical connection is unavailable to any other component, even if your application is not currently using that connection. Unlike a shareable connection, an ushareable connection is not closed at the end of a transaction or servlet call.
  - Deadlock considerations. Depending on how your components interact with the database within a transaction, using unshared connections can lead to deadlock in the database. For example, within a transaction, component A gets a connection to data source X and updates table 1, and then calls component B. Component B gets another connection to data source X, and updates/reads table 1 (or even worse the same row as component A). In some circumstances, depending on the particular database, its locking scheme, and the transaction isolation level, a deadlock can occur.

    In the same scenario, but with a *shared* connection, deadlock does not occur because all the work is done on the same connection. It is worth noting that when writing code that uses shared connections, you use a strategy that calls for multiple work items to be performed on the same connection, possibly within the same transaction. If you decide to use an unshareable connection, you must set the *maximum connections* property on the connection factory or data source correctly. An exception might occur for waiting connection requests if you exceed the maximum connections value, and unshareable connections are not being closed before the connection wait time-out is exceeded.

Shareable connections

Some characteristics of connection handles that are retrieved with a *res-sharing-scope* of **shareable** are described in the following sections.

- **The possible benefits of shared connections**
  - Within an instance of connection sharing, application components can share a managed connection with one or more connection handles, depending on how the handle is retrieved and which connection properties are used.
  - They can more efficiently use resources. Shareable connections are not valid outside of their sharing boundary. For this reason, at the end of a sharing boundary (such as a transaction) the connection handle is no longer associated with the managed connection it was using within the sharing boundary (this applies only when using the cached handle pattern). The managed connection is returned to the free connection pool for reuse. Connection resources are not held longer than the end of the current sharing scope.

    If the cached handle pattern is used, then the next time the handle is used within a new sharing scope, the application server run time ensures that the handle is reassociated with a managed

connection that is appropriate for the current sharing scope, and has the same properties with which the handle was originally retrieved. Remember that it is not appropriate to change properties on a shareable connection. If properties are changed, other components that share the same connection might experience unexpected behavior. Futhermore, when using cached handles, the value of the changed property might not be remembered across sharing scopes.

- **The possible drawbacks of shared connections**
  - Sharing within a single component (such as an enterprise bean and its related Java objects) is not always supported. The current specification allows resource adapters the choice of only allowing one active connection handle at a time.

    If a resource adapter chooses to implement this option then the following scenario results in an *invalid handle exception*: A component using shareable connections gets a connection and uses it. Without closing the connection, the component calls a utility class (Java object) that gets a connection handle to the same managed connection and uses it. Because the resource adapter only supports one active handle, the first connection handle is no longer valid. If the utility object returns without closing its handle, the first handle is not valid and triggers an exception at any attempt to use it.

    Note: This exception occurs only when calling a utility object (a Java object).

    Not all resource adapters have this limitation; it occurs only in certain implementations. The WebSphere Relational Resource Adapter (RRA) does not have this limitation. Any data source used through the RRA does not have this limitation. If you encounter a resource adapter with this limitation you can work around it by serializing your access to the managed connection. If you always close your connection handle before getting another (or close your handle before calling code that gets another handle), and before returning from a method, you can allow two pieces of code to share the same managed connection. You simply cannot use the connection for both events at the same time.
  - Trying to change the *isolation level* on a shareable JDBC-based connection in a global transaction (that is supported by the RRA) causes an exception. The correct way to get connections with different transaction isolation levels is by configuring the IBM extended resource-reference.
  - Closing connection handles for shareable connections by an application is NOT supported and causes errors. However, you can avoid this limitation by using the Relational Resource Adapter.

Lazy connection association optimization

The Java Platform, Enterprise Edition (Java EE) Connector (J2C) connection manager implemented *smart handle* support. This technology enables allocation of a connection handle to an application while the managed connection associated with that connection handle is used by other applications (assuming that the connection is not being used by the original application). This concept is part of the Java EE Connector Architecture (JCA) 1.5 specification. (You can find it in the JCA 1.5 specification document in the section entitled ″Lazy Connection Association Optimization.″) Smart handle support introduces use a method on the ConnectionManager object, the *LazyAssociatableConnectionManager()* method, and a new marker interface, the *DissociatableManagedConnection* class. You must configure the provider of the resource adapter to make this functionality available in your environment. (In the case of the RRA, WebSphere Application Server itself is the provider.) The following code snippet shows how to include smart handle support:

```
package javax.resource.spi;
import javax.resource.ResourceException;

interface LazyAssociatableConnectionManager { // application server
    void associateConnection(
        Object connection, ManagedConnectionFactory mcf,
        ConnectionRequestInfo info) throws ResourceException;
}

interface DissociatableManagedConnection { // resource adapter
    void dissociateConnections() throws ResourceException;
}
```

This DissociatableManagedConnection interface introduces another state to the Connection object: *inactive*. A Connection can now be active, closed, and inactive. The connection object enters the inactive state when a corresponding ManagedConnection object is cleaned up. The connection stays inactive until an application component attempts to re-use it. Then the resource adapter calls back to the connection manager to re-associate the connection with an active ManagedConnection object.

Transaction type and connection behavior

All connection usage occurs within the scope of either a global transaction or a local transaction containment (LTC) boundary. Each transaction type places different requirements on connections and impacts connection settings differently.

Connection sharing and reuse

You can share connections within a global transaction scope (assuming other sharing rules are met). You can also share connections within a shareable LTC. You can *serially reuse* connections within an LTC scope. A get/use/close connection pattern followed by another instance of get/use/close (to the same data source or connection factory) enables you to reuse the same connection. See the "Unshareable and shareable connections" on page 963 topic for more details.

JDBC AutoCommit behavior

All JDBC connections, when first obtained through a getConnection() call, contain the setting AutoCommit = TRUE by default. However, different transaction scope and settings can result in changing, or simply overriding, the AutoCommit value.

- If you operate within an LTC and have its resolution-control set to *Application*, AutoCommit remains *TRUE* unless changed by the application.
- If you operate within an LTC and have its resolution-control set to *ContainerAtBoundary*, the application must **not** touch the AutoCommit setting. The WebSphere Application Server run time sets the AutoCommit value to *FALSE* before work begins, then commits or rolls back the work, as appropriate, at the end of the LTC scope.
- If you use a connection within a global transaction, the database ignores the AutoCommit setting so that the transaction service that controls the commit and rollback processing can manage the transaction. This action takes place upon first use of the connection to do work, regardless of the user changing the AutoCommit setting. After the transaction completes, the AutoCommit value returns to the value it had before the first use of the connection. So even if the AutoCommit value is set to *TRUE* before the connection is used in a global transaction, you need not set the value to *FALSE* because the value is ignored by the database. In this example, after the transaction completes, the AutoCommit value of the connection returns to *TRUE*.
- If you use multiple distinct connections within a global transaction, all work is guaranteed to commit or roll back together. This is not the case for a local transaction containment (LTC scope). Within an LTC, work done on one connection commits or rolls back independently from work done on any other connection within the LTC.

One-phase commit and two-phase commit connections

The type and number of resource managers, such as a database server, that must be accessed by an application often determines the application transaction requirements. Consequently each type of resource manager places different requirements on connection behavior.

- A two-phase commit resource manager can support two-phase coordination of a transaction. That support is necessary for transactions that involve other resource managers; these transactions are global transactions. See "Transaction support in WebSphere Application Server" on page 1455 for further explanation.
- A one-phase commit resource manager supports only one-phase transactions, or LTC transactions, in which that resource is the sole participating datastore. Again, see "Transaction support in WebSphere Application Server" on page 1455 for further explanation.

One-phase commit resources are such that work being done on a one phase connection cannot mix with other connections and ensure that the work done on all of the connections completes or fails atomically. The product does not allow more than one one-phase commit connection in a global transaction. Futhermore, it does not allow a one-phase commit connection in a global transaction with one or more two-phase commit connections. You can coordinate only multiple two-phase commit connections within a global transaction.

WebSphere Application Server provides *last participant support*, which enables a single one-phase commit resource to participate in a global transaction with one or more two-phase commit resources.

Note that any time that you do multiple getConnection() calls using a resource reference that specifies res-sharing-scope=Unshareable, you get multiple physical connections. This situation also occurs when res-sharing-scope=Shareable, but the sharing rules are broken. In either case, if you run in a global transaction, ensure the resources involved are enabled for two-phase commit (also sometimes referred to as *JTA Enabled*). Failure to do so results in an XA exception that logs the following message:

```
WTRN0063E: An illegal attempt to enlist a one phase capable resource with existing two phase capable
resources has occurred.
```

Application scoped resources

Use this page to view brief descriptions of the resources that are bundled with your application. You can view individual resource settings by clicking on the resource name.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Application scoped resources**.

Each table row corresponds to a resource that is bundled with your application. Click a resource name or the corresponding provider name to view an administrative console page where you can edit the object configuration settings.

Name:

Specifies the administrative name that was assigned to this resource.

Click this name to view a page where you can edit the configuration settings.

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name of the resource.

Data type String

Resource type:

Specifies the type of resource, such as a data source or a J2C connection factory.

Provider:

Specifies the resource provider that supplies the class information for this resource object.

Click the provider name to view a page where you can edit the configuration settings.

Description:

Specifies a text description of the resource.

Cache instances

An application uses a cache instance to store, retrieve, and share data objects within the dynamic cache.

Each cache instance can be configured independently for Java Naming and Directory Interface (JNDI) name, cache size, priority, and disk offload. Objects that are stored in a particular cache instance are not affected by other cache instances. This means that if you store an object named **object_1** with a value of `object_data` in *cache_instance_x*, you can also store an object with the same name, but different value in *cache_instance_y*.

Objects that are stored in a particular cache instance are available to applications on other servers by accessing a cache instance of the same name. The two servers must be within the same replication domain to share data.

There are two types of cache instances, object cache instances and servlet cache instances.

An object cache instance is a location in addition to the default shared dynamic cache where Java 2 Platform, Enterprise Edition (J2EE) applications can store, distribute, and share objects. After configuring object cache instances, you can use the DistributedMap or DistributedObjectCache interfaces in the `com.ibm.websphere.cache` package to programmatically access your cache instances.

See the Additional Application Programming Interfaces (APIs) for more information about the DistributedMap or DistributedObjectCache interfaces.

Servlet cache instances are locations in addition to the default dynamic cache where dynamic cache can store, distribute, and share the output and the side effects of an invoked servlet. By configuring a servlet cache instance, your applications have greater flexibility and better tuning of cache resources. The Java Naming and Directory Interface (JNDI) name that is specified for the cache instance in the administrative console maps to the <cache-instance> element in the `cachespec.xml` configuration file. Any <cache-entry> elements that are specified within a <cache-instance> element are created in that specific cache instance. Any <cache-entry> elements that are specified outside of a <cache-instance> element are stored in the default dynamic cache instance.

See Using servlet cache instances for more information.

Data access: Resources for learning

Use the following links to find relevant supplemental information about data access. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product, but it can be useful for understanding concepts or functions used by the application server. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information:

- “Technologies for data access”
- “Databases” on page 973
- “Tools” on page 973

Technologies for data access

- JDBC 3.0 API Documentation
- **Java Persistence API:**
  - Java Persistence API FAQ

  - Introduction to Spring 2 and JPA
- J2EE Connector Architecture Version 1.5 specification
- Enterprise JavaBeans Technology (Source for download of the Enterprise Javabeans 3.0 specification)
- Java 2 Platform, Enterprise Edition (J2EE)
- **Container-managed relationships:** Enterprise JavaBeans 2.0 Container-Managed Persistence Example. Although this article addresses the EJB 2.0 specification, you might find parts of it pertinent to your environment.
- **Resource adapters:** The J2EE Connector Architecture Resource AdapterDeveloper Technical Articles & Tips -- Articles: Database Access (Sun Developer Network)
- Java Management Extensions (JMX)
- **Miscellaneous articles from the Sun Developer Network and IBM developerWorks Web sites:**
  - Sharing connections in WebSphere Application Server V5 This article is still pertinent to WebSphere Application Server Version 6.0 and later. However, be aware that the container-managed authentication type is deprecated.
  - Database authentication in WebSphere Application Server V5 This article is still pertinent to WebSphere Application Server Version 6.0 and later. However, be aware that the container-managed authentication type is now deprecated.
  - Understanding WebSphere Application Server EJB access intents
- Supported hardware, software, and APIs

Databases

- **Cloudscape:**
  - IBM Cloudscape product information
  - IBM Cloudscape information center
- DB2 database software
- Informix

Tools

- Rational Application Developer for WebSphere Software

Developing data access applications

Data access applications allow you to manipulate data from outside sources for use within your application serving environment.

About this task

You can access data in various ways:
- using standard or extended APIs
- using container-managed persistence beans
- using bean-managed persistence beans, session beans, or Web components.
- using Service Data Objects (SDO)

1. Decide how to implement data access.

   The Enterprise JavaBeans (EJB) programming model provides several distinct server-side component types: entity, session, and message-driven beans, and servlets. Of these types, entity beans are typically used to model business components in an application. Entity beans have both *state* and *behavior*.

   The state of entity beans is persistent and is stored in a database. As changes are made to an entity bean, its state is kept in synchronization with the database record representing the bean. There are two types of entity beans provided by the EJB model and these two types differ in the mechanism

used to provide persistence. These two types of entity beans are *container-managed persistence* (CMP) beans and *bean-managed persistence* (BMP) beans.

- With BMP beans, the developer manually produces code to manage the persistent state of the bean.
- With CMP beans, the EJB container manages the persistent state of the bean. Persistent state management is a complex and difficult task; using CMP beans allows the developer to concentrate on business logic by delegating persistence behavior to the container.

  Typical examples of CMP beans are *Customer*, *Account*, and so on. Because CMP beans are objects, their data (state) is accessed using field accessors. For example, a *Customer* entity bean is likely to have fields such as *name* and *phoneNumber*. These pieces of data are accessed using the accessor methods *getName()/setName()* and *getPhoneNumber()/setPhoneNumber()*. As a developer, you are not concerned with how this data is eventually stored and retrieved from the backend database and can assume that the integrity of the data is maintained by the container.

See the “Developing enterprise beans” on page 132 article for information on developing entity beans.

Note:

- To maximize the efficiency of application requests to relational databases, consider using Structured Query Language in Java (SQLJ) when developing BMP and CMP beans. This option is available for applications that use the DB2 JDBC Universal Driver to access DB2 databases.

  The only exception to this driver requirement applies to SQLJ-backed BMP beans that access DB2 for z/OS; this schema requires the DB2 for z/OS Legacy Driver (required for the DB2 for z/OS Local JDBC Provider RRS).
- Also consider using cursor holdability for potential performance gains; see the “JDBC application cursor holdability support” on page 999 article for details.

An alternative to developing entity beans is using the Service Data Objects (SDO) framework, which is a unified framework for data application development. With SDO, you do not need to be familiar with a technology-specific API in order to access and utilize data. You need to know only one API, the SDO API, which lets you work with data from multiple sources, including relational databases, entity EJB components, XML pages, Web services, the Java Connector Architecture, JavaServer Pages, and more.

2. Look up a data source or connection factory using a resource reference (Looking up data sources with resource references for relational access). *Do not perform this step if you work with CMP beans. The EJB container handles this process for CMP beans.*

   To run applications on WebSphere Application Server, your code must use resource references to logically named data sources or connection factories. Mapping the resource references to actual resources is usually done at assembly time. The Application Server administrator configures those resources.

   - For relational database access, administrators configure a JDBC provider and associated data sources, which work with the embedded WebSphere Relational Resource Adapter.
   - For non-relational database access, administrators install a Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) resource adapter onto an application server and configure associated connection factories.
3. Get a connection to a data source or a connection factory. (See the ″Getting connections″ section of Connection life cycle for details.) *Do not perform this step if you work with CMP beans. The EJB container handles this process for CMP beans.*

   The connection management architecture for both relational and procedural access to enterprise information systems (EIS) is based on the Java EE Connector Architecture (JCA) specification. The Connection Manager (CM), which pools and manages connections within an application server, is capable of managing connections obtained through both resource adapters (RAs) defined by the JCA specification, and data sources defined by the JDBC Extensions Specification.
4. Use thread identity to assign an owner to the connection.

Extensions to data access APIs

If a single data access API does not provide a complete solution for your applications, use WebSphere Application Server extensions to achieve interoperability between both the JCA and JDBC APIs.

Applications that draw from diverse and complex resource management configurations might require use of both the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) API and the Java Database Connectivity (JDBC) API. However, in some cases the JDBC programming model does not completely integrate with the JCA (even though full integration is a foundation of the JCA specification). These inconsistencies can limit data access options for an application that uses both APIs. WebSphere Application Server provides API extensions to resolve the compatibility issues.

For example:

Without the benefit of an extension, applications using both APIs cannot modify the properties of a shareable connection after making the connection request, if other handles exist for that connection. (If no other handles are associated with the connection, then the connection properties can be altered.) This limitation stems from an incompatibility between the connection-configuration policies of the APIs:

The Connector Architecture (JCA) specification supports relaying to the resource adapter the specific properties settings at the time you request the connection (using the getConnection() method) by passing in a *ConnectionSpec* object. The ConnectionSpec object contains the necessary connection properties used to get a connection. After you obtain a connection from this environment, your application does not need to alter the properties. The JDBC programming model, however, does not have the same interface to specify the connection properties. Instead, it gets the connection first, then sets the properties on the connection.

WebSphere Application Server provides the following extensions to fill in such gaps between the JDBC and JCA specifications:

- WSDataSource interface - this interface extends the *javax.sql.DataSource* class, and enables a component or an application to specify the connection properties through the WebSphere Application Server *JDBCConnectionSpec* class to get a connection.
  - getConnection(JDBCConnectionSpec) - this method returns a connection with the properties specified in the JDBCConnectionSpec class.
  - For more information see the **WSDataSource** API documentation topic (as listed in the API documentation index).
- JDBCConnectionSpec interface - this interface extends the *com.ibm.websphere.rsadapter.WSConnectionSpec* class, which extends the *javax.resources.cci.ConnectionSpec* class. The standard ConnectionSpec interface provides only the interface marker without any get() and set() methods. The WSConnectionSpec and the JDBCConnectionSpec interfaces define a set of get() and set() methods used by the WebSphere Application Server run time. This interface enables the application to specify all the essential connection properties in order to get an appropriate connection. You can create this class from the WebSphere WSRRAFactory class. For more information see the **JDBCConnection** API documentation topic (as listed in the API documentation index).
- WSRRAFactory class - this is a factory class for the WebSphere Relational Resource Adapter, which allows the user to create a JDBCConnectionSpec object or other resource adapter related object. For more information see the **WSRRAFactory** API documentation topic (as listed in the API documentation index).
- WSConnection interface - this is an interface that allows users to call WebSphere proprietary methods on SQL connections; those methods are:
  - setClientInformation(Properties props) - See the Example: Setting client information with the setClientInformation(Properties) API topic for more information and examples of setting client information.

- Properties getClientInformation() - This method returns the properties object that is set using *setClientInformation(Properties)*. Note that the properties object returned is not affected by implicit settings of client information.
- WSSystemMonitor getSystemMonitor() - This method returns the SystemMonitor object from the backend database connection if the database supports System Monitors. The backend database will provide some connection statistics in the SystemMonitor object. The SystemMonitor object returned is wrapped in a WebSphere object (*com.ibm.websphere.rsadapter.WSSystemMonitor*) to shield applications from dependency on any database vendor code. See *com.ibm.websphere.rsadapter.WSSystemMonitor* Java documentation for more information. The following code is an example of using the *WSSystemMonitor* class:

```
import com.ibm.websphere.rsadapter.WSConnection;
...
try{
 InitialContext ctx=new InitialContext();
 // Perform a naming service lookup to get the DataSource object.
 DataSource ds=(javax.sql.DataSource]ctx.lookup("java:comp/jdbc/myDS");
} catch (Exception e) {;}

WSConnection conn=(WSConnection)ds.getConnection();
WSSystemMonitor sysMon=conn.getSystemMonitor();
if (sysMon!=null) // indicates that system monitoring is supported on the current backend database
 {
  sysMon.enable(true);
  sysMon.start(WSSystemMonitor.RESET_TIMES);
  // interact with the database
  sysMon.stop();
  // collect data from the sysMon object
 }
conn.close();
```

  The WSConnection interface is part of the *plugins_root*/com.ibm.ws.runtime_6.1.0.jar file.

Example: Using IBM extended APIs for database connections

Using the WSDataSource extended API, you can code your JDBC application to define connection properties through an object *before* obtaining a connection. This behavior increases the chances that the application can share a connection with another component, such as a CMP.

If your application runs with a shareable connection that might be shared with other container-managed persistence (CMP) beans within a transaction, it is recommended that you use the WebSphere Application Server extended APIs to get the connection. When you use these APIs, you cannot port your application to other application servers.

You can code with the extended API directly in your JDBC application; instead of using the DataSource interface to get a connection, use the WSDataSource interface. The following code segment illustrates WSDataSource:

```
import com.ibm.websphere.rsadapter.*;

...

// Create a JDBCConnectionSpec and set connection properties. If this connection is shared with
the CMP bean, make sure that the isolation level is the same as the isolation level that is mapped by
 the Access Intent defined on the CMP bean.

JDBCConnectionSpec connSpec = WSRRAFactory.createJDBCConnectionSpec();

connSpec.setTransactionIsolation(CONNECTION.TRANSACTION_REPEATABLE_READ);

connSpec.setCatalog("DEPT407");
```

```
//Use WSDataSource to get the connection

Connection conn = ((WSDataSource)datasource).getConnection(connSpec);
```

Example: Using IBM extended APIs to share connections between CMP beans and BMP beans

Within an application component that accesses data through JDBC objects (such as a bean-managed persistence (BMP) bean), you can use a WebSphere extended API to define connection properties through an object *before* obtaining a connection. This behavior increases the chances that the BMP bean can share a connection with a container-managed persistence (CMP) bean.

If your BMP bean runs with a shareable connection that might be shared with other container-managed persistence (CMP) beans within a transaction, it is recommended that you use the WebSphere Application Server extended APIs to get the connection. When you use these APIs, you cannot port your application to other application servers.

In this case, use the extended API WSDataSource interface rather than the DataSource interface. To ensure that both the CMP and bean-managed persistence (BMP) beans are sharing the same physical connection, define the same access intent profile on both the CMP and BMP beans. Inside your BMP method, you can get the right isolation level from the relational resource adapter helper class.

```
package fvt.example;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.ejb.CreateException;
import javax.ejb.DuplicateKeyException;
import javax.ejb.EJBException;
import javax.ejb.ObjectNotFoundException;
import javax.sql.DataSource;

// following imports are used by the IBM extended API
import com.ibm.websphere.appprofile.accessintent.AccessIntent;
import com.ibm.websphere.appprofile.accessintent.AccessIntentService;
import com.ibm.websphere.rsadapter.JDBCConnectionSpec;
import com.ibm.websphere.rsadapter.WSCallHelper;
import com.ibm.websphere.rsadapter.WSDataSource;
import com.ibm.websphere.rsadapter.WSRRAFactory;

/**
 * Bean implementation class for Enterprise Bean: Simple
 */

public class SimpleBean implements javax.ejb.EntityBean {
 private javax.ejb.EntityContext myEntityCtx;

 // Initial context used for lookup.

 private javax.naming.InitialContext ic = null;

 // define a JDBCConnectionSpec as instance variable

 private JDBCConnectionSpec connSpec;

 // define an AccessIntentService which is used to get
 // an AccessIntent object.

 private AccessIntentService aiService;

 // AccessIntent object used to get Isolation level

 private AccessIntent intent = null;
```

```
// Persitence table name

private String tableName = "cmtest";

// DataSource JNDI name

private String dsName = "java:comp/env/jdbc/SimpleDS";

// DataSource

private DataSource ds = null;

// bean instance variables.

private int id;
private String name;

/**
 * In setEntityContext method, you need to get the AccessIntentService
 * object in order for the subsequent methods to get the AccessIntent
 * object.
 * Other ejb methods will call the private getConnection() to get the
 * connection which has all specific connection properties
 */

public void setEntityContext(javax.ejb.EntityContext ctx) {
 myEntityCtx = ctx;

 try {
  aiService =
   (AccessIntentService) getInitialContext().lookup(
    "java:comp/websphere/AppProfile/AccessIntentService");
  ds = (DataSource) getInitialContext().lookup(dsName);
 }
 catch (javax.naming.NamingException ne) {
  throw new javax.ejb.EJBException(
   "Naming exception: " + ne.getMessage());
 }
}

 /**
 * ejbCreate
 */

public fvt.example.SimpleKey ejbCreate(int newID)
 throws javax.ejb.CreateException, javax.ejb.EJBException {
 Connection conn = null;
 PreparedStatement ps = null;

 // Insert SQL String

 String sql = "INSERT INTO " + tableName + " (id, name) VALUES (?, ?)";

 id = newID;
 name = "";

 try {
                   // call the common method to get the specific connection

  conn = getConnection();
 }
 catch (java.sql.SQLException sqle) {
  throw new EJBException("SQLException caught: " + sqle.getMessage());
 }
 catch (javax.resource.ResourceException re) {
  throw new EJBException(
```

```
    "ResourceException caught: " + re.getMessage());
  }

  try {
   ps = conn.prepareStatement(sql);
   ps.setInt(1, id);
   ps.setString(2, name);

   if (ps.executeUpdate() != 1) {
    throw new CreateException("Failed to add a row to the DB");
   }
  }
  catch (DuplicateKeyException dke) {
   throw new javax.ejb.DuplicateKeyException(
    id + "has already existed");
  }
  catch (SQLException sqle) {
   throw new javax.ejb.CreateException(sqle.getMessage());
  }
  catch (CreateException ce) {
   throw ce;
  }
  finally {
   if (ps != null) {
    try {
     ps.close();
    }
    catch (Exception e) {
    }
   }
  }
  return new SimpleKey(id);
 }

  /**
  * ejbLoad
  */

 public void ejbLoad() throws javax.ejb.EJBException {

  Connection conn = null;
  PreparedStatement ps = null;
  ResultSet rs = null;

  String loadSQL = null;

  try {
                    // call the common method to get the specific connection

   conn = getConnection();
  }
  catch (java.sql.SQLException sqle) {
   throw new EJBException("SQLException caught: " + sqle.getMessage());
  }
  catch (javax.resource.ResourceException re) {
   throw new EJBException(
    "ResourceException caught: " + re.getMessage());
  }

  // You need to determine which select statement to be used based on the
  // AccessIntent type:
  // If READ, then uses a normal SELECT statement. Otherwise uses a
  // SELECT...FORUPDATE statement
  // If your backend is SQLServer, then you can use different syntax for
  // the FOR UPDATE clause.

  if (intent.getAccessType() == AccessIntent.ACCESS_TYPE_READ) {
```

```
  loadSQL = "SELECT * FROM " + tableName + " WHERE id = ?";
 }
 else {
  loadSQL = "SELECT * FROM " + tableName + " WHERE id = ? FOR UPDATE";
 }

 SimpleKey key = (SimpleKey) getEntityContext().getPrimaryKey();

 try {
  ps = conn.prepareStatement(loadSQL);
  ps.setInt(1, key.id);
  rs = ps.executeQuery();
  if (rs.next()) {
   id = rs.getInt(1);
   name = rs.getString(2);
  }
  else {
   throw new EJBException("Cannot load id = " + key.id);
  }
 }
 catch (SQLException sqle) {
  throw new EJBException(sqle.getMessage());
 }
 finally {
  try {
   if (rs != null)
    rs.close();
  }
  catch (Exception e) {
  }
  try {
   if (ps != null)
    ps.close();
  }
  catch (Exception e) {
  }
  try {
   if (conn != null)
    conn.close();
  }
  catch (Exception e) {
  }
 }
}

       /**
        * This method will use the AccessIntentService to get the access intent;
        * then gets the isolation level from the DataStoreHelper
        * and sets it in the connection spec; then uses this connection
        * spec to get a connection which has the specific connection
        * properties.
        **/

private Connection getConnection()
 throws java.sql.SQLException, javax.resource.ResourceException, EJBException {

 // get current access intent object using EJB context
 intent = aiService.getAccessIntent(myEntityCtx);

 // Assume this bean only supports the pessimistic concurrency
 if (intent.getConcurrencyControl()
  != AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC) {
  throw new EJBException("Bean supports only pessimistic concurrency");
 }

 // determine correct isolation level for currently configured database
 // using DataStoreHelper
```

```
  int isoLevel =
   WSCallHelper.getDataStoreHelper(ds).getIsolationLevel(intent);
   connSpec = WSRRAFactory.createJDBCConnectionSpec();
  connSpec.setTransactionIsolation(isoLevel);

  // Get connection using connection spec
  Connection conn = ((WSDataSource) ds).getConnection(connSpec);
  return conn;
 }
```

Recreating database tables from the exported table data definition language

When the WebSphere Application Server deployment tooling deploys an EJB jar file containing container-managed persistence (CMP) enterprise beans, it selects the target database and creates a corresponding *Table*.ddl file. This file contains the SQL statement necessary to generate the database table for your CMP beans.

About this task

The following steps demonstrate the process for creating tables in DB2.

1. Extract the *Table*.ddl file from your CMP enterprise bean JAR file and save it on your database server.
   - Save the file to a temporary directory on your work station. Transfer the file to a data set on your DB2 for z/OS system.
2. Run the *Table*.ddl file.
   - Specify the data set as the input data set to SPUFI, and run the program.

Results

The database tables are created.

CMP bean associated technologies

WebSphere Application Server delivers container-managed persistence (CMP) services beyond the standards set by the Enterprise JavaBean (EJB) specification.

According to the specification, the EJB container synchronizes the state of CMP beans with the underlying database, and manages the relationships (container-managed relationships, or CMR's) among entity beans. Thus the EJB specification relieves bean developers from writing any database-specific code; instead, they can focus on writing business logic. WebSphere Application Server offers the following additional CMP functions to increase development efficiency even more, as well as optimize the run-time performance of business logic:

Entity bean inheritance

Inheritance is a key aspect of object-oriented software development and is a capability currently missing from the EJB specification.

The use of inheritance enables a developer to define fields, relationships, and business logic in a superclass entity bean that are inherited by all subclasses. See the section *EJB inheritance* of the Rational Application Developer (RAD) documentation for details on using inheritance with WebSphere Application Server and entity beans.

Access Intent Policies

Access intent policies provide Java Platform, Enterprise Edition (Java EE) application developers the mechanism by which they can indicate the intent of an application's interaction with the essential state for entity beans in order that the persistence mechanisms can make appropriate optimizations. For example, if it is known that an entity is not updated during the course of a

transaction, then the persistence management is able to ease up on the concurrency control and still maintain data integrity by disallowing update operations on that bean for the duration of the transaction.

Caching data across transactions

Data caching across transactions is a configurable option set by the bean deployer that can greatly improve performance. Essentially, this is for data that changes infrequently. The option is known as *LifetimeInCache*. The data for an entity configured for lifetime in cache is stored in a cache until its specified lifetime expires. Requests on the entity during that configured lifetime use the cached data, and do not result in the execution of queries against the underlying data store. Lifetime can be expressed as time elapsed since the data was retrieved from the data store or until a specific time of day or week. The *LifetimeInCache* value can be one of the following:

Off The LifetimeInCache setting is ignored. Beans of this type are only cached in a transaction scoped cache. The cached data for this instance is not valid when the transaction is completed.

ElapsedTime

The value in the LifetimeInCache setting is added to the current time when the transaction (in which the bean instance is retrieved) is completed. The cached data for this instance is not valid after this time. The value of the LifetimeInCache setting can add up to minutes, hours, days, and so on.

ClockTime

The value of LifetimeInCache represents a particular time of day. The value is added to the immediately preceding or following midnight to calculate a future time value, which is then treated as for Elapsed Time. Using this setting enables you to specify that all instances of this bean type have their cached data invalidated at a specific time no matter when the data were retrieved.

The use of preceding or following midnight to calculate a future time value depends on the value of LifetimeInCache. If LifetimeInCache plus preceding midnight is earlier than the current time, then the following midnight is used.

When you use the ClockTime setting, the value of LifetimeInCache must not represent more than 24 hours. If it does, the cache manager subtracts increments of 24 hours from it until a value less than or equal to 24 hours is achieved. To invalidate data at 12 midnight, you set LifetimeInCache to zero (0).

WeekTime

This setting is similar to ClockTime, except the value of LifetimeInCache is added to the preceding or following Sunday midnight (actually, 11:59 PM on Saturday plus 1 minute). In this case, the LifetimeInCache value can represent more than 24 hours, but not more than 7 days.

See the *LifetimeInCache* help sections of the assembly tool for more details.

Note:

Because the data used by an entity bean can be loaded by previous transactions, if you configure the bean as LifeTimeInCache, the isolation level and update lock (access intent policies) for the bean are lost for the current transaction. This can cause data integrity problems if your application has logic to calculate information from read-only data, and then save the result in another bean. This makes it important to perform read-read consistency checking to ensure the data get locked properly if loading the data from in-memory cache; otherwise, data is updated to the database without knowing the underlining data is changed, causing previous changes to be lost. For more information, see “Configuring read-read consistency checking with an assembly tool” on page 199.

Read-only entity beans

Declaring entity beans as read-only potentially increases the performance enhancement offered by caching. Both features operate on the same principle: to minimize the overhead incurred by

frequent reloading of entity beans from data in persistent storage. When you designate entity beans as read-only, you can specify the reload requirements and frequency, according to the needs of your application.

To use this function, you declare the bean type as read-only by selecting a particular set of bean caching options, through a selection list within the assembly tooling. See “Developing read-only entity beans” on page 173 for details.

Container-managed persistence restrictions and exceptions

Some external software that directly impact your applications can limit container-managed persistence (CMP) features. However, you can work around these limitations.

In each case, only very specific behaviors of the software place restrictions on your CMP beans. The following tips help you prevent these behaviors.

CMP deployment and Sybase IMAGE type restriction

When deploying enterprise beans with container managed persistence (CMP) types that are non-primitive and do not have a natural JDBC mapping, the deployment tool maps the CMP type to a binary type in the database, where it is stored as a serialized instance. For Sybase, the tool uses the JDBC type *LONG VARBINARY*. The Sybase driver maps *LONG VARBINARY* to the native type *IMAGE*.

Although the type *VARBINARY* has fewer restrictions than *IMAGE* in Sybase, you cannot use it because it is limited to a size of 255 bytes, which is too small for typical serialized Java objects.

The specific restrictions on the IMAGE type are:

- You cannot use the *IMAGE* type in the *WHERE* clause of an SQL query. You can encounter this restriction whenever an enterprise bean contains an EJB-QL query that has a CMP type in the *WHERE* clause, which maps to the *IMAGE* type in the Sybase relational database.
- You cannot use *IMAGE* type in select queries marked *DISTINCT*. This situation arises in these user scenarios:
  - When the *DISTINCT* key word is specified in an EJB-QL select query having a Java type mapping to IMAGE.
  - When Enterprise beans have finder and ejbSelect() methods returning java.util. Set and have CMP types mapping to *IMAGE*.

To work around this restriction, edit the EJB mappings in the Rational Application Developer toolset and do either of the following:

- If you are **sure** that the serialized instance of the CMP type is **never** larger than 255 bytes, you can change the CMP type mapping from *IMAGE* or *LONG VARBINARY* to *VARBINARY*.
- Map the CMP type to multiple RDB fields through a composer. For example, if the CMP type is a Java object X with an int field and a string field, then map X to two RDB fields *INTEGER* and *VARCHAR*, using a composer. Refer to the Rational Application Developer documentation for more information about using composers.

A ClassCastException exception occurs when running CMP 1.1 beans

If you created your Enterprise JavaBeans (EJB) application using Rational Application Developer or WebSphere Studio Application Developer Integration Edition, Version 4.0.x , and the application contains container managed persistence (CMP) 1.1 beans with associations (relationships), you might receive a *java.lang.ClassCastException* exception when you run your application on WebSphere Application Server.

The cast operation generated by Rational Application Developer or WebSphere Studio Application Developer Integration Edition, Version 4.0.x, does not use the *javax.rmi.PortableRemoteObject.narrow(...)*

object to convert the remote object to the remote interface of CMP beans in the *XToYLink.java* (or *YToXLink.java*) class where X and Y are CMP 1.1 beans.

Recommended response

1. Locate the following methods in all link classes, for example, XToYLink.java and YToXLink.java where X and Y are CMP 1.1 beans:

   ```
   public void secondaryAddElementCounterLinkOf(javax.ejb.EJBObject anEJB)
   public void secondaryRemoveElementCounterLinkOf(javax.ejb.EJBObject anEJB)
   public void secondarySetCounterLinkOf(javax.ejb.EJBObject anEJB)
   ```

2. Add the javax.rmi.PortableRemoteObject.narrow(...) object to convert the remote object to the remote interface of CMP beans.

For example, change the following original method:

```
public void secondaryAddElementCounterLinkOf(javax.ejb.EJBObject anEJB) throws java.rmi.RemoteException {
        if (anEJB != null)
               ((X) anEJB).secondaryAddY((Y) getEntityContext().getEJBObject());
```

to:

```
public void secondaryAddElementCounterLinkOf(javax.ejb.EJBObject anEJB) throws java.rmi.RemoteException {
        if (anEJB != null)
             ((X) anEJB).secondaryAddY((Y)
javax.rmi.PortableRemoteObject.narrow(getEntityContext().getEJBObject(), Y.class));
```

Application performance and entity bean behavior

WebSphere Application Server allows you to override two behaviors that are required by the EJB specification, because your application might benefit from handling these aspects of bean data management in a slightly different manner.

Application-managed persistent store synchronization for findBy methods

Sections 10.5.3 and 12.1.4.2 of the EJB 2.0 and 2.1 specifications require that prior to running a query as part of any findBy method (except for findByPrimaryKey), the EJB container writes out to persistent storage the state of any entity beans of the type that are enlisted in the current transaction. Stated another way, the container performs the following actions:

1. Creates a list of beans that are both enlisted in the current transaction and are of the same type that the findby method is returning
2. Stores the state of these enterprise beans to persistent storage before running the query

If the state of an EJB instance is not altered in the current transaction, the store operation is skipped for that instance. This practice ensures that the query is performed on the most current state of all the persistent data, reducing the chance of data integrity issues.

However, there are scenarios where it is inefficient and wasteful for the EJB container to automatically perform this action on every findBy method. Examples of this would be where the application itself ensures that the most current data is used on findBy queries, or where the application can tolerate some non-current data as part of the query results.

WebSphere Application Server allows you to initiate the synchronization process under application control, and to disable the container-managed synchronization for specific EJB types within your application. Careful use of these functions can improve the performance of your application without sacrificing data integrity. Details are covered in “Manipulating the synchronization of entity beans and datastores” on page 985.

Avoiding ejbStore invocations on non-modified entity bean instances

The EJB specification requires that the EJB container invoke the user-provided ejbStore method on all entity beans within a transaction when that transaction is committed. For container-managed persistence (CMP) beans (as opposed to bean-managed persistence beans) this operation is usually unnecessary, because this method on CMP beans is often empty. Even in cases where the method is not empty, the application might only require the method to be called if the bean's persistent state is modified during the current transaction.

WebSphere Application Server provides a mechanism for you to indicate if you want this behavior for specific EJB types within the application. Details are covered in "Avoiding ejbStore invocations on non-modified EntityBean instances" on page 986.

Manipulating the synchronization of entity beans and datastores

You can indicating that a particular EJB type should not synchronize its state to persistent storage prior to each findBy invocation by using environment variables or a marker interface.

About this task

There are two options available for indicating that a particular EJB type should not synchronize its state to persistent storage prior to each findBy invocation:

- Set an EJB environment variable within the bean's deployment descriptor
- Have the bean implementation class implement a marker interface. This second technique is especially useful if you have a number of bean implementations that all extend a single root class; in this case you can have the root class implement the marker interface, causing all beans that extend this class to inherit the behavior as well.
- **To use the EJB environment variable technique**, edit the EJB deployment descriptor using any standard Java Platform, Enterprise Edition (Java EE) development tool. For information on your tool options, consult Assembly tools.
  1. Start the tool.
  2. Select the EJB deployment descriptor of the bean with which you want to work.
  3. Create an EJB environment variable with the name **com/ibm/websphere/ejbcontainer/disableFlushBeforeFind**.
  4. Set the type of this variable to **java.lang.Boolean**.
  5. Set the value to True to prevent the pre-find synchronization, or False to enable the default behavior.
  6. Save your changes.
- **To use a marker interface**, code your bean implementation class to implement the **com.ibm.websphere.ejbcontainer.disableFlushBeforeFind** interface. The bean implementation class need not directly implement the interface; any parent class can implement the interface. See the **com.ibm.websphere.ejbcontainer** package in the **Reference** > **Developer** > **API documentation** section of the information center.

  Ensuring data integrity for queries performed during a transaction

  If you choose to disable the automatic pre-find synchronization for certain bean types, it is very important that your application use other means to ensure that queries performed during the transaction are not performed on data that might no longer be valid. You can use the flushCache method on the com.ibm.websphere.ejbcontainer.EJBContextExtension class (an extension of javax.ejb.EJBContext) to perform a manual synchronization to persistent storage at times that are defined by the application. For more information on EJBContextExtension and its related classes SessionContextExtension, EntityContextExtension and MessageDrivenContextExtension, see the **com.ibm.websphere.ejbcontainer** package in the **Reference** > **Developer** > **API documentation** section of the information center.

Avoiding ejbStore invocations on non-modified EntityBean instances

You can configure your EntityBean instances to bypass an invocation of the ejbStore method if they have not been modified during the current transaction.

About this task

There are two options available for indicating that a particular EJB type should only have its ejbStore method invoked if the bean has been modified during the current transaction:

- Set an EJB environment variable within the bean's deployment descriptor
- Have the bean implementation class implement a marker interface. This second technique is especially useful if you have a number of bean implementations that all extend a single root class; in this case you may have the root class implement the marker interface, causing all beans that extend this class to inherit the behavior as well.
- **To use the EJB environment variable technique**, edit the EJB deployment descriptor using any standard Java Platform, Enterprise Edition (Java EE) development tool. Use the following steps as a guide. (For information on your tool options, consult the Assembly tools article.)
  1. Start the tool.
  2. Select the EJB deployment descriptor of the bean you want to work with.
  3. Create an EJB environment variable with the name **com/ibm/websphere/ejbcontainer/ disableEJBStoreForNonDirtyBeans**.
  4. Set the type of this variable to **java.lang.Boolean.**
  5. Set the value to True to avoid the ejbStore invocation, or False to enable the default behavior.
  6. Save your changes.
- **To use a marker interface**, code your bean implementation class to implement the **com.ibm.websphere.ejbcontainer.DisableEJBStoreForNonDirtyBeans** interface. The bean implementation class need not directly implement the interface; any parent class can implement the interface. See the **com.ibm.websphere.ejbcontainer** package in the **Reference** > **Developer** > **API documentation** section of the information center.

The benefits of using resource references

WebSphere Application Server requires your code to reference application server resources (such as data sources or J2C connection factories) through logical names, rather than access the resources directly in the Java Naming and Directory Interface (JNDI) name space. These logical names are called *resource references*.

Application Server requires use of resource references for the following reasons:

- If application code looks up a data source directly in the JNDI naming space, every connection that is maintained by that data source inherits the properties that are defined in the application. Consequently, you create the potential for numerous exceptions if you configure the data source to maintain shared connections among multiple applications. For example, an application that requires a different connection configuration might attempt to access that particular data source, resulting in application failure.
- It relieves the programmer from having to know the name of the actual data source or connection factory at the target application server.
- You can set the default isolation level for a data source through resource references. With no resource reference you get the default for the JDBC driver you use.

Example of using a resource reference

The following code invokes a data source by creating a place holder for it through the lookup method. Using the logical name *jdbc/Section*, the code stores the place holder in the JNDI subcontext

java:comp/env/; hence *jdbc/Section* becomes a resource reference. (The subcontext *java:comp/env/* is the name space that WebSphere Application Server provides exclusively for object references within application code.)

```
javax.sql.DataSource specificDataSource  =
   (javax.sql.DataSource) (new InitialContext()).lookup("java:comp/env/jdbc/Section");
//The method InitialContext()).lookup creates the logical name, or resource reference, jdbc/Section.
```

Generally, an actual data source is configured later as an administrative task.

The logical name *jdbc/Section* is officially declared as a resource reference in the application deployment descriptor. You can then associate the resource reference with the JNDI name of the actual data source in several ways:

- If you know the data source JNDI name at the point of application assembly, specify the name on the resource references Bindings page.
- Specify the data source JNDI name during application deployment.
- Map the resource reference to the data source JNDI name when you configure the application after deployment.

This act of association is called *binding* the resource reference to the data source.

See the article Application bindings for information on all types of required resource bindings.

Requirements for setting isolation level

This article discusses the criteria and effects of setting isolation levels for data access components that comprise EJB 2.x modules.

In an Enterprise JavaBean (EJB) 1.1 module, you can set the isolation level at the method level or bean level. This capability also applies to container-managed persistence (CMP) 1.1 beans that you assemble into *EJB 2.x modules*. (WebSphere Application Server permits the deployment descriptor of a CMP bean to declare the version level of 1.1, regardless of the overall module version.)

However, the ability to set isolation level at the method or bean level does **not** apply to other enterprise beans within an EJB 2.x module, including *CMP 2.x beans*. WebSphere Application Server Version 5.0 removed this capability from EJB 2.0 modules to deliver an architecture that ultimately provides more efficient connection use.

Consequently, later versions of the product enforce the following restrictions on declaring isolation level for CMP 2.x beans—as well as session beans, message-driven beans, and bean managed persistence (BMP) beans that you assemble into EJB 2.x modules:

- You cannot specify isolation level on the EJB method level or bean level.
- If you configure a JDBC application, a bean-managed persistence (BMP) bean, or a servlet to participate in global transactions, any connection that is shared cannot accept a user-specified isolation level. WebSphere Application Server can only set a user-specified isolation level on a connection that is not shared within a global transaction. *Generally, you want to refrain from specifying isolation levels on shareable connections.*

The configuration for the isolation level is determined by the type of bean that is used by the component:

Isolation level on connections used by 2.x CMP beans

In a EJB 2.x module, when a CMP 2.x bean uses a new data source to access a backend database, the isolation level is determined by the WebSphere Application Server run time, based on the type of access intent assigned to the bean or the calling method. Other non-CMP connection users can access this same data source and also use the access intent and application profile support to manage their concurrency control.

Connections used by other 2.x enterprise beans and other non-CMP components

For all other JDBC connection instances (connections other than those used by CMP beans), you can specify an isolation level on the data source resource reference. For shareable connections that run in global transactions, this method is the only way to set the *isolationLevel* for connections. Trying to directly set the isolation level through the *setTransactionIsolation()* method on a shareable connection that runs in a global transaction is not allowed. To use a different isolation level on connections, you must provide a different resource reference. Set these defaults through your assembly tool.

Each resource reference associates with one isolation level. When your application uses this resource reference Java Naming and Directory Interface (JNDI) name to look up a data source, every connection returned from this data source using this resource reference has the same isolation level.

Components needing to use shareable connections with multiple isolation levels can create multiple resource references, giving them different JNDI names, and have their code look up the appropriate data source for the isolation level they need. In this way, you use separate connections with the different isolation levels enabled on them.

It is possible to map these multiple resource references to the same configured data source. The connections still come from the same underlying pool, however; the connection manager does not allow sharing of connections requested by resource references with different isolation levels. Consider the following scenario:

- A data source is bound to two resource references: *jdbc/RRResRef* and *jdbc/RCResRef*.
- RRResRef has the *RepeatableRead* isolation level defined. RCResRef has the *ReadCommitted* isolation level defined.

If your application wants to update the tables or a BMP bean updates some attributes, it can use the *jdbc/RRResRef* JNDI name to look up the data source instance. All connections returned from the data source instance have a RepeatableRead isolation level. If the application wants to perform a query for read only, then it is better to use the *jdbc/RCResRef* JNDI name to look up the data source.

If you do not specify the isolation level:

The product does not require you to set the isolation level on a data source resource reference for a non-CMP application module. If you do not specify isolation level on the resource reference, or if you specify TRANSACTION_NONE, the WebSphere Application Server run time uses a default isolation level for the data source. Application Server uses a default setting based on the JDBC driver.

For most drivers, WebSphere Application Server uses an isolation level default of TRANSACTION_REPEATABLE_READ. For Oracle drivers, however, Application Server uses an isolation level of TRANSACTION_READ_COMMITTED. Use the following table for quick reference:

Database:	DB2	Oracle	Sybase	Informix	Apache Derby	SQL Server
Default isolation level: *(for connections used by non-CMP entities)*	RR	RC	RR	RR	RR	RR

- **Note:** These same default isolation levels are used in cases of direct JNDI lookups of a data source.

- RR = JDBC Repeatable read (TRANSACTION_REPEATABLE_READ)
- RC = JDBC Read committed (TRANSACTION_READ_COMMITTED)

To customize the default isolation level, you can use the webSphereDefaultIsolationLevel custom property for the data source. In most cases you should define the isolation level in the deployment descriptor when you package the EAR file, but in certain situations you might need to customize the default isolation level. This property will have no effect if any of the above options are used, and this custom property is provided for those situations in which there is no other means of setting the isolation level.

Use the following values for webSphereDefaultIsolationLevel custom property:

Possible values	JDBC isolation level	DB2 isolation level
8	TRANSACTION_SERIALIZABLE	Repeatable Read (RR)
4 (default)	TRANSACTION_REPEATABLE_READ	Read Stability (RS)
2	TRANSACTION_READ_COMMITTED	Cursor Stability (CS)
1	TRANSACTION_READ_UNCOMMITTED	Uncommitted Read (UR)

To define this custom property for a data source:

1. Click **Resources** → **JDBC provider** → ***JDBC_provider***.
2. Click **Data sources** in the Additional Properties section.
3. Click the name of the data source.
4. Click **Custom properties**.
5. Create the webSphereDefaultIsolationLevel custom property.
   a. Click **New**.
   b. Enter webSphereDefaultIsolationLevel for the name field.
   c. Enter one of the possible values in the value field.

Application Server sets the isolation level by prioritizing the available settings. Application Server will set the isolation level based on the values for the following, in this order:

1. Resource reference isolation level
2. Isolation level that is specified by the access intent policy
3. Custom property that configures an isolation level
4. Application Server's default setting.

Data source lookups for enterprise beans and Web modules:

During either application assembly or deployment, you must bind the resource reference to the JNDI name of the actual resource in the run-time environment. You can take this action in the assembly tool or as one of the steps during installation of the application EAR file.

Bean-managed persistence bean: When developing your bean-managed persistence (BMP) bean you generally lack knowledge about the name of the data source on the target application server. In your code, do not look up the data source directly. Instead, you look up the resource reference from the *java:comp/env namespace* file. Let us assume that you look up the resource reference named *ref/ds* as illustrated in the code below.

```
javax.sql.DataSource dSource = (javax.sql.DataSource)((new InitialContext()).lookup("java:comp/env/ref/ds"));
```

In the assembly tool, you specify the name **ref/ds** in the Resource Reference page on the General Tab. If you know the name of the data source you can specify it in this Resource References page on the Bindings Tab. Note that if you do not specify it here , you must provide this Java Naming and Directory Interface (JNDI) name when you install the application EAR file.

Container-managed persistence bean: The data source binding process for the container-managed persistence (CMP) bean is the same process that you perform for bean-managed persistence (BMP) beans. Use the data source JNDI name as a WebSphere binding property for each bean during application assembly.

Servlets and JavaServer Pages Files: In a servlet application, you look up the data source exactly as you look it up in the BMP bean case.

Direct and indirect JNDI lookup methods for data sources:

You can use a direct or indirect method for the Java Naming and Directory Interface (JNDI) name (such as jdbc/DataSource) to look up a data source.

Direct

When you use a JNDI name such as jdbc/myDatasource, the application server assigns default values to the resource reference data. An informational message resembling the following is logged to document the default values:

```
[10/5/07 11:40:38:468 CDT] 0000002e ConnectionFac W   J2CA0294W: Direct JNDI lookup of resource jdbc/myDatasource.
   The following default values are used:

[Resource-ref CMConfigData key items]
res-auth:                 1 (APPLICATION)
res-isolation-level:      0 (TRANSACTION_NONE)
res-sharing-scope:        true (SHAREABLE)
loginConfigurationName:   null
loginConfigProperties:    null

[Resource-ref non-key items]
isCMP1_x:                 false (not CMP1.x)
isJMS:                    false (not JMS)
commitPriority            0
Java EE Name:                not set
Resource ref name:        not set
isCMP:                    false (not set)
```

The first of these attributes, *res-auth*, dictates what type of authentication is done. This default setting says that the component-managed authentication alias is used if you do not specify an activation specification or you do not specify the username and password on the getConnection call. It says that the container-managed alias is not used.

The second of these settings, *res-isolation-level*, says that the isolation level is set to the ″default″ settings. For an enterprise bean, you can set this in the Enterprise JavaBeans (EJB) bean itself. For a servlet getting a connection, this results in the isolation level being Repeatable_Read. This is a fairly restrictive isolation level. This can lead to lowered performance, because application requests will lock more rows than with a less restrictive isolation level.

Finally, the *res-sharing-scope* is set to **Shareable**, meaning a Shareable connection is used. For some applications, a Shareable connection is fine. For others, in particular those servlets that get multiple connections within a single service() method, it is not.

To avoid any surprises that might accompany these settings, you should change your application to use an indirect JNDI name instead of the direct JNDI name, and you should create a resource reference.

Indirect

To use values that are different from the defaults, use an assembly tool to define your resource reference. The resource reference can also be created in the EJB Deployment Descriptor (ejb-jar.xml), Web Deployment Descriptor (web.xml), or Application Client Deployment Descriptor (application-client.xml)

editors using an assembly tool. After you define the resource reference, you can do an indirect JNDI lookup (using the java:comp/env context). Then the values for the resource reference properties that are defined in the resource reference are used and the J2CA0122I message no longer appears. Read the topic on creating a resource reference for more information.

Access intent and isolation level:

The *access intent* service enables developers to precisely tune the management of application persistence.

Access intent enables developers to configure applications so that the EJB container and its agents can make performance optimizations for entity bean access. Entity beans and entity bean methods are configured with access intent policies. A policy is acted upon by either the combination of the WebSphere EJB container and Persistence Manager (for container-managed persistence (CMP) entities) or by bean-managed persistence (BMP) entities directly. Note that access intent policies apply to entity beans only.

Predefined access intent policies

Seven predefined access intent policies are available. The policies are composed of different attributes. The *access type* is of primary interest and controls the isolation level, lock type, and duration of locks obtained when bean data is read from the database.

A pessimistic access type indicates to hold locks for the duration of the transaction under which the data loads. An optimistic type indicates to drop locks immediately after the data is read from the backend. A *read* type indicates that the run time must not allow updates to the data; any attempt to do so on data read under a *read* type results in an exception. *Update* types permit you to change data.

Though a pessimistic update policy is designed to hold update locks on data records, it does not block threads with other policies that try to access the same data records. When two threads that run pessimistic update policies access a given record, they serialize (but not block) other threads that run pessimistic read or optimistic policies and try to access the same record.

The seven access intent policies and their attribute definitions follow:

wsPessimisticUpdate

- Access type = Pessimistic update
- Collection scope = Transaction
- Collection increment = 1
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsOptimisticUpdate

- Access type = Optimistic update
- Collection scope = Transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsOptimisticRead

- Access type = Optimistic read
- Collection scope = Transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticRead

- Access type = Pessimistic read
- Collection scope = Transaction
- Collection increment = 25

- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticUpdate-Exclusive

- Access type = Pessimistic update
- Exclusive = true
- Collection scope = Transaction
- Collection increment = 1
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticUpdate-NoCollision

- Access type = Pessimistic update
- No collision = true
- Collection scope = Transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticUpdateWeakestLockAtLoad

- ***default policy**
- Access type = Pessimistic Update
- Promote = true
- Collection scope = transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

Note that to support connection sharing, you must ensure that all data loaded in the same transaction is under the same isolation level. Verify that all participating methods that drive loads are configured with either a pessimistic access type or an optimistic access type.

Access intent -- isolation levels and update locks:

WebSphere Application Server access intent policies provide a consistent way of defining the isolation level for CMP bean data across the different relational databases in your environment.

Within a deployed application, the combination of an access intent policy *concurrency definition* and *access type* signifies the isolation level value that Application Server sets on a database connection. (See the articles cejb_cncr.dita and cdat_accint.dita for more information on concurrency and access type.) This combination of properties also signifies the update lock flag that Application Server passes to the database through a JDBC prepared statement.

Databases do not provide as many isolation level definitions as WebSphere Application Server. Databases define an isolation level as one of only three types. Furthermore, only one parameter indicates the type of isolation level that the databases set on incoming connections. Each of the three types can be represented by a *different* parameter value, as determined by each database vendor. For example, one database might define an isolation level as RR (JDBC Repeatable read), whereas a different database might define the same isolation level as RC (JDBC Read committed).

Because of this inconsistency, WebSphere Application Server does not map access intent policies to the parameter values. Instead, Application Server maps access intent policies to the types of isolation level that are common across all database vendors.

The following matrix shows how access intent policies correspond to different database isolation levels and update lock settings:

Access Intent profile	Isolation level						Update lock implementation
	DB2	Oracle*	SyBase	Informix	Apache Derby	SQL Server	
wsPessimisticUpdate-Weakest LockAtLoad (Default policy)	RR	RC	RR	RR	RR	RR	No (*Oracle, Yes)
wsPessimisticUpdate	RR	RC	RR	RR	RR	RR	Yes
wsPessimisticRead	RR	RC	RR	RR	RR	RR	No
wsOptimisticUpdate	RC	RC	RC	RC	RC	RC	No
wsOptimisticRead	RC	RC	RC	RC	RC	RC	No
wsPessimisticUpdate No-Collisions	RC	RC	RC	RC	RC	RC	No
wsPessimisticUpdate-Exclusive	S	S	S	S	S	S	Yes

- RC = JDBC Read Committed
- RR = JDBC Repeatable Read
- S = JDBC Serializable
- * Oracle does not support JDBC Repeatable Read (RR). Therefore, wsPessimisticUpdate-weakestLockAtLoad and wsPessimisticUpdate behave the same way on Oracle as do wsPessismisticRead and wsOptimisticRead. Because of an Oracle restriction, the OracleXADataSource JDBC class cannot run with an S transaction isolation level. Therefore, you cannot use this class to run an application containing enterprise beans with access intent policies that are configured to cause the bean to load with S isolation.
- Setting access intent policies per EJB method support is deprecated for Version 6.0. It is recommended that you set access intent only for the entire bean.

Note: MS SQL Server 2005 offers a new option for the Read Committed isolation level and a new option for the Serializable isolation level:

- Read Committed with Snapshots
- Transaction Snapshot (for Serializable)

Both options use optimistic locking. To use Read Committed with Snapshots instead of Read Committed, enable the READ_COMMITTED_SNAPSHOT setting for the database according to the MS SQL Server 2005 documentation. To use Transaction Snapshot instead of Serializable, configure the custom data source property, snapshotSerializable, to ″true″ and enable the ALLOW_SNAPSHOT_ISOLATION setting for the database according to the MS SQL Server 2005 documentation.

Structured Query Language (SQL) keywords and restrictions

The following table shows which SQL keywords are used during update intent locking, as well as any restrictions imposed on the SQL.

Database	SQL syntax used for locking update	join restrictions	order by restrictions	subselect restrictions	aggregation restrictions
DB2	FOR UPDATE OF	not allowed	not allowed	not allowed	not allowed
DB2 UDB for iSeries (V5R3 and earlier)	FOR UPDATE OF	not allowed	allowed with limitations*	allowed with limitations*	not allowed

Database	SQL syntax used for locking update	join restrictions	order by restrictions	subselect restrictions	aggregation restrictions
DB2 UDB for iSeries (V5R4 and later)	WITH RS/RR USE AND KEEP EXCLUSIVE LOCKS	not allowed	allowed with limitations*	allowed with limitations*	not allowed
DB2 on z/OS V8.x	WITH RS/RR USE AND KEEP UPDATE LOCKS	none	none	none	none
DB2 UDB workstation V8.2	WITH RS/RR USE AND KEEP UPDATE LOCKS	none	none	none	none
Oracle	FOR UPDATE	none	none	none	none
Apache Derby	FOR UPDATE OF	not allowed	not allowed	not allowed	not allowed
Informix	FOR UPDATE	not allowed	not allowed	not allowed	not allowed
Sybase	FOR UPDATE	not allowed	not allowed	not allowed	not allowed
Sqlserver	UPDLOCK	not allowed	not allowed	not allowed	not allowed

* Note: For details on the limitations for these permitted SQL restrictions, refer to the DB2® Universal Database™ for iSeries SQL Reference. You can find this document in the iSeries Information Center, Version 5 Release 4. In the Contents navigation area, click **Database** > **Reference** > **SQL Reference**.

Custom finder SQL dynamic enhancement:

To ensure data integrity for applications using custom finders defined on Enterprise JavaBeans (EJB) version 1.1 home interfaces, WebSphere Application Server Version 6.x uses custom finder Structured Query Language (SQL) dynamic enhancement to maintain correct SQL locking semantics.

WebSphere Application Server uses SQL clauses applied to the custom finder SQL statements for those custom finders defined with the *Update* attribute and certain method-level isolation level settings. These dynamic enhancements are applied only if the backend data store supports these clauses.

This support takes affect at run time when the run time attempts to execute container-managed persistence (CMP) persistence operations associated with the custom finders. To ensure that the SQL dynamic enhancements occur correctly for custom finders defined on an EJB version 1.1 home interface accessing a backend data store that requires the special SQL locking clauses, WebSphere Application Server provides new Java Virtual Machine (JVM) and bean (module) properties. These properties enable you to indicate which custom finders should be enhanced, provided the backend store supports the SQL clauses. For more information about these properties, see Custom finder SQL dynamic enhancement properties.

There are several important items to consider when using this functionality:

- This support **only** applies to EJB version 1.1 CMP Custom Finder methods
- Option A CMP beans and CMP beans involved in an inheritance relationship are not supported
- Applications using this capability in WebSphere Application Server for z/OS Version 4.x continue to function, but you must address some compatibility issues:
  - The default behavior of WebSphere Application Server Version 5.x and above is the opposite of the Version 4.x product, that is, the default for 5.x and above is **not** to enhance custom finder SQL statements unless directed to by specific settings. If your WebSphere Application Server for z/OS installation relies on the automatic dynamic enhancement of all custom finders in all applications installed, you must set the *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* indicator to **all**.

– If an application contains a bean which has the *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* indicator set into its env-var settings, that indicator continues to be used, provided the dynamic SQL enhancement features of the product at Version 5.x and above are enabled. For more information, see Custom finder SQL dynamic enhancement properties

Custom finder SQL dynamic enhancement properties:

Use this page to modify custom finder SQL dynamic enhancement properties settings.

To ensure that the Structured Query Language (SQL) dynamic enhancements occur correctly for custom finders defined on an EJB 1.1 Home interface that uses a backend data store that requires the special SQL locking clauses, the following Java virtual machine (JVM) and bean (module) properties are provided. These properties enable you to indicate which custom finders to enhance, assuming the backend data store supports the SQL clauses.

For z/OS, to view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* **> Control** (to define the property in the Control) or **Servant** (to define the property in the Servant) **> Java and process management > Process definition > Java virtual machine > Custom properties**.

com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent:

Used to indicate which enterprise beans should have custom finder SQL dynamic enhancement enabled at runtime.

This property takes effect at the server level. Any EJB 1.1 home interface-defined custom finder (prefix named *find*) that has *Update* as an access intent is a candidate for custom finder SQL dynamic enhancement based on its specified isolation level. If the backend data store requires special SQL semantics, they are applied. The particular SQL used varies according to the isolation level you choose for beans in the application, as well the backend data base being used. If set to **all**, custom finder SQL dynamic enhancement is enabled for all custom finders defined in any beans that are installed into the container. If set to **J2EENAME[:J2EENAME]**, where *J2EENAME* is a fully qualified package or bean name, custom finder SQL dynamic enhancement is enabled for only the custom finders defined in the beans that are installed into the container and represented by the bean names denoted.

Data type	String
Range	Valid values are **all** or **J2EENAME[:J2EENAME]**
Default	Enhancement behavior not active

Note: Some of your applications might use custom finders that have been manually coded and already contain the SQL locking clauses, or keywords *ORDER BY* and *DISTINCT* on the *SELECT* operation. In these instances, if the run time attempts SQL dynamic enhancement, the possibility exists of introducing malformed SQL statements to the underlying backend data store. If an application contains these custom finders, then you must be careful when specifying the value for the JVM property *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent*. A value of **all** causes custom finder SQL dynamic enhancement to occur for every custom finder method defined with an access intent of *Update* found in all beans that are installed in the application server, thus introducing malformed SQL for that subset of custom finders.

To prevent this from happening, ***do not*** set the server-wide setting to **all**. Instead, use the bean method level property, *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel* to indicate on a per bean basis only those custom finder methods that should have the custom finder SQL dynamic enhancement executed on them at run time.

com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel:

Used to indicate custom finder SQL dynamic enhancement be enabled at the method level on a particular bean.

When a bean is defined with this property set to a list of one or more custom finder methods, any custom finder (prefix named *find*) defined on the home interface that has a matching method name and parameter signature has SQL locking semantics applied at run time. This occurs only if the custom finder method has an access intent of *Update* specified and the backend data store supports the SQL clauses. The particular SQL used varies according to the isolation level chosen for the application as well as the backend data store being used.

Data type	String
Range	Valid value is a string of this form: **method1(parm1,parm2,..parmn):method2(parm1,parm2,..parmn):methodn(...)**

com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent:

Used by WebSphere Application Server for z/OS Version 4.x users to indicate that the SQL enhancement capability should **not be** applied to applications installed in the WebSphere Application Server for z/OS product.

The default behavior of the WebSphere Application Server for z/OS Version 4.x product is to perform the dynamic SQL enhancements. For those z/OS users choosing not to participate in dynamic SQL enhancement of custom finders in the Version 4.x product, this attribute is used to make this indication at both the bean and the server level.

At the bean level, a name/value pair consisting of this attribute name and a value of **true** disables the SQL enhancement of any custom finder defined on the given bean's home interface.

At the server level, an entry into the WebSphere Application Server for z/OS server property file with a value of **true** disables the SQL enhancement of all beans installed in the given server.

This custom finder enhancement attribute continues to be supported by the runtime at the bean level in the Version 5.x product. Its use as a server wide indicator has been deprecated by the fact that the default behavior of Version 5.x is to ***not*** dynamically enhance custom finder SQL.

Note: If your WebSphere Application Server for z/OS installation relies on the automatic dynamic enhancement of all custom finders in all applications installed, you should set the *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* indicator to **all**. If an application contains a bean that has the *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* indicator set into its *env-var* settings, that indicator continues to be used, provided the dynamic SQL enhancement features of the Version 5.x product are enabled as described above.

Data type	String
Range	Valid values are **true** and **false**

Some notes about precedence:

- The *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel* attribute overrides any server-wide or bean level attribute setting

- Any bean listed through a *J2EE Name* in the *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* indicator causes dynamic enhancement to occur for custom finders defined for that bean, even if the default behavior is in effect for the server in question.
- The *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* attribute disables a particular bean's use of this feature if the server-wide setting or bean setting is enabled and no method level settings are specified.

Accessing data using Java EE Connector Architecture connectors

To access data from a Java EE Connector Architecture (JCA) compliant application in WebSphere Application Server, you configure and use resource adapters and connection factories.

About this task

An application component uses a connection factory to access a connection instance, which the component then uses to connect to the underlying enterprise information system (EIS). Examples of connections include database connections, Java Message Service connections, and SAP R/3 connections.

As indicated in the Java EE Connector Architecture (JCA) Specification, each enterprise information system (EIS) needs a resource adapter and a connection factory. This connection factory is then accessed through the following programming model. If you use Rational Application Development (RAD) tools, most of the following deployment descriptors and code are generated for you. This example shows the manual method of accessing an EIS resource.

1. Declare a connection factory resource reference in your application component deployment descriptors, as described in this example:

```
<resource-ref>
   <description>description</description>
   <res-ref-name>eis/myConnection</res-ref-name>
   <res-type>javax.resource.cci.ConnectionFactory</res-type>
   <res-auth>Application</res-auth>
</resource-ref>
```

2. During the deployment process, configure each resource adapter and associated connection factory through the console. See the topics on installing a resource adapter and configuring a connection factory for more information.
3. Locate the corresponding connection factory for the EIS resource adapter using Java Naming and Directory Interface (JNDI) lookup in your application component, during run time.
4. Get the connection to the EIS from the connection factory.
5. Create an interaction from the connection object.
6. Create an *InteractionSpec* object. Set the function to execute in the InteractionSpec object.
7. Create a record instance for the input and output data used by function.
8. Execute the function through the Interaction object.
9. Process the record data from the function.
10. Close the connection.

Example

The following code segment shows how an application component might create an interaction and execute it on the EIS:

```
javax.resource.cci.ConnectionFactory connectionFactory = null;
javax.resource.cci.Connection connection = null;
javax.resource.cci.Interaction interaction = null;
javax.resource.cci.InteractionSpec interactionSpec = null;
javax.resource.cci.Record inRec = null;
javax.resource.cci.Record outRec = null;
```

```
try {
// Locate the application component and perform a JNDI lookup
   javax.naming.InitialContext ctx = new javax.naming.InitialContext();
   connectionFactory = (javax.resource.cci.ConnectionFactory)
ctx.lookup("java:comp/env/eis/myConnection");

// create a connection
   connection = connectionFactory.getConnection();

// Create Interaction and an InteractionSpec
   interaction = connection.createInteraction();
   interactionSpec = new InteractionSpec();
   interactionSpec.setFunctionName("GET");

// Create input record
   inRec = new javax.resource.cci.Record();

// Execute an interaction
   interaction.execute(interactionSpec, inRec, outRec);

// Process the output...

} catch (Exception e) {
   // Exception Handling
}
finally {
    if (interaction != null) {
         try {
              interaction.close();
         }
         catch (Exception e) {/* ignore the exception*/}
   }
   if (connection != null) {
        try {
             connection.close();
        }
        catch (Exception e) {/* ignore the exception */}
   }
}
```

JDBC application development tips

By using best practices to help maximize the efficiency of JDBC queries, you can potentially increase application performance.

Most of the following recommendations assume that you use DB2 on z/OS.

- Program according to the most current JDBC specifications.
- Use prepared statements to allow dynamic statement cache of DB2 on z/OS.
- Do not include literals in the prepared statements; use a parameter marker ″?″ to allow dynamic statement cache of DB2 on z/OS.
- Use the right getXxx method by each data type of DB2.
- Turn auto commit off when just read-only operations are performed.
- Use explicit connection context objects.
- When coding an iterator, you have a choice of named or positioned. Positioned iterators have the better performance potential.
- Close prepared statements before reusing the statement handle to prepare a different SQL statement within the same connection.
- As a bean developer, you have the choice of using JDBC or Structured Query language in Java (SQLJ) queries. JDBC makes use of dynamic SQL whereas SQLJ generally is static and uses pre-prepared plans. SQLJ requires an extra step to create and bind the plan whereas JDBC does not. SQLJ, as a general rule, is faster than JDBC.

- With JDBC and SQLJ, you are better off writing specific calls that retrieve just what you want rather than generic calls that retrieve the entire row. There is a high per-field cost.

JDBC application cursor holdability support

The cursor holdability feature can reduce the overhead of JDBC interaction with your relational database, thereby helping to increase application performance.

By activating cursor holdability, you keep a result set available across transaction boundaries for use by multiple JDBC calls. The holdability setting triggers a database cursor to keep newly updated rows active beyond the commit of the transaction that generated the new values, or result set. Hence the cursor makes the result set available for use by statements in a subsequent transaction.

Setting cursor holdability

Use one of the following techniques to set cursor holdability. For more details, see the JDBC 3.0 specification, available at the Sun Microsystems, Inc., Web site at http://java.sun.com.

- Specify the ResultSet.HOLD_CURSORS_OVER_COMMIT parameter when creating or preparing a statement using the createStatement, prepareStatement, or prepareCall methods.
- Invoke the setHoldability method on the Connection object. The cursor holdability value that you set with this method becomes the default. If you specify cursor holdability on the Statement object, that value overrides the value that you specified on the connection.

  You cannot specify cursor holdability on a shareable connection after that connection is referenced by a second handle. Invoking the holdability method at this point generates an exception. If you want to set cursor holdability on a shareable connection, invoke the method before the connection is enlisted. Otherwise a shareable connection retains the same holdability value that applied in the previous enlistment.
- Check your database documentation to see if the product supports cursor holdability as a data source property. DB2, for example, responds to the holdability trigger if you set it as a data source custom property.

The impact of connection and transaction behaviors on cursor holdability

Setting cursor holdability in WebSphere Application Server results in the following behavior for different transaction events:

- When a connection is closed, all statements and result sets are closed even if you have set cursor holdability.
- When a transaction is rolled back, all result sets are closed even if you have set cursor holdability.
- When a local transaction is committed, both shareable and unshareable connections can have an open result set across a transaction boundary.
- When a global transaction is committed, unshareable connections can have an open result set across a transaction boundary. For shareable connections, the statements and result sets are closed even if you have set cursor holdability; the holdability value does not impact shareable connections participating in global transactions.
- When a local transaction scope ends, either at the method level or the activity session level, all statements and result sets for shareable connections are closed. Statements and result sets for unshareable connections remain open until the close method is called on the connection.

Note: For a global transaction with an unshareable connection, the backend database has responsibility for supporting cursor holdability.

Data access bean types

For easy data access programming, WebSphere Application Server provides a special class library that implements many methods of the JDBC API for you. The library is essentially a set of *data access beans*.

There are several sets of classes referred to as data access beans. To make things clearer, you can refer to the classes by the name of the JAR file that contains them:

databeans.jar - This JAR file ships with WebSphere Application Server. This file contains classes that enable you to access the database using the JDBC API.

ivjdab.jar - This JAR file ships with Visual Age for Java. This file contains all of the classes in the databeans.jar file and classes that support easy use of the data access beans from the Visual Age for Java Visual Composition Editor.

dbbeans.jar - This JAR file ships with Rational Application Developer. This file contains a set of data access beans to more closely conform to the JDBC 2.0 *RowSet* standard.

For the current product, data access beans remain unchanged from WebSphere Application Server Version 4.0. The *com.ibm.db* package is provided to support existing applications that use data access beans.

IBM strongly suggests that any new applications using data access beans be developed using the *com.ibm.db.beans* package that is provided with Rational Application Developer.

If you want to continue using applications that use the com.ibm.db package, see the WebSphere Application Server Version 4.0 documentation concerning data access beans.

If you want to create new applications that use the com.ibm.db.beans package, see the Rational Application Developer documentation concerning data access beans. An example is shown here: Example: Using data access beans

Example: Using data access beans

```
package example;
import com.ibm.db.beans.*;
import java.sql.SQLException;

public class DBSelectExample {

 public static void main(String[] args) {

  DBSelect select = null;

  select = new DBSelect();
  try {

   // Set database connection information
   select.setDriverName("COM.ibm.db2.jdbc.app.DB2Driver");
   select.setUrl("jdbc:db2:SAMPLE");
   select.setUsername("userid");
   select.setPassword("password");

   // Specify the SQL statement to be executed
   select.setCommand("SELECT * FROM DEPARTMENT");

   // Execute the statement and retrieve the result set into the cache
   select.execute();

   // If result set is not empty
   if (select.onRow()) {
    do {
     // display first column of result set
     System.out.println(select.getColumnAsString(1));
     System.out.println(select.getColumnAsString(2));
    } while (select.next());
   }
```

```
   // Release the JDBC resources and close the connection
   select.close();

  } catch (SQLException ex) {
   ex.printStackTrace();
  }
 }
}
```

Accessing data from application clients

To access a database directly from a Java Platform, Enterprise Edition (Java EE) application client, you retrieve a *javax.sql.DataSource* object from a resource reference configured in the client deployment descriptor. This resource reference is configured as part of the deployment descriptor for the client application, and provides a reference to a pre-configured data source object.

About this task

Note that data access from an application client uses the JDBC driver connection functionality directly from the client side. It does not take advantage of the additional pooling support available in the application server run time. For this reason, your client application should utilize an enterprise bean running on the server side to perform data access. This enterprise bean can then take advantage of the connection reuse and additional added functionality provided by the product run time.

1. Import the appropriate JDBC API and naming packages:

   ```
   import java.sql.*;
   import javax.sql.*;
   import javax.naming.*;
   ```

2. Create the initial naming context:

   ```
   InitialContext ctx = new InitialContext();
   ```

3. Use the *InitialContext* object to look up a data source object from a resource reference.

   ```
   javax.sql.DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/myDS");
   //where jdbc/myDS is the name of the resource reference
   ```

4. Get a *java.sql.Connection* from the data source.
   - If no user ID and password are required for the connection, or if you are going to use the *defaultUser* and *defaultPassword* that are specified when the data source is created in the Application Client Resource Configuration tool (ACRCT) in a future step, use this approach:

     ```
     java.sql.Connection conn = ds.getConnection();
     ```

   - Otherwise, you should make the connection with a specific user ID and password:

     ```
     java.sql.Connection conn = ds.getConnection("user", "password");
     //where user and password are the user id and password for the connection
     ```

5. Run a database query using the *java.sql.Statement*, *java.sql.PreparedStatement*, or *java.sql.CallableStatement* interfaces as appropriate.

   ```
   Statement stmt = conn.createStatement();
   String query   = "Select FirstNme from " + owner.toUpperCase() + ".Employee where LASTNAME = '" + searchName + "'";
   ResultSet rs   = stmt.executeQuery(query);
   while (rs.next()) {    firstNameList.addElement(rs.getString(1));
   }
   ```

6. Close the database objects used in the previous step, including any *ResultSet*, *Statement*, *PreparedStatement*, or *CallableStatement* objects.
7. Close the connection. Ideally, you should close the connection in a *finally* block of the *try...catch* statement wrapped around the database operation. This action ensures that the connection gets closed, even in the case of an exception.

   ```
   conn.close();
   ```

Data access with Service DataObjects, API versions 1.0 and 2.01

The Service DataObjects (SDO) framework is a data-centric, disconnected, XML-integrated, data access mechanism that provides a source-independent result set.

- SDO is data-centric because it eliminates the need for client applications to work with special formats of data, such as the object representations of the Enterprise JavaBean (EJB) API. Instead, clients work with easily traversable graphs of DataObjects.
- SDO is disconnected because the retrieved result is independent of any back end data store connections or transactions.
- SDO is XML-integrated in that it provides services to easily convert retrieved data to and from XML format.

Put simply, SDO is a framework for data application development, which includes an architecture and API. SDO does the following:

- Simplifies the Java Platform, Enterprise Edition (J2EE) data programming model.
- Abstracts data in a service oriented architecture (SOA).
- Unifies data application development.
- Supports and integrates XML.
- Incorporates J2EE patterns and best practices.

The Service DataObjects framework provides a unified framework for data application development. With SDO, you do not need to be familiar with a technology-specific API in order to access and utilize data. You need to know only one API, the SDO API, which lets you work with data from multiple data sources, including relational databases, entity EJB components, XML pages, Web services, the Java Connector Architecture, JavaServer Pages, and more.

Unlike some of the other data integration models, SDO does not stop at data abstraction. The SDO framework also incorporates a good number of J2EE patterns and best practices, making it easy to incorporate proven architecture and designs into your applications. For example, the majority of Web applications today are not (and cannot) be connected to backend systems 100 percent of the time; so SDO supports a disconnected programming model. Likewise, many applications tend to be remarkably complex, comprising many layers of concern. How will data be stored? Sent? Presented to end users in a GUI framework? The SDO programming model prescribes patterns of usage that allow clean separation of each of these concerns.

SDO components

An architectural overview of SDO describes each of the components that make up the framework and explains how they work together. The first three components listed are ″conceptual″ features of SDO: They do not have a corresponding interface in the API.

SDO clients

SDO clients use the SDO framework to work with data. Instead of using technology-specific APIs and frameworks, they use the SDO programming model and API. SDO clients work on SDO DataObjects and do not need to know how the data they are working with is persisted or serialized.

Data mediator services

Data mediators services (DMS) are responsible for creating a DataGraph from data sources, and updating data sources based on changes made to a DataGraph. (A DataGraph is an envelope object that contains service data objects.)

The DMS provides the mechanism to move data between a client and a data source. It is created with back end specific metadata. The metadata defines the structure of the DataGraph that is produced by the DMS as well as the query to be used against the back end. When the DMS is requested to produce a DataGraph, it queries its targeted back end and transforms the native result set into the DataGraph format. Once the DataGraph is returned, the DMS no longer has any

reference to it, making it stateless with respect to the DataGraph. When the DMS is requested to flush modifications of an existing DataGraph to the back end, it extracts the changes made from the original state of the DataGraph and flushes those changes to the back end. A DMS typically employs some form of optimistic concurrency control strategy to detect update collisions.

WebSphere Application Server provides functionality for two separate Data Mediator Services. If you simply need to retrieve data from a relational data source and return a DataGraph, using the Java Database Data Mediator Service is a good choice. However, if you have business logic, then you probably want an object oriented (OO) rendering of the data into entity beans. One could consider SDOs as an object rendering of data similar to entity beans. But entity beans have better Object-Relational (OR) mapping tools, and the EJB container and persistence manager for entity beans offer more sophisticated caching policies. Your best choice then is the EJB Data Mediator Service. The EJB mediator can work with these caches. Also, the entity bean programming model is a single level store model. You can navigate from entity to entity and the container and persistence manager either prefetches or lazily fetches in data as needed. On update, the programmer commits the transaction and the container and persistence manager do the work of tracking updated beans and writing them back to the data store and in memory cache.

Data sources

Data sources are not restricted to backend data sources (for example, persistence databases). A data source contains data in its own format. In the case of the SDO 1.0 API, only the DMS accesses data sources; SDO applications do not. The applications only work with SDO 1.0 DataGraphs.

Each of the following components corresponds to a Java interface in the SDO programming model.

DataObjects

As the fundamental components of SDO, DataObjects provide a common view of structured data for SDO clients. DataObjects can hold multiple different attributes of any serializable type (such as string or integer); more complex DataObjects can also contain simpler DataObjects. DataObjects hold all of their data in *properties*.

SDO version 1.0 DataObjects are always linked together and contained in DataGraphs. The version 1.0 DataObject interface provides simple creation and deletion methods (createDataObject() with various signatures and delete()), and reflective methods to get their types (instance class, name, properties, and namespaces). The interface also supports static object types that you create from external code generators. See the article ″Dynamic and static object types for the JDBC DMS″ for more information.

DataGraphs

A DataGraph is a structured result returned in response to a service request. The DMS transforms the native backend query results into the DataGraph, which is independent of the originating backend data store. This makes the DataGraph easily transferable between different data sources. The DataGraph is composed of interconnected nodes, each of which is an SDO DataObject. It is independent of connections and transactions of the originating data source. The DataGraph keeps track of the changes made to it from its original source. This change history can be used by the DMS to reflect changes back to the original data source. DataGraphs can easily be converted to and from XML documents enabling them to be transferred between layers within a multi-tiered system architecture. A DataGraph can be accessed in either breadth-first or depth-first manner, and it provides a disconnected data cache that can be serialized for Web services

The DataGraph returned by the mediator can contain either dynamic or generated static DataObjects. Use of generated classes gives type safe interfaces for easier programming and better run time performance. The EMF generated classes must be consistent in name and type with the schema that would be created for dynamic DataObjects except that additional attributes and references can be defined. Only those attributes and references specified in the query are filled in with data. Remaining attributes and references are not set.

Change summary

SDO 1.0 change summaries are contained by DataGraphs and are used to represent the changes that have been made to a DataGraph returned by the DMS. They are initially empty (when the DataGraph is returned to a client) and populated as the DataGraph is modified. Change summaries are used by the DMS at backend update time to apply the changes back to the data source. They enable the DMS to efficiently and incrementally update data sources by providing lists of the changed properties (along with their old values) and the created and deleted DataObjects in the DataGraph. Information is added to the change summary of a DataGraph only when the change summary's logging is activated. Change summaries provide methods for DMS to turn logging on and off.

Note: The SDO 1.0 change summary is not a client API; it is used only by the DMS.

Properties, types, and sequences

DataObjects hold their contents in a series of properties. Each property has a type, which is either an attribute type such as a primitive (for example, int) or a commonly used data type (for example, Date) or, if a reference, the type of another DataObject. Each DataObject provides read and write access methods (getters and setters) for its properties. Several overloaded versions of these accessors are provided, allowing the properties to be accessed by passing the property name (String), number (int), or property metaobject itself. The String accessor also supports an XPath-like syntax for accessing properties. For example you can call get(″department[number=123]″) on a company DataObject to get its first department whose number is 123. Sequences are more advanced. They allow order to be preserved across heterogeneous lists of property-value pairs.

For more introductory information

For a good introduction to SDO 1.0 that also includes a small sample application, refer to the IBM developerWorks paper ″Introduction to Service DataObjects.″

Note: To fully understand the EJB data mediator service you need a good understanding of the EJB programming model. For more information refer to the articles ″Task overview: Using enterprise beans in applications″ and ″Service Data Objects: Resources for learning.″

Java DataBase Connectivity Mediator Service

The Java Database Connectivity (JDBC) Data Mediator Service (DMS) is the Service Data Objects (SDO) component that connects to any database that supports JDBC connectivity. It provides the mechanism to move data between a DataGraph and a database.

A regular JDBC call returns a result set in a tabular format. This format does not directly correspond to the object-oriented data model of Java, and can complicate navigation and update operations. When a client sends a query for data through the JDBC DMS, the JDBC result set of tabular data is transformed into a DataGraph composed of related DataObjects. This enables clients to navigate through a graph to locate relevant data rather than iterating through rows of a JDBC result set. After altering the DataGraph, all of the changes can be committed together and propagated back to the database by the JDBC DMS. Between the processes of being populated and being committed, the DataGraph is disconnected from the database, and there are no locks held on the data accessed. Being disconnected allows multiple changes to be made to the graph without making additional round trips to the database, improving performance.

The JDBC DMS is created with backend-specific metadata. The metadata defines the structure of the DataGraph that is produced by the DMS, as well as the query to be used against the back end.

Metadata for the Data Mediator Service:

A Data Mediator Service (DMS) is the Service Data Object (SDO) component that connects to the back end database. It is created with back end specific metadata. The metadata defines the structure of the DataGraph that is produced by the DMS as well as the query to be used against the back end.

Metadata is composed of the following components:

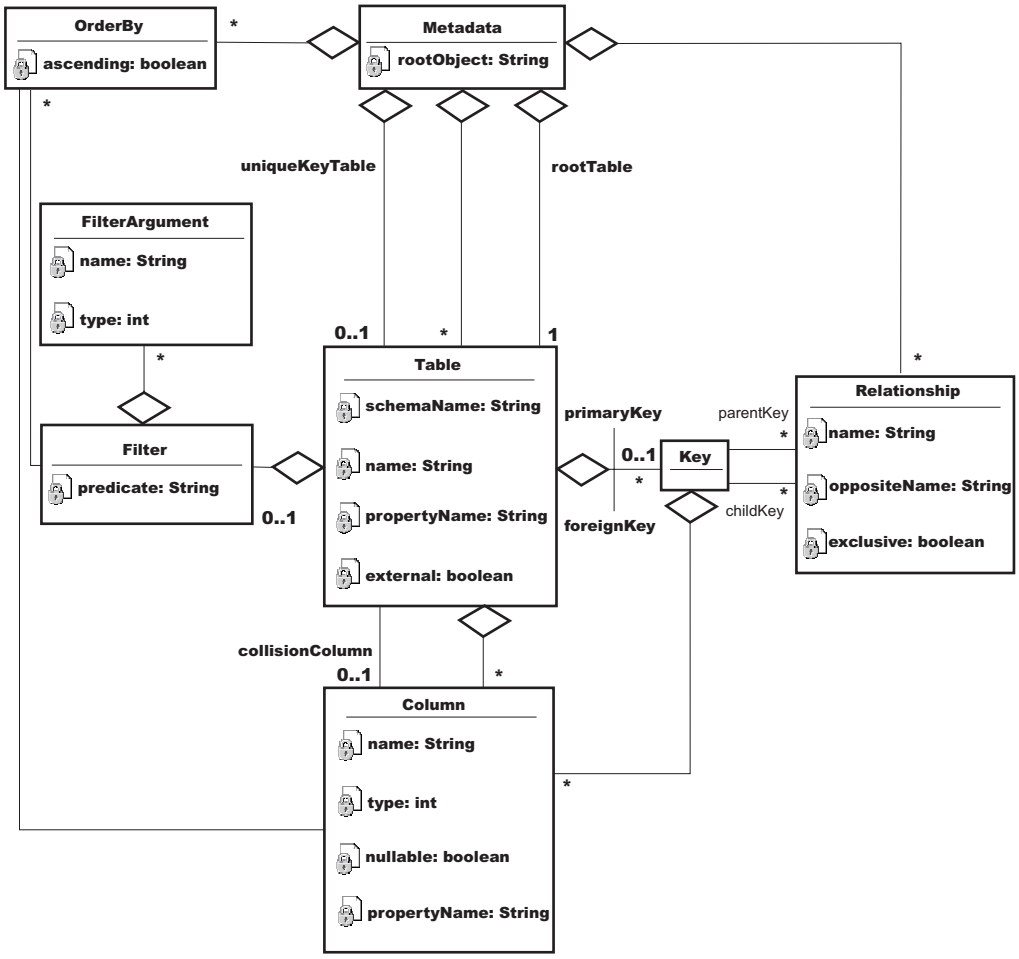


Table This represents a table within the target database and is composed of the following items:

Name This is the database table name. A table might also have a property name that can be used to specify the name of the DataObject that corresponds to this table. By default, the property name is the same as the table name.

Columns

The subset of database table columns to return from the database. A column has a type that corresponds to a JDBC type and it can prohibit null entries. A column has a name that corresponds to the name in the database and an optional property name that identifies the column name in the DataObject. By default, the property name is the same as the column name in the database.

Primary Key

The column (or columns) used to uniquely identify a row within the table.

Note: Keys may be composed of multiple columns. The following example illustrates creation of a compound primary key :

```
Key pk = MetadataFactory.eINSTANCE.createKey();
pk.getColumns().add(xColumn);
pk.getColumns().add(yColumn);
coordinateTable.setPrimaryKey(pk);
```

If a table is related to this one and is the child table, it uses the same method to create the foreign key to point to this coordinate table.

Foreign Key
The column (or columns) used to relate the table to another table in the metadata. There is an assumed positional mapping between compound primary keys and foreign keys. For example, if a parent table has a primary key such as (x,y) with respective types (integer, string), then it is expected that any pointing foreign key is also (x', y') with respective types (integer, string).

Note: Keys may be composed of multiple columns. The following example illustrates the creation of a compound foreign key :

```
Key fk = MetadataFactory.eINSTANCE.createKey();
fk.getColumns().add(xColumn);
fk.getColumns().add(yColumn);
coordinateTable.getForeignKeys().add(fk);
```

If a table is related to this one and is the child table, it uses the same method to create the foreign key to point to this coordinate table.

Filter A structured query language (SQL) WHERE clause predicate that can be given with or without parameters to fill in later. This is added to the DataGraph SELECT statement WHERE clause. It is not parsed or interpreted in any way; it is used as is. If given with parameters to fill in later, these parameters become arguments passed into the JDBC DMS when getting the DataGraph. Filters are used with generated queries only. If a supplied query is given, the metadata filters are ignored.

Relationship
Relates two tables through the primary key of the parent table and the foreign key of the child table. Relationships are composed of the following items:

Name This is the name given to the relationship, usually associated with how the two tables are related. If *Customers* is the parent table and *Orders* is the child table, then the default name of the relationship is Customers_Orders.

Opposite Name
This is the name used to navigate from the child DataObject to the parent DataObject.

Parent Key
The primary key of the parent table.

Child Key
The foreign key of the child table that points to the parent key.

Exclusive
By default, a Relationship causes the generated query to use an inner join operation on the two tables involved in the relationship. This means that it only returns the parent entries that have children, that is, child entries pointing to them. If the value of the Exclusive attribute is set to false, the query uses a left outer join operation instead and returns all parent entries, even those without children.

Ordering
Columns used for ordering the tables. Can be either ascending or descending. When specified, this causes generated queries to contain an ORDER BY clause.

Dynamic and static object types for the JDBC DMS:

DataObjects of the Service Data Object (SDO) 1.0 Specification can use static types as well as dynamic types. If you know that a particular dataGraph schema meets all of your application query requirements, you can generate static SDO code for potential runtime benefits.

With dynamic types, the information that defines the shape of a DataGraph is constructed at runtime. The DataGraph schema is created by the JDBC data mediator service (DMS) from the metadata provided upon creation. The JDBC DMS only requires the metadata and a connection to a data source to produce the DataGraph with dynamic typing. This is the default method for creating the JDBC DMS.

If you know the shape of the DataGraph at development time, you can use a code generator to create strongly typed interfaces (static data API code) that simplify DataGraph navigation, provide better compile-time checking for errors, and improve performance. For more information about the metamodels from which you can generate static SDO code, consult the introduction of the SDO 1.0 Specification. The introduction contains a list of the specification scope requirements, where you can find a brief discussion on support for static data API. Note that the dynamic API is still available when you use strongly typed DataObjects.

With the code generator you create classes for each DataObject type in the DataGraph. Each class contains *getter()* and *setter()* methods for each property in the DataObject. This enables a client to call type-safe methods rather than passing in the name of a property. For example, instead of calling the property *DataObject.get("CUSTFIRSTNAME")*, the generated types can contain a DataObject.getCustFirstName() method. If you are accessing a related DataObject, an accessor returns a strongly-typed DataObject rather than a regular DataObject. For example, *DataObject.get("Customers_Orders")* returns a DataObject, but *DataObject.getOrders()* returns an object of type Order.

To use static typing with the JDBC DMS, the metadata, a connection to the data source, and the DataGraph schema need to be provided to the JDBCMediatorFactory class *create* methods. In this case, the JDBC DMS metadata does not determine the shape of the DataGraph, but does give the DMS information about the backend data source and the way it maps to a DataGraph.

When using strongly typed DataObjects, it is important to make sure that the query matches the DataGraph schema. The query is not required to fill all of the data objects and properties in the schema, but a query cannot return data objects or properties that are not defined in the DataGraph schema. For example, a DataGraph schema might define *Customer* and *Order* DataObjects, but a query might only return Customer objects. Also, the Customer object might define properties for *ID*, *Name*, and *Address*, but the query might not return an address. In this case, the value of the address property is null, and the value is not updated in the database when the *applyChanges()* method is called. In this example, the query could not return a *Phone* property because it has not been defined as a property on the Customer object. When a query attempts this, the DMS returns an invalid metadata exception.

JDBC mediator supplied query:

An SDO client can supply the JDBC Data Mediator Service (DMS) with a SELECT statement to replace the statement that is generated from the DMS metadata.

When the SDO client instantiates a DMS, the DMS uses the defining metadata to generate a basic SELECT statement. Substituting that query gives you the ability to specify parameter markers; therefore you have more control over the client data that populates a dataGraph. Use a standard SQL SELECT string for a client-supplied query.

With both supplied queries and generated queries, UPDATE, INSERT, and DELETE statements are automatically generated for each DataObject. They are applied when the mediator commits the changes made to the DataGraph back to the database.

Parameter DataObjects for supplied queries

Clients can use a parameter DataObject to supply arguments to an SQL SELECT query. A parameter DataObject is a DataObject, but is not part of any DataGraph. It is constructed by the JDBC DMS when

requested by the client. The ParameterDataObject for supplied queries is created based on the query given to the mediator. Every parameter in the query is given a name like *arg0, arg1, ..., argX*.

Because a parameter DataObject is a DataObject, you can set its properties using either the property name or an index value. The properties can be referenced by their argX name, or by the number associated with that parameter, 0, 1, ... , X. For example, your query is “SELECT CUSTFIRSTNAME WHERE CUSTSTATE = ? AND CUSTZIP = ?”. This supplied query contains two parameters. The first parameter corresponds with CUSTSTATE and can be set using the string “arg0” or the index 0. The second parameter corresponds with CUSTZIP and can be set using the string “arg1” or the index 1. Here is sample code of how they are set. This code assumes that you have already set up the metadata and mediator with the metadata and the aforementioned supplied query. Using the index value method, you code:

```
DataObject parameters = mediator.getParameterDataObject();
parameter.setString(0, "NY");
parameter.setInt(1, 12345);
DataObject graph = mediator.getGraph(parameters);
```

Using the property name method, you code:

```
DataObject parameters = mediator.getParameterDataObject();
parameters.setString("arg0", "NY");
parameters.setInt("arg1", 12345);
DataObject graph = mediator.getGraph(parameters);
```

The results are the same for both cases.

Limitations

The JDBC DMS generated SQL SELECT query is not fully supported on Oracle or Informix. This is because the mediator takes advantage of the ResultSetMetaData interface in JDBC 2.0 and requires it to be fully implemented. Oracle, Informix, DB2/390, and older supported versions of Sybase do not implement the ResultSetMetaData interface completely. The supplied select approach can still be used with these databases with one limitation: **column names in the Metadata must be unique across all tables.** An InvalidMetadataException occurs if the select statement returns a column with a name that appears multiple times in the metadata. For instance, if the Customer and the Order tables both contain a column named “ID”, this would be invalid and cause problems. The way to fix this is to change the name of at least one of the matching columns in the database to better distinguish the two columns from each other. For the Customer table, the column name could be changed to “CUSTID,” as it is in the examples. The Order column name could be changed to “ORDERID”. If you change the Customer column name, you do not have to change the Order column name, but for consistency it may be a good idea.

JDBC mediator generated query:

If you do not provide a structured query language (SQL) SELECT statement, then the data mediator service (DMS) generates one using the metadata provided at instance creation.

The internal query engine uses information in the metadata about tables, columns, relationships, filters, and order bys to construct a query. As with the supplied queries, UPDATE, DELETE, and INSERT statements are automatically generated for each DataObject to be applied when the mediator commits the changes made to the DataGraph back to the database.

Filters

Filters define an SQL WHERE clause that might contain parameter markers. These are added to the DataGraph SELECT statement WHERE clause. Filters are used as is; they are not parsed or interpreted in any way so there is no error checking. If you use the wrong name, predicate, or function, it is not detected and the generated query is not valid. If a Filter WHERE clause contains parameter markers, then the

corresponding parameter name and type are defined using Filter arguments. Parameter DataObjects fill in these parameters before the graph is retrieved. An example of the Filters and Parameter DataObjects for generated queries follows.

Note: Because of the tree-like nature of the DataGraph, any table at a branch appears in more than one subquery in the final union with the root table appearing in all paths. This means that it is not possible to filter on a table that appears in more than one path independent of all other paths. All filters defined on a particular table are joined by a boolean AND, and used everywhere that table appears.

Parameter DataObjects for generated queries

Clients use a Parameter DataObject to supply arguments that are applied to the filters provided in the DMS metadata. A Parameter DataObject is a DataObject, but is not part of any DataGraph. It is constructed by the JDBC DMS when requested by the client. The Parameter DataObject for generated queries is created based on the mediator's metadata. Every argument of every filter of every table is put into the Parameter DataObject. Unlike the supplied query Parameter DataObject, the parameters have the name assigned to them by the Filter arguments. The Parameter DataObject uses this name to map to the parameter to be filled in. The following sample code illustrates how a filter is created for a table in the mediator metadata. It also demonstrates the use of a Parameter DataObject to pass filter parameter values to a mediator instance. The sample assumes that the Customer table has already been defined:

```
// The factory is a MetadataFactory object
Filter filter = factory.createFilter();
filter.setPredicate("CUSTSTATE = ? AND CUSTZIP = ?");

FilterArgument arg0 = factory.createFilterArgument();
arg0.setName("customerState");
arg0.setType(Column.String);
queryInfo.getFilterArguments().add(arg0);

FilterArgument arg1 = factory.createFilterArgument();
arg1.setName("customerZipCode");
arg1.setType(Column.Integer);
queryInfo.getFilterArguments().add(arg1);

// custTable is the Customer Table object
custTable.setFilter(filter);


..... // setting up mediator


DataObject parameters = mediator.getParameterDataObject();

// Notice the first parameter is the name given to the
// argument by the FilterArgument.
parameter.setString("customerState", "NY");
parameter.setInt("customerZipCode", 12345);
DataObject graph = mediator.getGraph(parameters);
```

Order-by

Ordering of query results is specified using OrderBy objects that identify a column from a table to sort the results. This ordering can be either ascending or descending. The OrderBy objects are part of the metadata and are automatically applied to generated queries. An example of this for a customer table results to be sorted by first names is as follows:

```
// This example assumes that the custTable, a table in
// the metadata, and factory, the MetaDataFactory
// object, have already been created.
Column firstName = ((TableImpl)custTable).getColumn("CUSTFIRSTNAME");
```

```
OrderBy orderBy = factory.createOrderBy();
orderBy.setColumn(firstName);
orderBy.setAscending(true);
metadata.getOrderBys().add(orderBy);
```

Note: Even though Order-bys are defined on each table in the metadata, the RDBMS model requires them to be applied to the final query. This has many implications. For example, you cannot order a table and then use that in a join to another table and propagate the ordering in the first table. Because a result set is a union of all the tables in the DataGraph, the nature of the single result set requires that it be padded with nulls, which can affect the order-bys, particularly in the non-root tables. This can give unexpected results.

External Tables

An external table is a table defined in the metadata that is not needed in the DataGraph returned by the JDBC DMS. This might be appropriate when you want to filter the result set based on data from a table but that table's data is not needed in the result set. An example of this with the Customers and Orders relationship would be to filter the results to return all customers who ordered items with an order date of the first of the year. In this case, you do not want any order information returned, but you do need to filter on the order information. Making the Orders table external excludes the orders information from the DataGraph and therefore reduces the DataGraph's size, improving efficiency. To designate a table as external, you call the *setExternal(true)* method from a table object in the JDBC DMS metadata. If the client tries to access an external table from the DataGraph, an illegal argument exception occurs.

Note: Many RDBMSs require that an orderby column appear in the final result set; the columns from an external table cannot in general be used to order a result set. Order-bys are actually applied to the result set (the word ″set″ is key here), and not to intermediate query results.

General limitations of generated queries

In understanding the limitations of the query generation feature in the JDBC DMS, there are two things to keep in mind. The first is that the DataGraph imposes a model that is a directed, connected graph with no cycles (that is, a model that is a tree) on a relational model that is a non-directed, potentially disconnected graph with cycles. *Directed* means that the developer chooses the orientation of the graph by picking a root table. *Connected* means that all tables that are a member of the DataGraph are reachable from the root. Any tables that are not reachable from the root cannot be included in the DataGraph. In order for a table to be reachable from the root, there must be at least one foreign key relationship defined between each pair of tables in the DataGraph. *No cycles* means that there is only one foreign key relationship between a pair of tables in the DataGraph. The tree nature of the DataGraph determines how the queries are built, and what data is returned from a query.

The second item to keep in mind is the following high level description of how query generation produces read queries for a DataGraph:

1. The JDBC DMS creates a single result set (that is, a DataGraph) whether the DataGraph is composed from a single table or from multiple tables.
2. Each path through the foreign key relationships in DMS Metadata from root to leaves represents a separate path. The data for that path is retrieved by using joins across the foreign keys defined between the tables in the path. The joins are by default inner joins.
3. All the paths in a DataGraph are unioned together in order to create a single result set by the query that is generated by the mediator, and are thus treated independently of one another.
4. Any user-defined filtering is done first on the tables. Then the result is joined to the rest of the path.
5. Relational databases generally require order-bys to be applied to the entire final result set and not on intermediate results.

JDBC mediator performance considerations and limitations:

Use these tips to help you determine if a JDBC Data Mediator Service suits the requirements of your application serving environment.

Miscellaneous database limitations
- Sybase before Version 12.5.1 does not support in-line queries in the “from” clause, and therefore does not support multiple table DataGraphs with filters. To use the Service Data Object in WebSphere Application Server use Sybase Version 12.5.1.
- The Informix Dynamic Server does not support sub-selects, which are needed for multiple table graphs. Use Informix Extended Parallel Server.
- Oracle 8i does not support the ANSI join syntax. The mediator in multiple table cases requires Oracle 9i or 10g.

General performance recommendations
- Evaluate if your target projects are well suited to these technologies. In general, projects that are read-intensive and require disconnected data are good candidates.
- Limit the number of tables in the metadata. One or two is best because relationships, with respect to filters, become ambiguous when graphs have many branches.
- Work with small data sets as often as possible to avoid consuming excessive amounts of memory within your applications. You can limit the amount of data returned to the SDO by specifying filters in the metadata objects or by using paging.
- For Web applications, if the DataGraph is not too large and is to be reused later, store it in the user session.

JDBC mediator transactions:

You can specify that the JDBC mediator either act as transaction manager, or refrain from such activities in the case of external transaction management (performed by the SDO client).

Mediator managed transactions

A JDBC connection is wrapped in a connection wrapper and passed to the Data Mediator Service (DMS) during the instance creation. The ConnectionWrapper object contains the connection that is used by the JDBC DMS and indicates whether the mediator manages the current transaction. When the JDBC DMS manages the transaction, it performs commit and rollback operations as required. However, the DMS does not perform any transaction management activities if the wrapped connection is currently engaged in another transaction.

Using the createConnectionWrapper method for active transaction management is the general practice.

Non-mediator managed transactions

When a *passive* connection wrapper is passed to the DMS, the DMS takes no managerial action; a passive wrapper is generally intended for an existing transaction that is under external management. Commit or rollback operations are not performed by the connection wrapper in this case.

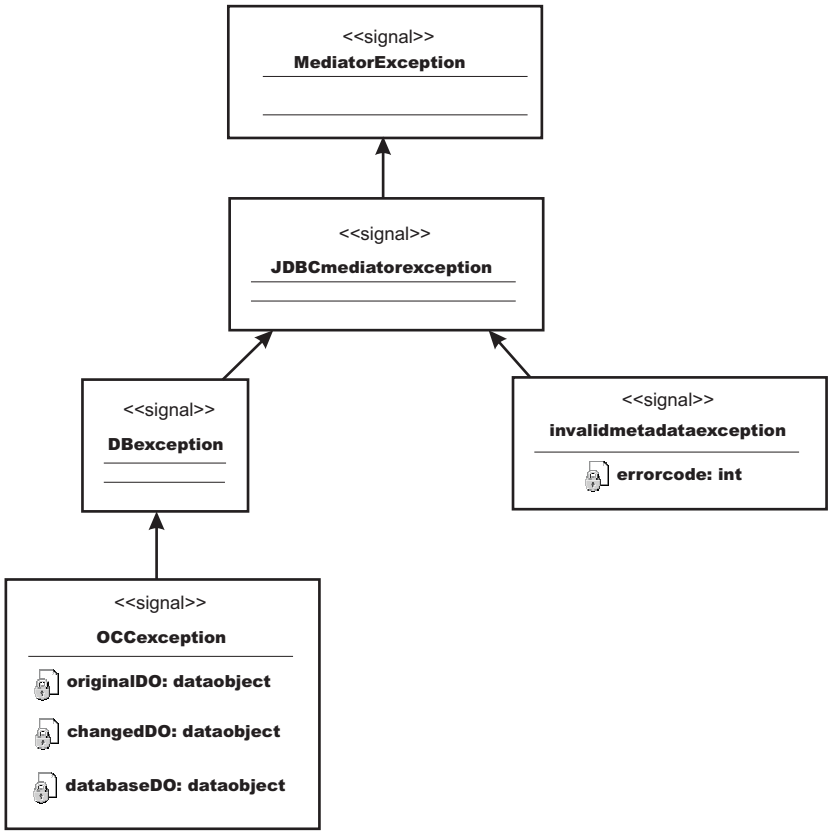
Use the createPassiveConnectionWrapper method.

Protection against referential integrity (RI) violations

The JDBC Data Mediator Service safeguards data transactions from incurring RI violations and other database logic violations. When the JDBC DMS applies the updates of a data graph to a back end, it automatically orders the change operations so that they do not violate database RI policy. Similarly, the DMS filters counter operations (such as INSERT and DELETE) so that opposing client requests can perform updates in a logical order. The client deletes one object, and then creates an entirely separate object with the same primary key. The DMS transforms these two operations into an update operation that modifies the existing database object.

JDBC mediator exceptions:

JDBC mediator exceptions either surface errors reported by the database, or indicate use of non-valid metadata in the attempt to instantiate the DMS.



The `Mediator` exception is the root exception of all the data mediator services, and the `JDBCMediator` exception is the root exception for the JDBC DMS in particular.

The `DB` exception occurs when an error is reported by the database. This can occur several ways:
- when the connection being used has the AutoCommit property set to *true*, but the JDBC DMS is controlling the transaction and needs it to be set to false
- when an unsupported database is trying to be used
- when other backend database errors occur during commit or rollback.

An optimistic concurrency control (OCC) exception occurs when the applyChanges() operation results in an data collision. When this occurs, the exception contains the original row values, current row values, and the attempted row values. These values are used to help recover from the error.

An `InvalidMetadata` exception occurs for invalid metadata supplied to the JDBC DMS upon creation. This can happen when a query requires tables or columns that are not defined in the metadata, or when there are identical column names for different tables for the Oracle, Informix, and older supported versions of Sybase databases.

Example: Forcing OCC data collisions and JDBC mediator exceptions:

The following example forces a collision to demonstrate detection and shows the exception that occurs as a result.

```
// This example assumes that a mediator has already
// been created and the first name in the list is Sam.
// It also assumes that the Customer table has an OCC
// column and the metadata has set this column to be
// the collision column.

DataObject graph1 = mediator.getGraph();
DataObject graph2 = mediator.getGraph();

DataObject customer1 = (DataObject)graph1.getList("CUSTOMER").get(0);
customer1.set("CUSTFIRSTNAME", "Bubba");

DataObject customer2 = (DataObject)graph2.getList("CUSTOMER").get(0);
customer2.set("BOWLERFIRSTNAME", "Slim");

mediator.applyChanges(graph2);

try
{
 mediator.applyChanges(graph1);
}
catch (OCCException e)
{
// Since graph1 was obtained before graph2 and
// graph2 has already been submitted, trying to
// apply the same changes to graph1 causes
// this OCC Exception.

assertEquals("Sam", e.getOriginalDO(). getString("CUSTFIRSTNAME"));
assertEquals("Bubba", e.getChangedDO(). getString("CUSTFIRSTNAME"));
assertEquals("Slim", e.getDatabaseDO(). getString("CUSTFIRSTNAME"));
}
```

Defining optimistic concurrency control for the JDBC Mediator:

Implement an optimistic concurrency control (OCC) strategy for the JDBC DMS to diagnose transaction problems that are caused by update collisions.

About this task

An *update collision* occurs when client data that populates a data graph is changed in the database before the data graph can submit the modifications of the client. If you configure the JDBC DMS for OCC, the DMS issues an OCC-specific exception when such a data collision happens. The OCC exception contains collision details such as the original row values, current row values, and the attempted row values. The client application uses these values to determine how to recover from the collision. For example, the application can reread the data and restart the transaction.

Be aware, however, that when one exception occurs, there is no way of knowing whether more exceptions exist deeper in the data graph schema and therefore are not displayed.

To activate OCC for the data mediator service, you must incorporate OCC columns into your database tables.

Add an *OCC Integer* column to a given table, and specify that this column is to be used for OCC in the metadata. The defined OCC collision column is reserved for the exclusive use of the mediator. If there is no OCC column defined for a table, the DMS does not monitor and notify you of update collisions. The following generic code segments create this setup.

1. Create the OCC column

   ```
   Column collisionColumn = table.addIntegerColumn("OCC_COUNT");
   ```

2. Ensure that it does not allow null values

```
collisionColumn.setNullable(false);
```

3. Designate the column as the table collision column

```
table.setCollisionColumn(collisionColumn);
```

For a fully-fledged code example that forces a collision to demonstrate the OCC exception, see the topic “Example: Forcing OCC data collisions and JDBC mediator exceptions” on page 1012.

JDBC mediator integration with presentation layer:

The JDBC Data Mediator Service (DMS) can be used in conjunction with Web application presentation layer technologies such as JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces (JSF).

This discussion assumes a general understanding of both of the JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces (JSF) technologies. In particular for JSF, the UIData component and the general file structure of a JSF dynamic Web application should be known. For a brief overview of both JSF and JSTL refer to the links at “Service Data Objects: Resources for learning” on page 1031.

The JDBC DMS and JSTL work well together because the JSTL access code is equivalent to the code necessary to access attributes and lists inside of a DataObject. For example, in relation to a root Customer DataObject, the JSTL expression:

```
${rootDO.CUSTOMER[index].CUSTNAME}
```

is equivalent to the Java code for a DataObject of:

```
rootDO.getList("CUSTOMER").get(index).get("CUSTNAME")
```

The reason for this is the dot notation in the JSTL expression language correlates to a *getter()* method in Java code, and the bracket notation allows you to access elements inside a list.

The JDBC DMS and JSF fit well together because the DataGraph produced by the JDBC DMS is able to populate a JSF UIData component without having to be transformed. The UIData component uses a *dataTable* tag that takes a list as its input to populate the table. This works out well with the DataGraph because all you need to pass into the dataTable is the root list of the DataGraph. The most common way to lay out the DataGraph in the dataTable is to display each attribute of the DataObject from the list retrieved from the root in its own column, and to embed each additional relationship to the DataObject in a new dataTable contained within the parent DataObject’s row. Using this method instead of a traditional ResultSet table eliminates duplicate information and makes it easier to see the separation of the parent object’s children. An example of how the Customer and Order scenario is laid out in a dataTable is shown in “Example: Using JavaServer Faces and JDBC Mediator dataTables”

Example: Using JavaServer Faces and JDBC Mediator dataTables:

This example shows code that would be located inside of a Faces JSP page. It demonstrates how to use JavaServer Faces and JDBC Mediator dataTables in an application.

It contains the UIData component dataTable tag with all of the customer’s information, along with their orders. Each Customer attribute has its own column. The Customer Orders are embedded in another dataTable containing each of the Order attributes in separate columns. This embedded dataTable of Orders is like any other Customer attribute, having its own column inside each Customer row.

```
<h:dataTable id="table1" value=">{pc_Customers.customer}" var=
"varcustomer" styleClass="dataTable">

  <h:column id="column1">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Customerid" id=
        "text1"></h:outputText>
    </f:facet>
    <h:outputText id="text2" value=">{varcustomer.CUSTOMERID}"
    styleClass="outputText">
```

```
    <f:convertNumber />
   </h:outputText>
   </h:column>

 <h:column id="column2">
   <f:facet name="header">
<h:outputText styleClass="outputText" value="Custfirstname"
    id="text3"></h:outputText>
  </f:facet>
  <h:outputText id="text4" value=">{varcustomer.CUSTFIRSTNAME}"
    styleClass="outputText">
   </h:outputText>
 </h:column>

 <h:column id="column3">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Custlastname"
       id="text5"></h:outputText>
   </f:facet>
   <h:outputText id="text6" value=">{varcustomer.CUSTLASTNAME}"
     styleClass="outputText">
   </h:outputText>
 </h:column>

 <h:column id="column4">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Custstreetaddress"
       id="text7"></h:outputText>
   </f:facet>
   <h:outputText id="text8" value=">{varcustomer.CUSTSTREETADDRESS}"
     styleClass="outputText">
   </h:outputText>
 </h:column>

 <h:column id="column5">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Custcity" id="text9">
       </h:outputText>
   </f:facet>
   <h:outputText id="text10" value=">{varcustomer.CUSTCITY}"
      styleClass="outputText">
   </h:outputText>
 </h:column>

 <h:column id="column6">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Custstate" id=
        "text11"></h:outputText>
   </f:facet>
   <h:outputText id="text12" value=">{varcustomer.CUSTSTATE}"
     styleClass="outputText">
   </h:outputText>
 </h:column>

 <h:column id="column7">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Custzipcode"
        id="text13"></h:outputText>
   </f:facet>
   <h:outputText id="text14" value=">{varcustomer.CUSTZIPCODE}"
      styleClass="outputText">
   </h:outputText>
 </h:column>

 <h:column id="column8">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Custareacode"
```

```
        id="text15"></h:outputText>
    </f:facet>
    <h:outputText id="text16" value=">{varcustomer.CUSTAREACODE}"
       styleClass="outputText">
    <f:convertNumber />
    </h:outputText>
  </h:column>

  <h:column id="column9">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custphonenumber"
         id="text17"></h:outputText>
    </f:facet>
    <h:outputText id="text18" value=">{varcustomer.CUSTPHONENUMBER}"
      styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column10">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Customers_orders"
         id="text19"></h:outputText>
    </f:facet>

    <h:dataTable id="table2" value=">{varcustomer.CUSTOMERS_ORDERS}"
      var="varCUSTOMERS_ORDERS" styleClass="dataTable">

     <h:column id="column11">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Ordernumber"
        id="text20"></h:outputText>
   </f:facet>
   <h:outputText id="text21"
      value=">{varCUSTOMERS_ORDERS.ORDERNUMBER}"
      styleClass="outputText">
     <f:convertNumber />
   </h:outputText>
 </h:column>

 <h:column id="column12">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Orderdate"
        id="text22"></h:outputText>
   </f:facet>
  <h:outputText id="text23" value=">{varCUSTOMERS_ORDERS.ORDERDATE}"
     styleClass="outputText">
     <f:convertDateTime />
   </h:outputText>
 </h:column>

 <h:column id="column13">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Shipdate"
        id="text24"></h:outputText>
   </f:facet>
   <h:outputText id="text25"
     value=">{varCUSTOMERS_ORDERS.SHIPDATE}"
     styleClass="outputText">
     <f:convertDateTime />
   </h:outputText>
 </h:column>

 <h:column id="column14">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Customerid"
       id="text26"></h:outputText>
   </f:facet>
```

```
   <h:outputText id="text27"
       value=">{varCUSTOMERS_ORDERS.CUSTOMERID}" styleClass="outputText">
     <f:convertNumber />
   </h:outputText>
 </h:column>

 <h:column id="column15">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Employeeid"
        id="text28"></h:outputText>
   </f:facet>
   <h:outputText id="text29"
       value=">{varCUSTOMERS_ORDERS.EMPLOYEEID}" styleClass="outputText">
     <f:convertNumber />
   </h:outputText>
 </h:column>

    </h:dataTable>
  </h:column>
</h:dataTable>
```

JDBC mediator paging:

Paging can be useful for moving through large data sets because it can limit the amount of data pulled into memory at any given time. The JDBC DMS API provides two interfaces that implement paging.

If the metadata provided to the data mediator service (DMS) defines customers and the page size is set to ten, then the first page is a DataGraph containing the first ten customer DataObjects. The next page is another DataGraph with the next ten Customers, and so forth.

One thing to note is that the JDBC DMS provides paging at the root of the graph. That is, there is no restriction on the number of related DataObjects returned. For example, if the metadata provided to the DMS defines customers and related orders, it is the customers that are paged. If the page size is set to ten, then the first page is a graph with the first 10 customers and all related orders for each customer.

There are two interfaces provided by the DMS that you can take advantage of, the **Pager** and the **CountingPager**. The Pager interface provides a cursor-like *next()* method capability. The next() function returns a graph representing the next page of data from the entire data set specified by the mediator metadata. There is also a *previous()* function available with the same capabilities, only going backward. The CountingPager interface enables you to retrieve a specific page number. The following example illustrates paging through a large set of customer instances using a CountingPager interface with a maximum of 5 DataObjects from the root table per page.

```
CountingPager pager = PagerFactory.soleInstance.createCountingPager(5);
int count = pager.pageCount(mediator);
for (int i = 1, i <= count, i++) {
DataObject graph = pager.page(i, mediator);
// Iterate through all returned customers in the
// current page.
Iterator iter = graph.getList("CUSTOMER").iterator();
 while (iter.hasNext()) {
  DataObject cust = (DataObject) iter.next();
System.out.println(cust.getString("CUSTFIRS NAME"));
 }
}
```

If you try to move before the first page or after the last available page, a JDBC mediator exception occurs.

JDBC mediator serialization:

The DataGraph produced by the JDBC DMS can be serialized and written out to a file, or sent across a network.

The following example illustrates serialization and de-serialization of a graph:

```
// This example assumes the creation of the Customer
// metadata and the JDBC DMS.

DataObject object = mediator.getGraph();
DataGraph origGraph = object.getDataGraph();

FileOutputStream out = new FileOutputStream("test.datagraph");
ObjectOutputStream oos = new ObjectOutputStream(out);
oos.writeObject(origGraph);
out.close();

FileInputStream in = new FileInputStream("test.datagraph");
ObjectInputStream oin = new ObjectInputStream(in);
DataGraph graph = (DataGraph) oin.readObject();
DataObject obj = (DataObject) graph.getRootObject();

// Now, the DataObject retrieved from the input stream
// obj is equal to the original variable object put
// through the output stream.
```

Enterprise JavaBeans Data Mediator Service

The Enterprise JavaBeans (EJB) Data Mediator Service (DMS) is the Service Data Objects (SDO) Java interface that, given a request in the form of EJB queries, returns data as a DataGraph containing DataObjects of various types.

This is different from a typical EJB finder or ejbSelect method, which also takes an EJB query but returns a collection of EJB objects that are all of the same type or a collection of container managed persistence (CMP) values.

The EJB DMS enables you to specify an EJB query that returns a data graph (the DataGraph) of data objects (DataObjects). The query can be expressed as a compound EJB query that is contained in a string array of EJB query statements. One advantage of using a DataGraph is that much of the code written in an EJB facade session bean that creates, populates, and updates copy helper objects can be replaced with a DataGraph and a DMS.

Note: The EJB DMS has support for EJB2.x container managed persistence (CMP) entity beans only. It does not support EJB 3.0 modules.

You can obtain a DataGraph using the getGraph call, either from EJB instances cached in the container, or the query request can be compiled into SQL and executed directly against the data source.

Updated DataObjects can be written back to the data store by using the applyChanges method in one of two ways. The updates can be translated into SQL and applied directly to the data store or can be written back through EJB accessor methods. Writing back directly to the data store can improve performance because it avoids EJB activation. However, if business logic or EJB container function is required by the application, writing back through EJB is the preferred approach. When writing back through EJB, you can specify a user defined MediatorAdapter method to ensure customized handling of changed DataObjects. This customization can include application specific optimistic concurrency control, invoking business methods on the EJB to perform updates, update of computed values in the DataObject, and calling application specific create methods on EJBHome.

Update processing is not dependent on how the DataGraph was originally retrieved. In other words, it is possible to retrieve a DataGraph directly from the data source, but have the deferred updates applied through the enterprise bean or the other way around.

Regardless of which update approach you use, an optimistic concurrency control algorithm is used. Fields designated as consistency fields are read during the update to ensure that the current value is equal to the old value DataObject field.

Runtime processing

An EJB mediator request is a compound EJB query, which consists of an ordered list of regular EJB queries. Each query in the compound query defines an SDO. The SELECT clause of the query specifies the CMP fields or expressions to return in the DataObject. The WHERE clause specifies the filtering conditions. The first query in the list is considered to be the ROOT node in the DataGraph. The FROM clause of a query, other than the first, specifies an EJB relationship that is used to create references between DataObjects. More details about how the DataGraph schema is derived from the query can be found in “DataGraph schema” on page 1029.

EJB data mediator service programming considerations:

When you begin writing your applications to take advantage of the Enterprise JavaBeans (EJB) data mediator service (DMS) provided in the product, consider the following items.

EJB programming model

Only a subset of the EJB programming model is supported by the EJB data mediator service.

- When using EJB collection parameters to retrieve data from EJB instances, or when using applyChanges to update EJB instances:
  - The EJB DMS uses local interfaces for enterprise beans. *Getter* and *setter* calls for container-managed persistence (CMP) fields must be promoted to the local interface, as well as any EJB methods used in query expressions.
  - For the mediator to create an EJB, there must be a create method using the primary key class as the only argument method defined on the EJB home. If no such method exists, you must supply an adapter that handles the create operation. Also, the EJBLocalHome interface defined for the EJB must include (in addition to the create method) the following method:

    ```
    findByPrimaryKey(<key class>)
    remove (java.lang.Object)
    create (<key class>)
    ```
- When invoking the applyChanges method directly to the database, the following occur:
  - you bypass container update. You should force a refresh as soon as possible by transaction termination and using appropriate container cache options.
  - you bypass EJB container-managed relationship (CMR) maintenance. You must rely on database RI to maintain those relationships not retrieved into the DataGraph.
- CMP fields must be the allowed types. See “EJB mediator query syntax” on page 1022 for a list of those types.
- CMP fields of user-defined types that use EJB converters/composer are not supported.

The following table shows limitations in the EJB programming model that are **not** supported by the EJB DMS.

	retrieve direct from db	retrieve from EJB Container	update direct to db	update through EJB
EJB persistence inheritance	No	No	No	No
EJB cmp field with converter	No	Yes	No	Yes

Transactional

- All mediator calls, including create, must be done within a transaction scope – either a user transaction or a container transaction. The various mediator calls including, create, getGraph, and applyChanges, do not have to be called within the same transaction. In fact, most often the calls are done in separate transactions.

Access intent

- When the mediator query references an EJB using its abstract schema name (ASN), data is retrieved directly from the database. The access intent and isolation level used on the data source connection is the access intent specified in the application profile for EJB dynamic query access intent. It is recommended that you define an optimistic access intent for your application because a DataGraph is intended to be used in a disconnected programming model.
- When the mediator is retrieving data using an EJB collection, the access intent specified in the application profile is used if the EJB requires activation.
- During applyChanges, optimistic concurrency control is used to verify certain fields in the DataObject before applying changes to the database. Updates are typically processed under a different transaction from the retrieval. Therefore, to avoid lost updates it is necessary to verify that another transaction has not updated the data. When defining the EJB to RDB mapping you can specify one or more EJB fields as optimistic Predicates. The fields are used for verification by comparing the current database value to the old value from the DataGraph change log. If the verification succeeds, then the current value of the fields is written to the database. If the comparison returns false and the update fails, an exception occurs. All of this is accomplished in a single update statement with extra predicates added, such as in the following example. The optimisticPredicate field is *myColumn1*.

  ```
  update  myTable set  myColumn1 = 'new value1', myColumn2='new value2'
  where myKey= 'key value' and myColumn1 ='old value1'
  ```

- When applyChanges is done through the EJB container, the current values of the enterprise beans are compared with the old values of the optimistic predicates fields. If the values are unequal an exception occurs.
- Provided that you have defined one or more EJB fields as optimisticPredicates, then for the SDO to be updateable, at least one of the optmisticPredicate fields must be retrieved into the data object. Otherwise, applyChanges returns an exception. The field should be updated either by the caller or a database trigger – the mediator does not automatically increment or set the field.
- Not all fields are verified, only those fields marked as optimisticPredicate in the EJB-RDB mapping.
- Note that the EJB mapping tool allows for the possibility of no optimisticPredicate fields. In this case the mediator will perform updates without any verification.
- Creation and deletion operations do not make use of the optimistic predicate fields.
- When applying changes through EJB instances, the EJB might have to be activated first. In this case, the appropriate access intent associated with the EJB methods apply. It is recommended that you run applyChanges in a profile that has pessimistic access intent, otherwise the optimistic concurrency logic is invoked twice – once when copying data object values to the EJB, and a second time when the persistence manager compares the old values of the EJB field values against the database record.
- The access intent used by the mediator when retrieving directly from the database is the default access intent defined for the EJB named in the first query statement.

Best practices

- You can call getGraph on one mediator instance, update the returned DataGraph, and then call applyChanges on a different mediator instance. However, while they do not need the same mediator instance, they do need the same *query shape*. The query shape is the number and order of query statements, the fields and relationships specified in the SELECT and FROM clauses, and so on.
- Avoid repeated calls to createMediator if possible. Use parameterized queries and use getGraph to pass in different parameter values.

EJB data mediator service data retrieval:

An Enterprise JavaBeans (EJB) mediator request is a compound EJB query. You can obtain a DataGraph using the *getGraph* call.

Directly from the data source

To retrieve data directly from the data source, specify your first EJB query to reference the Abstract Schema Name (ASN) of the EJB.

From the EJB container

To retrieve data through the EJB container, specify your first query to use an input parameter in the FROM clause referring to the EJB collection desired.

You should use this method when there is high likelihood that your EJB instances will be cached in the container. This way you avoid container flush and then read from the database to retrieve data.

For an example, see the section called Collection Input Parameter at “Example: Using query arguments with EJB mediator” on page 1023.

EJB data mediator service data update:

An Enterprise JavaBeans (EJB) mediator request is a compound EJB query. You can write an updated DataGraph back to the data source by using the applyChanges method.

The update can be applied directly to the data source or through EJB instances.

When applying changes through EJB instances an optional adapter class can be specified on the applyChanges method. Each changed data object is first passed to the adapter applyChange method. The adapter can process the change itself and return **true**, or have the EJB Mediator process the change by returning **false**.

The adapter can be used to customize the optimistic concurrency (OCC) logic, or process changes to read only DataGraph attributes, or process changes that require business logic.

There are two forms of the applyChanges method. The first, *applyChanges(DataObject)* takes the updated DataGraph and runs structured query language (SQL) insert, update, and delete statements directly against the database, bypassing the EJB container. The second form, *applyChanges(DataObject, MediatorAdapter)* processes updates using EJB instances and accessors. A null value for the MediatorAdapter is supported.

When to use an adapter with applyChanges

- Use when there are create methods other than create(PrimaryKey)
- Use when business methods must be called instead of container-managed persistence (CMP) *setter* methods
- Use when special optimistic caching logic is needed

How the adapter works

Three passes are made over the DataGraph log, passing changed DataObject to the adapter:

1. New DataObjects are passed. The adapter can create the object and set the CMP fields. Container-managed relationships (CMR) that reference enterprise beans not yet created are deferred until pass 2.
2. New and updated DataObjects are passed. CMRs deferred from pass 1 can be set at this time.
3. Deleted DataObjects are passed.

Example: using MediatorAdapter:

In this example, the adapter processes CREATE events for an EMP data object. The name and salary attributes are extracted from the data object and passed to the create method on the EmpLocalHome.

The create method returns an instance of Emp EJB and the primary key value is copied back to the DataObject. The caller can then obtain the generated key value. After processing, the adapter returns a value of **true**. All other changes are ignored by the adapter and processed by the EJB Mediator.

```
package com.example;
import com.ibm.websphere.sdo.mediator.ejb.*;
import javax.naming.InitialContext;
import commonj.sdo.ChangeSummary;
import commonj.sdo.DataObject;
import commonj.sdo.DataGraph;
import commonj.sdo.ChangeSummary;

// example of Adapter class calling a EJB create method.

public class SalaryAdapter implements MediatorAdapter{

 ChangeSummary log = null;
      EmpLocalHome empHome = null;

 public boolean applyChange(DataObject object,  int phase){

  if (object.getType().getName().equals("Emp")
       && phase == MediatorAdapter.CREATE){
   try{
    String name = object.getString("name");
    double salary = object.getDouble("salary");
    EmpLocal emp = empHome.create(name, salary);
    object.set("empid", emp.getPrimaryKey() ); // set primary key in SDO
    return true;
   } catch(Exception e){ // error handling code goes here
                  }
  }
  return true;
 }

 public void init (ChangeSummary log){
         try {
  this.log = log;
  InitialContext ic = new InitialContext();
  empHome = (EmpLocalHome)ic.lookup( "java:comp/env/ejb/Emp");
         } catch (Exception e) { // error handling code goes here
         }
 }

 public void end(){}
}
```

EJB mediator query syntax:

When you begin writing your applications to take advantage of the Enterprise JavaBeans (EJB) data mediator service (DMS) provided in the product, consider the following items.

- The EJB DMS takes as an input argument a compound EJB query which consists of an array containing EJB query language (QL) statements and an optional XREL command. The XREL command is a list of EJB relationships and must appear last in the array.
- Each EJB QL query returns data in the form of a Service DataObjects (SDO) instance. All of the SDO instances are merged into a DataGraph. The SELECT clause of each query specifies the container-managed persistence (CMP) fields or expressions to return in the SDO. The WHERE clause specifies the filtering conditions and you can define an ORDER BY clause. If two or more SELECTs

return the same SDO type, each SELECT must project the same CMP fields and expressions. For updatability, the primary key fields of the EJB must be projected. JOINs, UNIONs, and aggregation are not supported except in subqueries.

- A query in the array can refer to a prior query in the FROM clause by using the identification variable defined in the prior query and a relationship name. This relationship can be single or collection valued.
- Relationships are constructed between data object instances in the graph when a relationship is used in either the FROM clause or in the XREL command.
- Collection valued input arguments are supported in FROM clause If ?1 refers to a collection of Dept EJBs then the following query is valid for the mediator. The cast syntax is required to tell the query compiler the collection element type.

  ```
  select d.deptno from (Dept) ?1 as d
  ```

- The collection input argument is useful when it is desired to build a DataGraph from EJB instances that are cached in the EJB Container or persistence manager data cache.
- The SELECT clause can specify a list of CMP fields to retrieve (the wildcard * notation can be used to retrieve all CMP fields) or valid EJB query language expressions. CMP fields and expressions must be one of the following types:
  - Primitive types: boolean, byte, short, integer, long, float, double, char
  - Object wrapper types for the primitive types
  - Java.lang.String
  - Java.math.BigDecimal
  - java.math.BigInteger
  - byte [ ]
  - Java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - java.util.Date
  - java.util.Calendar
- All primary key CMP fields must be retrieved in order for the Service Data Objects (SDO) to be updateable; otherwise, applyChanges returns an exception.
- SDO attributes that come from EJB query language expressions such as *e.salary + e.bonus AS TOTAL_PAY* cannot be updated. If you try to make an update, applyChanges returns a QueryException.
- Aggregate expressions such as *SUM(e.salary)* are not allowed even though they are part of the EJB query language. Aggregate expressions can be used in subselects in the WHERE clause.

Example: Using query arguments with EJB mediator:

The following examples show how you can fine-tune your Enterprise JavaBeans (EJB) mediator query arguments.

A simple example

This query returns a DataGraph containing multiple instances of DataObjects of type (Eclass name) *Emp*. The data object attributes are *empid* and *name* and their data types correspond to the container-managed persistence (CMP) field types.

```
select  e.empid, e.name
from Emp as e
where e.salary > 100
```

The returned DataGraph serialized in its XML format looks like this:

```
  <?xml version="1.0" encoding="ASCII"?>
<datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
<root>
```

```
<Emp empid="1003" name="Eric" />
<Emp empid="1004" name="Dave" />
</root>
</datagraph:DataGraphSchema>
```

Query parameters

This example shows how parameter markers can be used. Recall that the syntax for parameter markers in an EJB query is a question mark followed by a number (?n). When calling the getGraph ( ) method on the EJBMediator, you can optionally pass an array of values. ?n refers to the value of parm[n-1]. The array of values can also be passed on the factory call to create the EJBMediator. Parameters passed on the getGraph( ) override any parameters passed on the create call.

```
select  e.empid, e.name
from Emp as e
where e.salary > ?1
```

Returning expressions and methods

This example illustrates that the data object attributes can be the return values of query expressions. EJB query expressions include arithmetic, date-time, path expressions, and methods. Input arguments and return values from methods are restricted to the list of supported data types (see “EJB mediator query syntax” on page 1022). A data object containing an updated attribute derived from an expression causes an exception to occur during the applyChanges process unless the user has provided a MediatorAdapter to handle the change.

```
select  e.empid as employeeId,
  e.bonus+e.salary as totalPay,
  e.dept.mgr.name as managerNam,
  e.computePension( ) as pension
from Emp as e
where e.salary > 100
```

Data object attribute names are derived from the CMP field names but can be overridden by using the *AS* keyword in the query. When specifying an expression, the *AS* keyword should always be used to give a name to the expression.

The * syntax

The notation *e.** is a short cut for specifying all the CMP fields (but not container-managed relationships) for an EJB. The following query means the same thing as *e.empid, e.name e.salary, e.bonus*.

```
select e.* from Emp as e
```

No primary key in select clause

This example shows a query that does not return the primary key field. However, unless the data object contains all the primary key fields for an EJB, updates to the DataGraph cannot be processed by the mediator. This is because the primary key is required to translate the changes into structured query language (SQL), or to convert DataObject references to EJB references. An exception when applyChanges tries to run.

```
select e.name, e.salary from Emp as e
```

Order by

DataObjects can be ordered.

```
select d.* from Dept d  order by d.name
 select e.* from in(d.emps) e order by e.empid desc
```

This results in the *Dept* objects being ordered by name and the *Emp* objects within each *Dept* being order by *empid* in descending order.

Navigating a multi-valued relationship

This compound query returns a DataGraph with DataObject classes *Dept* and *Emp*. The shape of the DataGraph reflects the path expressions used in the FROM clauses.

```
select d.deptno, d.name, d.budget from Dept d
   where d.deptno < 10
select e.empid, e.name, e.salary  from in(d.emps) e
   where e.salary > 10
```

In this case *Dept* is the root node in the DataGraph and there is a multivalued reference from *Dept* to *Emp* as shown:

```
<?xml version="1.0" encoding="ASCII" ?>
<datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
<root>
<Dept deptno="1" name="WAS_Sales" budget="500.0"
   emps="//@root/@Emp.1 //@root/@Emp.0" />
<Dept deptno="2" name="WBI_Sales" budget="450.0"
   emps="//@root/@Emp.3 //@root/@Emp.2" />
<Emp empid="1001" name="Rob" salary="100.0" EmpDept="//@root/@Dept.0" />
<Emp empid="1002" name="Jason" salary="100.0" EmpDept="//@root/@Dept.0" />
<Emp empid="1003" name="Eric" salary="200.0" EmpDept="//@root/@Dept.1" />
<Emp empid="1004" name="Dave" salary="500.0" EmpDept="//@root/@Dept.1" />
</root>
</datagraph:DataGraphSchema>
```

More on query parameters

Search conditions can be specified on any query. Input arguments are global to the query and can be referenced by number anywhere in the compound query. In the example above, the query arguments passed on the create or getGraph call should be in order { deptno value, salary value, deptno value }.

```
select d.* from Dept as d
   where d.deptno between ?1 and ?3
select e.* from in(d.emps) e
   where e.salary < ?2
```

Navigating a path with multiple relationships

The following query navigates the path composed of EJB relationships Dept.projs and Project.tasks and returns DataObjects for Dept, Emp and Project containing selected CMP fields.

```
select d.deptno, d.name from Dept as d
select p.projid  from in(d.projects) p
select t.taskid, t.cost from  in (p.tasks) t
```

The resulting data graph in XML format is shown here.

```
<?xml version="1.0" encoding="ASCII" ?>
<datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
<root>
<Dept deptno="1" name="WAS_Sales" projects="//@root/@Project.0" />
<Dept deptno="2" name="WBI_Sales" projects="//@root/@Project.1" />
<Project projid="1" ProjectDept="//@root/@Dept.0"
      tasks="//@root/@Task.0   //@root/@Task.2 //@root/@Task.1" />
<Project projid="2" ProjectDept="//@root/@Dept.1"
      tasks="//@root/@Task.3" />
<Task taskid="1" cost="50.0" TaskProject="//@root/@Project.0" />
<Task taskid="2" cost="60.0" TaskProject="//@root/@Project.0" />
```

```
<Task taskid="3" cost="900.0" TaskProject="//@root/@Project.0" />
<Task taskid="7" cost="20.0" TaskProject="//@root/@Project.1" />
</root>
</datagraph:DataGraphSchema>
```

Navigating multiple paths

Here is a mediator query returning a DataGraph with DataObjects for Dept with related employees and a second path that retrieves related projects and tasks.

```
select d.deptno, d.name from Dept d
select e.empid, e.name  from in(d.emps) e
select p.projid from in(d.projects) p
select t.taskid, t.cost from in(p.tasks) where t.cost > 10
```

The returned DataGraph looks like this:

```
<?xml version="1.0" encoding="ASCII" ?>
    <datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
    <root>
        <Dept deptno="1" name="WAS_Sales" projects="//@root/@Project.0"
 emps="//@root/@Emp.1 //@root/@Emp.0" />
        <Dept deptno="2" name="WBI_Sales" projects="//@root/@Project.1"
     emps="//@root/@Emp.3 //@root/@Emp.2" />
        <Project projid="1" ProjectDept = "//@root/@Dept.0"
     tasks="//@root/@Task.0 //@root/@Task.2 //@root/@Task.1" />
        <Project projid="2" ProjectDept="//@root/@Dept.1" tasks="//@root/@Task.3" />
      <Task taskid="1" cost="50.0" TaskProject="//@root/@Project.0" />
        <Task taskid="2" cost="60.0" TaskProject="//@root/@Project.0" />
        <Task taskid="3" cost="900.0" TaskProject="//@root/@Project.0" />
        <Task taskid="7" cost="20.0" TaskProject="//@root/@Project.1" />
        <Emp empid="1001" name="Rob" EmpDept="//@root/@Dept.0" />
        <Emp empid="1002" name="Jason" EmpDept="//@root/@Dept.0" />
         <Emp empid="1003" name="Eric" EmpDept="//@root/@Dept.1" />
         <Emp empid="1004" name="Dave" EmpDept="//@root/@Dept.1" />
  </root>
  </datagraph:DataGraphSchema>
```

Navigating a single valued relationship

The important thing to point out here is that even though Emp is the root data object in the graph, multiple Emp data objects will be related to the same Dept data object. So unlike the previous examples, the data graph does not have a tree shape when you look at the data object instances – there are multiple root Emp objects related to the same Dept object. But then after all it is a data graph, not a data tree. Note that mediator queries allow single valued path expressions in the FROM clause. This is a change from the standard EJB query syntax.

```
select e.empid, e.name  from Emp e
select d.deptno, d.name  from in(e.dept) d
```

And the DataGraph in XML format looks like:

```
<?xml version="1.0" encoding="ASCII" ?>
   <datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
   <root>
   <Emp empid="1001" name="Rob" dept="//@root/@Dept.0" />
   <Emp empid="1002" name="Jason" dept="//@root/@Dept.0" />
   <Emp empid="1003" name="Eric" dept="//@root/@Dept.1" />
   <Emp empid="1004" name="Dave" dept="//@root/@Dept.1" />
   <Dept deptno="1" name="WAS_Sales"
DeptEmp="//@root/@Emp.1 //@root/@Emp.0" />
   <Dept deptno="2" name="WBI_Sales"
 DeptEmp="//@root/@Emp.3 //@root/@Emp.2" />
 </root>
 </datagraph:DataGraphSchema>
```

Path expressions in the SELECT clause

This query is similar to the preceding one (both queries return employee data along with department number and name) but note the data graph contains only one data object type in this query (vs. two in the previous query). The fields deptno and name field are read only because they are result of a path expression in the SELECT clause and are not CMP fields of the Emp EJB.

```
select  e.empid as EmplId , e.name as EmpName ,
   e.dept.deptno as DeptNo , e.dept.name as DeptName
from Emp as e
```

Navigating a many: many relationship

The Emp to Task relationship is deemed a *many:many* relationship. The following query retrieves employees, tasks, and projects. There is only a single occurrence of any particular task DataObject in the DataGraph, even though it can be related to many employees.

```
select e.empid, e.name from Emp as e
select t.taskid, t.description from in(e.tasks) as t
select p.projid, p.cost from in(t.proj) as p
```

Multiple links between data objects

The EJB mediator enables you to retrieve data based on relationships and use the XREL command to construct one or more additional relationships based on data already retrieved. The mediator also enables retrieval of data based on ASNname and then construction of one or more relationships based on the data retrieved using the XREL command. The following query retrieves departments, employees that work in the departments, and the employees that manage the departments.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from in (d.emps) as e
select m.empid, m.name from in(d.manager) as m
```

The second and third *select* clauses both return instances of Emp DataObject. It is possible that the same Emp instance is retrieved through the d.emps relationship and the d.manager relationship. The EJB mediator creates one Emp instance, but creates both relationships.

The following query is processed as follows. Dept DataObjects are created from the data in the first query. Emp DataObjects are created from the data in the second query. Relationships in the graph are then constructed for any relationship used in either the FROM clause or an XREL keyword. During relationship construction, no additional data is retrieved. In this example, an employee who works in a department named *Dev* appears in the DataGraph. If this employee manages a department called *Sales*, the *manages* reference is empty. The Dev department was retrieved in the first query, not the Sales department.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from in (d.emps) as e
xrel d.manager
```

The emps and manager relationship are constructed based on the DataObject instances created from the queries. An employee whose name is ‘Dev’ but works in department ‘Sales’ will have a null dept relationship in the graph.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from Emp e where e.name like 'Dev%'
xrel d.emps, d.manager
```

The next example shows the retrieval of data objects for all the employees, projects, and tasks for a given department, and the linkage of employees with tasks.

```
select d.deptno from Dept d where d.deptno = 42
select e.empid from in(d.emps) e
select p.projid from in(d.projs) p
select t.* from in(p.tasks) t
  xrel e.tasks
```

If a task is assigned to an employee in department 42 then that link appears in the data graph. If the task is assigned to an employee not in department 42, then that link does not appear in the data graph because the data object was filtered out by the query. An XREL keyword can be followed by one or more EJB relationships. Bidirectional relationships can refer to either role name. Both source and target of the relationship must be retrieved by one or more queries.

Retrieving unrelated objects

The following query retrieves Dept and Task.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select t.taskid, t.startDate from Task t where t.startDate > '2005'
```

The following query retrieves Dept and Emps. Even though there are relationships between Dept and Emp (namely mgr and emps), neither relationship is used in FROM or XREL and so the resulting graph does not contain the relationship values.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from Emp e where e.dept.name like '%Dev%'
```

Retrieving null or empty relationships

This query returns departments that have no employees and employees with no department. Presumably the application wants to assign the employees to one of the departments. The purpose of xrel is to define the e.dept relationship (and the inverse role d.emps) into the graph schema.

```
select d.deptno, d.name from Dept d where d.emps is empty
select e.empid, e.name from Emp e where e.dept is null
xrel e.dept
```

Collection Input parameter

A collection of enterprise beans can be passed as an input argument to the ejb mediator and referenced in the FROM clause. Using a collection parameter satisfies the requirement to construct a data graph from a user collection of already activated enterprise beans.

```
select d.deptno, d.name from ((Dept) ?1) as d
select e.empid, e.name from in(d.emps) as e where e.salary > 10
```

The above query iterates through the collection of Dept beans and related Emp beans applying the query predicates and constructing the data graph. Values will be obtained from current values of the beans. An example of a program using an ejb collection parameter.

```
// this method runs in an EJB context and within a transaction scope
public DataGraph myServiceMethod() {
      InitialContext ic = new InitialContext();
      DeptLocalHome deptHome = ic.lookup("java:comp/env/ejb/Dept");
      Integer deptKey = new Integer(10);
      DeptEJB dept = deptHome.findByPrimaryKey( deptKey));
      Iterator i = dept.getEmps().iterator();
      while (i.hasNext()) {
       EmpEJB e = (EmpEJB)i.next();
       e.setSalary( e.getSalary() * 1.10); // give everyone a 10% raise
      }

      // create the query collection parameter
      Collection c = new LinkedList();
      c.add(dept);
```

```
      Object[] parms = new Object[] { c};  // put ejb collection in parm array.

      // collection containing the dept EJB is passed to EJB Mediator

      String[] query = new String[]
          { "select d.deptno, d.name from ((Dept)?1 ) as d",
             "select e.empid, e.name, e.salary " +
                 " from in (d.employees) as e",
            "select p.projno, p.name from in (d.projects) as p" };

      Mediator m = EJBMediatorFactory.getInstance().createMediator(
 query, parms);
      DataGraph dg = m.getGraph();
      return dg;
      // the DataGraph contains the updated and as yet uncommitted
//  salary information.  Dept and Emp data
      // is fetched through EJB instances active in the EJBContainer.
      //  Project data is retrieved from database using
      //   container managed relationships.
   }
```

XREL keyword:

The XREL keyword is used to build relationships independent of how the data was retrieved. XREL is valid only in Enterprise JavaBeans (EJB) Mediator queries.

XREL does not retrieve additional data, it only builds relationships from data already retrieved by the select statements. The relationships can be one-to-one, one-to-many, many-to-one, or many-to-many. The relationships can be unidirectional or bidirectional. If you specify a bidirectional relationship in an XREL, the inverse relationship is also established in addition to the specified relationship.

```
xrel := XREL identification_variable . { single_valued_cmr_field | collection_valued_cmr_field }
             [ , identification_variable . { single_valued_cmr_field | collection_valued_cmr_field } ]*
```

Examples: XREL keyword

This example retrieves all employees and all departments, and establishes the *emps* and *mgr* relationships.

```
select e.name from EmpBean e
  select  d.name from DeptBean d
  xrel  d.emps, d.mgr
```

Notice that the employees are retrieved through d.emps relationship, xrel d.mgr is to establish the *mgr* relationship for those employees who are also a manager.

```
 select  d.name from DeptBean d
   select e.name from in(d.emps) e
   xrel  d.mgr
```

DataGraph schema:

DataGraph schema created by the EJB mediator

The schema created by the mediator for a query consists of an Eclass for each query statement. The name of the Eclass is the Abstract Schema Name (ASN) of the Enterprise JavaBean (EJB). The Eattributes of the Eclass correspond to the container-managed persistence (CMP) fields or expressions returned by the query statement.

For static DataObjects, the Eclass name can be different provided that the Map argument is used on the createMediator call.

Each EJB relationship specified in the FROM or XREL clause adds an Ereference into the schema. EJB relationships can be unidirectional or bidirectional. However, all Ereferences are defined as bidirectional as this is needed to efficiently navigate the DataGraph on update. An inverse relationship name is generated in the case of a unidirectional EJB relationship. A generated name is of the format <ASName_source><ASName_target>. For example, if the ASNames are EmpBean and DeptBean, and the unidirectional relationship is *dept* going from EmpBean to DeptBean, the generated inverse name is **DeptBeanEmpBean**.

If no ECLass argument is used on createMediator, then the mediator creates a DataGraph schema with the following characteristics:
- the DataObject Eclass names are the corresponding EJB Abstract Schema Names (ASN)
- the DataObject attributes names and types are the expression names and types in the query SELECT clauses
- the DataObject reference names and types come from the EJB relationships referenced in the FROM clauses.

A *dummy* DataObject with the Eclass name of *DataGraphRoot* is also created and has containment reference to all the DataObjects. The reference is multivalued, using the EJB ASN name.

```
DataObject root =  m.getGraph( parms );
root.getType().getName();   // this would return the string "DataGraphRoot"

List depts = (List) root.get("DeptBean");
// the list of all DeptBean SDOs in the DataGraph

List emps = (List) root.get("EmpBean");
// the list of all EmpBean SDOs in the DataGraph
```

DataGraph containment patterns

References between Service Data Objects (SDO) can be defined as containment references, in which case when an SDO is deleted the delete is cascaded to all of the contained SDO. Also, when the DataGraph is serialized as an XML document, the contained SDO are nested within the parent SDO. Noncontained references are expressed as path expressions in the XML document.

Containment must be defined in the DataGraph schema. When the mediator defines the schema, the root SDO (named *DataGraphRoot*) contains all other SDO. EJB relationships are defined as noncontained SDO references.

When the caller defines the DataGraph schema, there are three patterns.

ROOT_CONTAINS_ALL
: In this pattern there is a dummy SDO that is the root. It is a dummy in the sense that it does not correspond to any EJB. Its purpose is to contain all other SDOs. If the mediator generates the graph schema, the dummy root has a class name of DataGraphRoot and it will have containing references whose names are the EJB ASN names. If the caller uses static schema, the root can have any name. The Eclass of the root is passed on the createMediator call.

ROOT_CONTAINS_SOME
: This pattern is applicable only for static schema. There is still a dummy SDO that is the graph root. Other SDO must either be contained by the Ereference that corresponds to the EJB relationship used in the query statement or the SDO must be contained by the dummy root.

NO_DUMMY ROOT
: This pattern is applicable only for static schema. There is no dummy root. The root SDO corresponds to the first query statement which must return only a single instance. Non-root SDOs must be contained by the Ereference corresponding to the EJB relationship used in the query statement.

Service Data Objects: Resources for learning

Use the following links to find relevant supplemental information about the service data object and various other functions that can be used with it. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Service Data Objects

For an introduction to Service Data Objects, refer to:
- Introduction to Service Data Objects

For an overview of the Service Data Objects specification, refer to:
- Service Data Objects

A good place to start to learn about the Eclipse Modeling Framework is:
- EMF Eclipse Modeling Framework

Information about XSD to SDO/EMF mapping for Version 6 can be found at:
- XML Schema to Ecore Mapping

Web application presentation layer technologies

For a brief overview of JavaServer Faces, refer to:
- Java Sun J2EE 1.4 tutorial

Good places to start to learn about JavaServer Pages Standard Tag Library are:
- JavaServer Pages Standard Tag Library
- A JSTL primer, Part 1: The expression language

Using the Java Database Connectivity data mediator service for data access

The following steps demonstrate how to create the metadata for a Java Database Connectivity (JDBC) data mediator service (DMS), as well as how to instantiate the DMS dataGraph.

1. Create the metadata factory. This can be used for creating metadata, tables, columns, filters, filter arguments, database constraints, keys, order-by objects, and relationships.

   ```
   MetadataFactory factory = MetadataFactory.eINSTANCE;
   Metadata metadata = factory.createMetadata();
   ```

2. Create the table for the metadata. You can do this two ways. Either the metadata factory can create the table and then the table can add itself to the already created metadata, or the metadata can add a new table in which case a new table is created. Because it involves fewer steps, this example uses the second option to create a table called CUSTOMER.

   ```
   Table custTable = metadata.addTable("CUSTOMER");
   ```

3. Set the root table for the metadata. Again, you can do this in two ways. Either the table can declare itself to be the root or the metadata can set its own root table. For the first option, code:

   ```
   custTable.beRoot();
   ```

   If you want to use the second option, you code:

   ```
   metadata.setRootTable(custTable)
   ```

4. Set up the columns in the table. The example table is called CUSTOMER. Each column is created using its type. The column types in the metadata can only be the types supported by the JDBC driver being used. If you have questions on which types the JDBC driver being used supports, consult the JDBC driver documentation.

```
Column custID = custTable.addIntegerColumn("CUSTID");
custID.setNullable(false);
```

This example creates a column object for this column, but does not for the remainder. The reason is because this column is the primary key, and is used to set the table's primary key after the rest of the columns are added. A primary key cannot be null; therefore custID.setNullable(false) prohibits this from happening. Adding the rest of the columns:

```
custTable.addStringColumn("CUSTFIRSTNAME");
custTable.addStringColumn("CUSTLASTNAME");
custTable.addStringColumn("CUSTSTREETADDRESS");
custTable.addStringColumn("CUSTCITY");
custTable.addStringColumn("CUSTSTATE");
custTable.addStringColumn("CUSTZIPCODE");
custTable.addIntegerColumn("CUSTAREACODE");
custTable.addStringColumn("CUSTPHONENUMBER");

custTable.setPrimaryKey(custID);
```

5. Create other tables as needed. For this example, create the Orders table. Each order is made by one Customer.

```
Table orderTable = metadata.addTable("ORDER");

Column orderNumber = orderTable.addIntegerColumn("ORDERNUMBER");
orderNumber.setNullable(false);

orderTable.addDateColumn("ORDERDATE");
orderTable.addDateColumn("SHIPDATE");
Column custFKColumn = orderTable.addIntegerColumn("CUSTOMERID");

orderTable.setPrimaryKey(orderNumber);
```

6. Create foreign keys for the tables that need relationships. In this example, orders have a foreign key that points to the customer who made the order. In order to create a relationship between the two tables, you must first make a foreign key for the Orders table.

```
Key custFK = factory.createKey();
custFK.getColumns().add(custFKColumn);

orderTable.getForeignKeys().add(custFK);
```

The relationship takes two keys, the parent key and the child key. Because no specific name is given, the default concatenation of CUSTOMER_ORDER is the name used for this relationship.

```
metadata.addRelationship(custTable.getPrimaryKey(), custFK);
```

The default relationship includes all customers who have orders. To get all customers, even if they do not have orders, you need this line as well:

```
metadata.getRelationship("CUSTOMER_ORDER")
                            .setExclusive(false);
```

Now that the two tables are related to one another you can add a filter to the Customer table to find customers with specific characteristics.

7. Specify any filters needed. In this example, set filters to the Customer table to find all the customers in a particular state, with a certain last name, who have made orders.

```
Filter filter = factory.createFilter();
filter.setPredicate("CUSTOMER.CUSTSTATE = ? AND CUSTOMER.CUSTLASTNAME = ?");

FilterArgument arg1 = factory.createFilterArgument();
arg1.setName("CUSTSTATE");
```

```
arg1.setType(Column.STRING);
filter.getFilterArguments().add(arg1);

FilterArgument arg2 = factory.createFilterArgument();
arg2.setName("CUSTLASTNAME");
arg2.setType(Column.STRING);
filter.getFilterArguments().add(arg2);

custTable.setFilter(filter);
```

8. Add any order by objects needed. In this example, set the order by object to sort by the customer's first name.

```
Column firstName = ((TableImpl)custTable).getColumn("CUSTFIRSTNAME");
OrderBy orderBy = factory.createOrderBy();
orderBy.setColumn(firstName);
orderBy.setAscending(true);
metadata.getOrderBys().add(orderBy);
```

   This completes the creation of the metadata for this JDBC DMS.

9. Create a connection to the database. This example does not show the creation of the connection to the database; it assumes that the SDO client calls the method *connect()* that does that.
10. Instantiate and initialize the JDBC DMS object (dataGraph). The SDO client performs these actions. For this example:

```
ConnectionWrapperFactory factory = ConnectionWrapperFactory.soleInstance;
connectionWrapper = factory.createConnectionWrapper(connect());
JDBCMediatorFactory mFactory = JDBCMediatorFactory.soleInstance;
JDBCMediator mediator = mFactory.createMediator(metadata, connectionWrapper);
DataObject parameters = mediator.getParameterDataObject();
parameters.setString("CUSTSTATE", "NY");
parameters.setString('CUSTLASTNAME', 'Smith');
DataObject graph = mediator.getGraph(parameters);
```

    Now that you have the dataGraph, you can manipulate the information as you wish. Some simple examples are contained in "Example: Manipulating data in a DataGraph object."

11. Submit the changed information to the DMS for updating the database.

Example: Manipulating data in a DataGraph object

This example shows how to do basic manipulation of data in a DataGraph object which you have created.

Using the simple DataGraph that was created during the task "Using the Java Database Connectivity data mediator service for data access" on page 1031, some typical data manipulation follows.

First **get the list of customers**, then for each customer **get every order**, then **print out** the customer's first name and order date. (For this example, assume that you already know the last name is Smith).

```
List customersList = graph.getList("CUSTOMER");
Iterator i = customersList.iterator();
while (i.hasNext())
{
 DataObject customer = (DataObject)i.next();
List ordersList = customer.getList("CUSTOMER_ORDER");
Iterator j = ordersList.iterator();
while (j.hasNext())
{
DataObject order = (DataObject)j.next();
System.out.print( customer.get("CUSTFIRSTNAME") + " ");
System.out.println( order.get("ORDERDATE"));
 }
}
```

Now **change** every customer with the name Will to be Matt.

```
i = customersList.iterator();
while (i.hasNext())
{
 DataObject customer = (DataObject)i.next();
 if (customer.get("CUSTFIRSTNAME").equals("Will"))
 {
  customer.set("CUSTFIRSTNAME", "Matt");
 }
}
```

Delete the first Customer entry.

```
((DataObject) customersList.get(0)).delete();
```

Add a new DataObject to the graph

```
DataObject newCust = graph.createDataObject("CUSTOMER");
newCust.setInt("CUSTID", 12345);
newCust.set("CUSTFIRSTNAME", "Will");
newCust.set("CUSTLASTNAME", "Smith");
newCust.set("CUSTSTREETADDRESS", "123 Main St.");
newCust.set("CUSTCITY", "New York");
newCust.set("CUSTSTATE", "NY");
newCust.set("CUSTZIPCODE", "12345");
newCust.setInt("CUSTAREACODE", 555);
newCust.set("CUSTPHONENUMBER", "555-5555");

graph.getList("CUSTOMER").add(newCust);
```

Submit the changes.

```
mediator.applyChanges(graph);
```

Using the EJB data mediator service for data access

The following steps use code samples to describe a simple instance of how to create the Enterprise JavaBeans (EJB) data mediator service (DMS) metadata.

1. A mediator instance is created using one of the create methods on the mediator factory (com.ibm.websphere.sdo.mediator.ejb.MediatorFactory) as in the following example

   ```
   import com.ibm.websphere.sdo.mediator.ejb.Mediator;
   import com.ibm.websphere.sdo.mediator.ejb.MediatorFactory;
   import com.ibm.webpshere.ejbquery.QueryException;
   import commonj.sdo.DataObject;

   try{
    String[] query =  { "select d.deptno,d.name from DeptBean as d" };
    Mediator m = MediatorFactory.getInstance().createMediator( query, null);
    DataObject root = m.getGraph();
   } catch (QueryException e) { ... }
   ```

2. There are 3 different forms of the createMediator method. The arguments are explained below.

   ```
   createMediator( query, parms)
   createMediator( query, parms, schema )
   createMediator( query, parms, schema,  typeMap,  pattern)
   ```

   The arguments to the createMediator method are:

String	query	array of EJB query statements
Object	parms	values for input parameters of the query statements
Eclass*	schema	the EClass of the root DataObject
Map*	typeMap	a java.util.Map that maps EJB Abstract Schema Names from the query statement into Eclass names
int*	pattern	the pattern used for containment

* used only when using caller provided schema

Connection thread identity

The application server for z/OS allows you to assign a thread identifier as an owner of a connection, when you first obtain the connection. The thread identity function only applies to Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) resource adapters and Relational Resource Adapter (RRA) wrappered Java Database Connectivity (JDBC) providers that support the use of thread identity for connection ownership.

In this article the term thread identity refers to the Java EE Identity (such as the RunAs Identity), as opposed to the OS thread identity. Refer to Synchronizing a Java thread identity and an operating system thread identity and Understanding Connection Manager RunAs Identity Enabled and operating system security for more information.

The following table lists the JCA resource adapter and JDBC provider processes that support thread identity and thread security. It also provides the level of thread identity support:

Connectors	Thread identity support	OS thread security
IMS Connector - local `ConnectionFactory` configuration	ALLOWED	Not supported
IMS Connector - remote `ConnectionFactory` configuration	NOTALLOWED	Not supported
CTG CICSECIConnector - local `ConnectionFactory` configuration	ALLOWED	Not supported
CTG CICSECIConnector - remote `ConnectionFactory` configuration	NOTALLOWED	Not supported
IMS JDBC Connector - local `ConnectionFactory` configuration (By default, IMS JDBC only supports this type of configuration.)	REQUIRED	True
RRA DB2 for z/OS local JDBC provider - data sources configured to the local DB2	ALLOWED	True
RRA DB2 Universal JDBC Driver Provider using Type 2 connectivity	ALLOWED	True
RRA DB2 Universal JDBC Driver Provider using Type 4 connectivity	NOTALLOWED	Not supported
WebSphere MQ JMS Provider: Connection Factory (TransportType = BINDINGS)	ALLOWED	True
WebSphere MQ JMS Provider - Connection Factory (TransportType = CLIENT)	NOTALLOWED	Not supported
WebSphere JMS Provider (such as Integral JMS Provider): Connection Factory	NOTALLOWED	Not supported

WebSphere Application Server for z/OS allows resource adapters and JDBC providers to define the level of thread identity support for the defined connection factories or data sources. The level of support can be:

- **`ALLOWED`**, which indicates thread identity for connection ownership is allowed for this configuration.

- **NOTALLOWED**, which indicates thread identity for connection ownership is not allowed for this configuration.
- **REQUIRED**, which indicates thread identity for connection ownership is required.

The thread identity function is only available in those server configurations where JCA connectors or JDBC providers access local z/OS resources through callable (not TCP/IP) interfaces. So, for example, CICS and IMS provide thread identity support only if the target CICS or IMS is configured on the same system as the z/OS WebSphere Application Server.

To use thread identity when getting connections to a connection factory or JDBC data source for your application, you must specify **resauth=Container** for the connection factory or JDBC data source. Use the Eclipse assembly tool or WebSphere Studio Application Developer Integration Edition (WSADIE) to indicate the **resauth=Container** setting.

When the level of thread identity support provided by the connector configuration is **ALLOWED**, if you want to use thread identity for the connections, you cannot specify a Container-managed alias when you define the connection factory or JDBC data source. If you specify a Container-managed alias, the userid defined by the alias is assigned as the owning id for the connections obtained by the application.

When the JDBC provider supports thread identity, the thread identity function is only used when data sources configured for that provider are used by Version 2.0 EJB modules and Version 2.3 servlets.

WebSphere Application Server for z/OS also allows supported resource adapters and JDBC providers to enable *OS thread security* in conjunction with thread identity support. You can use OS thread security when:
- The server configuration supports both thread identity and thread security.
- The Connection Manager RunAs Identity Enabled property is enabled.

  You can configure the server to allow Connection Manager RunAs Identity Enabled support. To enable this option, click **Security** → **Global security** → **z/OS security options** in the administrative console. On the z/OS security options panel, select the **Enable the connection manager RunAs thread identity** option, and click **Apply**.

If these conditions are met, the system creates an access control environment element (ACEE) for the user associated with the thread.

Users of previous versions of WebSphere Application Server for z/OS will note that the instructions for enabling OS Thread Security have changed. Previously, OS Thread Security was enabled via a checkbox named Enable Synch to Thread. This checkbox still exists, but it no longer is associated with any Connection Management functionality. Users who wish to enable OS Thread Security must now use the checkbox named Connection Manager RunAs Identity Enabled

Using thread identity support

The thread identity function allows you to assign a thread identifier as an owner of a connection when you first obtain the connection. This function only applies to Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) resource adapters and Relational Resource Adapter (RRA) wrappered Java Database Connectivity (JDBC) providers that support the use of thread identity for connection ownership.

About this task

In this article the term thread identity refers to the Java EE Identity (such as the RunAs Identity), as opposed to the OS thread identity. Refer to Synchronizing a Java thread identity and an operating system thread identity and Understanding Connection Manager RunAs Identity Enabled and operating system security for more information.

Perform the following steps to enable the thread identity function for the connection factories or JDBC provider data sources created with the supported JCA resource adapters and JDBC providers:

1. Define **resauth=Container** for the application resource. See the “Connection thread identity” on page 1035 article for details.
2. Ensure the JCA resource adapters or JDBC providers support the thread identity function.

   Review the supported resource adapters and data source providers, and the level of support: REQUIRED, ALLOWED, and NOTALLOWED. The article “Connection thread identity” on page 1035 contains a table of the JCA resource adapter processes and the JDBC provider processes that support thread identity and thread security.

   If the adapter or provider is not listed, then thread identity support is NOTALLOWED, by default.
3. Set the **Container-managed authentication alias** to NULL, if you configure the connector locally.

   When the connector is configured locally, the resource adapter determines the level of thread identity support as **ALLOWED**. If thread identity support is allowed and you specify **Container-managed authentication alias** as NULL, the connector uses the current thread identity as the owner for each connection that is created.

   When the resource adapter or JDBC provider determines that the level of thread identity support is **REQUIRED**, any specification for the **Container-managed authentication alias** is ignored. Thread identity support in this case always applies.
4. Determine connector behavior when Java 2 security is a factor. See the article “Security states with thread identity support” for more information.

   If you want the thread identity associated with a connection to be the thread identity, then you must enable Java 2 security. In the case of JDBC providers that support the thread identity function and require the thread to be pushed to the z/OS thread of execution, you must set the server **Connection Manager RunAs Identity Enabled** property to **true**.

   Note: With Bean Managed Persistence (BMP) beans, if you obtain a connection under the `ejbLoad()` or `ejbStore()` functions during pre-invoke or post-invoke method processing, your thread identity support does not become the **RunAs** identity because the container during this processing is running under server identity. With BMP beans, instead of using thread identity, specify a Container-managed alias to associate the user with the connection.
5. Set the `security.zOS.session.OMVSSRV` custom property to `true`. When the thread identity support is used, a security credential that is based on the current thread identity encapsulates the security information for the user that is associated with the connection. By default, the session type associated with the user is TSO. If you have WebSphere Application Server for z/OS users that use the thread identity support, you must define the users as TSO users. If you prefer not to define the users as TSO users, you can use the security.zOS.session.OMVSSRV custom property, which changes the session type for the user identity in the security credential from TSO to OMVSSRV. However, if you use the user information for authentication at the target EIS, such as IMS, the user must be an authorized OMVSSRV user.

   To specify the custom property, complete the following steps:

   a. Click **Security** → **Global security** → **Custom Properties**.
   b. Click **New**.
   c. In the Name field, type `security.zOS.session.OMVSSRV`.

      Note: This custom property name is case sensitive.
   d. In the value field, type `true`.
   e. Click **Apply** and **Save**.

Security states with thread identity support

Different JCA resource adapters and JDBC drivers provide different support for authenticating threads that transact with application server resources.

In this article the term thread identity refers to the Java Platform, Enterprise Edition (Java EE) Identity, such as the RunAs Identity, as opposed to the OS thread identity. Refer to Synchronizing a Java thread identity and an operating system thread identity and Understanding Connection Manager RunAs Identity Enabled and operating system security for more information.

The combinations of Java 2 security, server configurations, connector configurations, and container-managed alias support determine the processing that results when you use the thread identity function. Thread identity support is only available with specific JCA resource adapters and JDBC providers. See the article “Connection thread identity” on page 1035 for a table of resource adapter processes and JDBC provider processes that support thread identity. If your resource adapter or JDBC provider is in the supported list, use the following tables to determine the processing that occurs, based on the settings of the specified properties:

Table 24. Table 1. Security state

Global security enabled?	
Yes	No
Go to table 2.	Go to table 3.

Table 25. Table 2. Global security enabled

<table>
<tr><td colspan="8">Container-managed alias specified?</td></tr>
<tr><td colspan="4">No</td><td colspan="4">Yes</td></tr>
<tr><td colspan="4">Connector Allows or Requires Thread Identity?</td><td colspan="4">Connector Requires Thread Identity?</td></tr>
<tr><td>No</td><td colspan="3">Yes</td><td>No</td><td colspan="3">Yes</td></tr>
<tr><td rowspan="4">Processing is dependent on connector: may throw exception may default to connector user/ password custom properties</td><td colspan="3">Connector requires OS thread security?</td><td rowspan="4">Use specified alias</td><td colspan="3">Connector requires OS thread security?</td></tr>
<tr><td>No</td><td colspan="2">Yes</td><td>No</td><td colspan="2">Yes</td></tr>
<tr><td rowspan="2">Use identity associated with current thread</td><td colspan="2">Server Sync-To-Thread enabled?</td><td rowspan="2">Use identity associated with current thread</td><td colspan="2">Server Sync-To-Thread enabled?</td></tr>
<tr><td>No</td><td>Yes</td><td>No</td><td>Yes</td></tr>
<tr><td>Use Server identity</td><td>Use identity associated with current thread</td><td>Use server identity</td><td>Use identity associated with current thread</td></tr>
</table>

Table 26. Table 3. Global Security is not enabled

<table>
<tr><td colspan="4">Container-managed alias specified?</td></tr>
<tr><td colspan="2">No</td><td colspan="2">Yes</td></tr>
<tr><td colspan="2">Connector ALLOWS or REQUIRES thread identity o be used when getting a connection</td><td colspan="2">Connector REQUIRES thread identity to be used when getting a connection?</td></tr>
<tr><td>No</td><td>Yes</td><td>No</td><td>Yes</td></tr>
<tr><td>Processing is dependent on connector:
• May throw exception
• May default to connector user/password custom properties</td><td>User server identity</td><td>Use specified alias</td><td>Use server identity</td></tr>
</table>

Establishing custom finder SQL dynamic enhancement server-wide

Enable support for dynamic SQL enhancement of all custom finders, defined in all beans, by modifying the custom properties of your application server in the administrative console.

About this task

To establish this support on a server-wide basis (that is, dynamic SQL enhancement of all custom finders defined in all beans is enabled), use the following steps.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Control** or **Servant**. Select **Control** to define the property in the Control, **Servant** to define the property in the Servant.
7. In the Additional Properties area, select **Java Virtual Machine**.
8. Select **Custom Properties**.
9. Select **com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent** and enter a value of **all**. If the property is not present in the list, create a new property name, enter the name *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* and the value **all**.

Establishing custom finder SQL dynamic enhancement on a set of beans

You can enable support for all custom finders defined on beans by modifying your application server's custom properties through the administrative console.

About this task

To establish this support for all custom finders defined on a set of beans use the following steps.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. Select **Server Infrastructure** → **Java and Process Management** → **Process Definition**.
6. Select **Control** to define the property, or select**Servant** to define the property in the Servant.
7. Select **Java Virtual Machine**.
8. Select **Custom Properties**.
9. Select **com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent** and enter a value that corresponds to a list of beans that need this support, with each bean's name separated from the others by a colon (:). For example, *beanA:beanB:beanC*.

   If the property is not present in the list, create a new property name, enter the name *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* and enter the list as the value.

Establishing custom finder SQL dynamic enhancement for specific custom finders

You can add an environment variable to establish support for custom finders you want to use in data access applications.

About this task

To establish this support for specific custom finders use the following steps.

1. Start a Java Platform, Enterprise Edition (Java EE) application development environment of your choice.

2. Create or edit the application EAR file needing this support.
3. Check for an environmental variable called *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel* . If the variable does not already exist, add it to the EAR file.
4. Give the variable a value that corresponds to a list of method names (including parameter lists) with each name separated from the others by a colon (:).
5. Deploy and install the application.

Disabling custom finder SQL dynamic enhancement for custom finders on a specific bean

You can disable support for all custom finders defined on a specific bean by modifying your application's EAR file.

About this task

To disable this support for all custom finders defined on a specific bean, assuming that the server-wide support is enabled, follow these steps.

1. Start a Java Platform, Enterprise Edition (Java EE) application development environment of your choice.
2. Create or edit the application EAR file needing this support.
3. Check for an environmental variable called *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* with a value of **true**. If the variable does not already exist, add it to the EAR file.
4. Ensure that the server-wide setting *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* is in place on the target server.
5. Deploy and install the application.

Deploying Structured Query Language in Java (SQLJ) applications

Use Structured Query Language in Java (SQLJ) to develop data access applications that connect to DB2 databases. SQLJ is a set of programming extensions that enable you to use the Java programming language to embed statements that provide SQL (Structured Query Language) database requests.

About this task

The advantages of developing applications with SQLJ include improved performance and a shorter, more efficient development cycle. With SQLJ you can:

- Improve performance by using static SQL statements.
- Reduce the development cycle:
  - Write less code with the simpler SQLJ syntax, which reduces the number of lines of code that is required to execute statements, set parameters, and retrieve parameters.
  - Detect programming errors earlier in the development phase with the onlinecheck function, which performs data type validation and schema validation. See the DB2 documentation for a complete list of customization options.

Consider using SQLJ in situations where dynamic SQL is not needed, and where applications use DB2 as the database server.

Note: The application server includes enhanced SQLJ support for applications that use container-managed persistence (CMP). The new features include:

- Deploying CMP beans during the application installation in the application server.
- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without needing to reinstall the application.

These enhancements reduce the complexity of installing, deploying, and customizing SQLJ applications for both container-managed and bean-managed persistence.

1. Acquire the required drivers to deploy an SQLJ application in the application server. You need the following files, depending on the JDBC provider that you use:

JDBC provider type	Required files
DB2 Using IBM JCC Driver This driver is also known as: • IBM Data Server Driver for JDBC and SQLJ • IBM DB2 Driver for JDBC and SQLJ • IBM DB2 Universal JDBC Driver.	db2jcc.jar or db2jcc4.jar
DB2 Universal JDBC driver (deprecated)	db2jcc.jar

2. Deploy the SQLJ application.
   - Deploy applications that use container-managed persistence (CMP):
     - "Deploying SQLJ applications that use container-managed persistence (CMP)" with the DB2 Using IBM JCC Driver.
     - "Deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool" on page 1042.
   - "Deploying SQLJ applications that use bean-managed persistence, servlets, or sessions beans" on page 1043.
   - "Using embedded Structured Query Language in Java (SQLJ) with the DB2 for z/OS Legacy driver" on page 1053 (deprecated).
3. Customize and bind the SQLJ profiles. Before the application server can use an SQLJ application, the SQLJ statements must be processed for the database server. By default, four DB2 packages are created in the database; one package is created for each isolation level. The customization process augments the profiles with information that is specific to the database. If you do not customize the SQLJ profiles, the SQLJ application will use dynamic SQL like a JDBC application.
   - "Customizing and binding profiles for Structured Query Language in Java (SQLJ) applications" on page 1045.
   - Customizing and binding SQLJ profiles with the wsadmin tool.
   - "Customizing and binding SQLJ profiles with the db2sqljcustomize tool" on page 1047.

Deploying SQLJ applications that use container-managed persistence (CMP)

Embed Structured Query Language in Java (SQLJ) statements in your applications to maximize the efficiency of transactions with your databases. Before your applications can take advantage of SQLJ, you must deploy the application and customize the SQLJ profiles that are created. The application server provides functionality to use SQLJ as the persistence mechanism for enterprise beans that use container-managed persistence. Deploy the CMP beans in the application server to enable SQLJ support.

Before you begin

You need an application that uses SQLJ and container-managed persistence. Develop this application in Rational Application Developer or another development tool.

About this task

Deploy SQLJ applications in the application server to simplify the process of SQLJ translation and bean deployment. The application server includes these new features for SQLJ support:

- Deploying CMP beans during the application installation in the application server.
- Customizing and binding SQLJ profiles with the administrative console or scripting.

- Customizing and binding SQLJ applications again without needing to reinstall the application.

You can also deploy the SQLJ application using the ejbdeploytool. Read the topic on deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool for more information.

1. Create a top-down mapping to a DB2 database.
2. From your DB2 installation, copy the sqlj.zip file to a directory on your workstation.
3. Deploy the EAR file in the administrative console.
   a. Click **Applications** → **Install New application**.
   b. Select **Local file system** or **Remote file system**, and browse to the EAR file.
   c. Select **Detailed - Show all installation options and parameters**. Click **Next**.
   d. In **Step 1: Select installation options**, select **Deploy enterprise beans**. Configure any other options, and click **Next**.
   e. In **Step 3: Provide options to perform the EJB deploy**, select **SQLJ** for **Deploy EJB option - Database access type**.
   f. Enter the location of the sqlj.zip file in the **SQLj class path** field.
   g. Complete the installation process for the application.

What to do next

After the enterprise application is deployed, customize the SQLJ profiles using the administrative console, scripting, or the db2sqljcustomize tool:

- For administrative console support, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
- For scripting support, read the topic on the application management command group for the AdminTask object.
- For use of the db2sqljcustomize tool, read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool.

Deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool:

Embed Structured Query Language in Java (SQLJ) statements in your applications to maximize the efficiency of transactions with your databases. Before your applications can take advantage of SQLJ, you must deploy the application and customize the SQLJ profiles that are created. The application server provides functionality to use SQLJ as the persistence mechanism for enterprise beans that use container-managed persistence. Use the ejbdeploy tool to deploy the application.

About this task

You can deploy SQLJ applications with the ejbdeploy tool to deploy the enterprise application in a standalone environment.

Alternatively, the application server includes enhanced SQLJ support for applications that use container-managed persistence (CMP). The new features include:

- Deploying CMP beans during the application installation in the application server.
- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without needing to reinstall the application.

These enhancements reduce the complexity of installing, deploying, and customizing SQLJ applications for both container-managed and bean-managed persistence.Read the topic on deploying SQLJ applications that use container-managed persistence (CMP) for more information.

1. Create a top-down mapping to a DB2 database.

2. From your DB2 installation, copy the sqlj.zip file to a directory on your workstation.
3. Modify the Java build path of your enterprise bean JAR project to include the sqlj.zip file.
4. Use Rational Application Developer or the DB2 SQLJ translator to automatically translate SQLJ.
   - Use Rational Application Developer:
     a. From the Project Navigator, click ***EJB_JAR_PROJECT_NAME*** → ***SOURCE_FOLDER*** → **META-INF** → **backends** → ***database_version***.
     b. Open Map.mapxmi in the Mapping editor.
     c. On the **Overview** panel, highlight the name of your JAR project in the Enterprise Beans column. You must highlight the name of the JAR project, not the name of one of the enterprise beans that is listed.
     d. On the **Properties** panel, expand **SQLJ**.
     e. Set **Is using SQLJ?** to `True`.
     f. Set **Translator Module** to the fully qualified path of the sqlj.zip file on your workstation.
     g. Save the Map.mapxmi file.
     h. Export the enterprise archive (EAR) file.
   - Use the DB2 SQLJ translator. This tool creates a .java version of your .sqlj file and a serialized profile, with a .ser extension, that is used later in processing. Refer to the DB2 documentation for more information on the SQLJ translator tool.
5. Deploy the EAR file with the ejbdeploy tool.
   a. Verify that the app_server_root/bin directory is in your class path.
   b. Run the ejbdeploy command utility with the `-sqlj` option. The ejbdeploy command will generate an EAR file with the name you specify and an Ant script with the name *application_name*.ear.xml.

      For example: :

      ```
      ejbdeploy d:\application_name.ear
                working d:\deployed_application_name.ear
                -sqlj
                -dbvendor DB2UDB_V81
                -cp "C:\PROGRA~1\IBM\SQLLIB\java\sqlj.zip"
      ```

      Note: Supply the location of the SQLJ translator sqlj.zip file with -cp, which is the class path option The ejbdeploy command does not access sqlj.zip from your system class path.
6. Choose the option for customization.
   - Use the application server's SQLJ support. Install the deployed application to customize the SQLJ profiles with the application server or scripting.
     a. Install the enterprise application in the application server.

        Note: Do not select **Deploy enterprise beans** during the application installation process in the administrative console. If you redeploy the enterprise beans from the administrative console, you will lose the customization changes that you have made.
     b. Customize the SQLJ profiles.
        – For administrative console support, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
        – For scripting support, read the topic on the application management command group for the AdminTask object.
   - Customize and bind the SQLJ profiles with the db2sqljcustomize tool. Read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool.

Deploying SQLJ applications that use bean-managed persistence, servlets, or sessions beans

Embed Structured Query Language in Java (SQLJ) statements in your applications to maximize the efficiency of transactions with your databases. Before your applications can take advantage of SQLJ, you

must deploy the application and customize the SQLJ profiles that are created. You can use Rational Application Developer (RAD) or the DB2 SQLJ translator to translate the application before you deploy them on the application server.

Before you begin

Create an SQLJ application using Rational Application Developer or another development tool.

About this task

To deploy SQLJ applications that do not use container-managed persistence, you need to translate the SQLJ application first to configure it for the application server environment. After translation, you can choose to customize the SQLJ profiles in the application server, with scripting, or with the db2sqljcustomizer tool.

Note: The application server includes these new features for SQLJ support for applications that use bean-managed persistence:
- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without needing to reinstall the application.

1. Optional: Create a backup copy of your .java file. For example if your file is called MyServlet.java, copy MyServlet.java to MyServlet.java.bkup.
2. Optional: Rename your .java file to a file name with a .sqlj extension. For example, if your application is a servlet named MyServlet.java, rename MyServlet.java to MyServlet.sqlj
3. Optional: Edit the .sqlj file to convert the JDBC syntax to SQLJ syntax. When using SQLJ, if you want connection management for the application server to function properly, you must specify correct connection contexts.

   For example, convert the following JDBC operation:

   ```
   Connection con = dataSource.getConnection();
   Statement stmt = con.createStatement();
   stmt.execute("INSERT INTO users VALUES (1, 'user1')");
   con.commit();
   ```

   to the following SQLJ:

   ```
   // At the top of the file and just below the import statements, define Connection_Context
   #sql context Connection_context;
   .
   .
   Connection con = dataSource.getConnection();
   .
   .
   Connection_context ctx1 = new Connection_context(con);
   .
   .
   #sql [ctx1] {INSERT INTO users VALUES (1, 'user1')};
   .
   .
   con.commit(); ctx1.close();
   ```

   When you run the SQLJ translator, the .java file that the tool creates will have the same name as your old .java file, providing you with a seamless transition to the SQLJ technology.
4. From your DB2 installation, copy the sqlj.zip file to a directory on your workstation. Modify the Java build path of your enterprise bean Java archive (JAR) file project to include the sqlj.zip file.
5. Use Rational Application Developer or the DB2 SQLJ translator to automatically translate SQLJ.
   - Use Rational Application Developer:
     a. In the Project Navigator, right-click your JAR project, and select **Add SQLJ Support...**.
     b. Select the boxes for the applications for which you want SQLJ support.

c. In the **SQLJ JAR file** field, type the fully qualified path to the sqlj.zip file that you previously copied to your workstation.
d. Click **Finish**.
e. Export the enterprise archive (EAR) file.

- Use the DB2 SQLJ translator. This tool creates a .java version of your .sqlj file and a serialized profile, with a .ser extension, that is used later in processing. Refer to the DB2 documentation for more information on the SQLJ translator tool.

6. Package your JAR file for the enterprise application.
7. Install the application onto the application server, or customize the profiles with the db2sqljcustomize tool.
   - Customize the profiles with the application server.
     a. Package the JAR file for your enterprise beans, servlets, and any .ser files into an enterprise archive.
     b. Install the application in the application server, and customize SQLJ profiles with the administrative console or the wsadmin tool.

        Note: Do not select **Deploy enterprise beans** during the application installation process in the administrative console. If you redeploy the enterprise beans from the administrative console, you will lose the customization changes that you have made.

        The application server provides enhanced support for SQLJ applications. Install the SQLJ application in the application server , and you can customize and bind SQLJ profiles through the administrative console or scripting:
        - To customize the SQLJ profiles with the administrative console, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
        - To customize SQLJ profiles with scripting, read the topic on the application management command group for the AdminTask object.
   - To use the db2sqljcustomize tool, read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool for more information.

Customizing and binding profiles for Structured Query Language in Java (SQLJ) applications

Simplify the process of customizing and binding SQLJ profiles for your applications by performing these functions in the administrative console or with scripting. SQLJ profiles must be customized and bound before the enterprise application can use the application's embedded SQL.

Before you begin

You must have an SQLJ application that has already been deployed and installed in the application server.

For SQLJ applications that use container-managed persistence, you can deploy the application in two ways:

- Deploy the SQLJ application in the application server. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) for more information.
- Deploy SQLJ applications with the ejbdeploy tool. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool.

For SQLJ application that use bean-managed persistence, see the topic on deploying SQLJ applications that use bean-managed persistence, servlets, or session beans.

About this task

To take advantage of SQLJ applications in the application server, you need to customizing the SQLJ profiles that contain the embedded SQL statements. By default, four DB2 packages are created in the

database; one for each isolation level. The customization process augments the profiles with information that is specific to the DB2 database. The database uses this information at run time.

In addition to profile customization, you need to bind the customized profiles to the DB2 database. Profile binding should only take place after the SQLJ profiles are customized.

You can also customize and bind profiles with scripting or the db2sqljcustomize tool:

- For scripting support, read the topic on the application management command group for the AdminTask object.
- For information on the db2sqljcustomize tool, read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool for more information. If you customize profiles with the db2sqljcustomize tool, you will need to reinstall the application.

1. Make sure the necessary database tables exist, as described in the topic on deploying data access applications.
2. Navigate to the SQLJ application that is installed in the application server. Click **Applications** → **Websphere enterprise applications** → ***app_name***.

   Note: Do not run multiple sessions of the administrative console to customize and bind profiles that are in the same EAR file.
3. Navigate to the SQLJ profiles section. Click **SQLj profiles**. When you click this link, the application server expands the EAR file for the application into a temporary directory; there might be a delay before the panel for SQLJ profiles is displayed.
4. Select **Customize and bind profiles** or **Bind packages**. Choose your option based on the profiles with which you are working:
   - If your profiles have not been customized, or you want to customize the profiles again, choose **Customize and bind profiles**.
   - If the profiles are already customized, choose **Bind packages**.
5. Choose to select profiles or a profile group to customize and bind.
   - Select profiles from the list that is provided.
     a. Select the profiles from the list and click **Add**. The list displays the SQLJ profiles that are present in the enterprise application.

        Note:
        – Select more than one profile by holding CTRL.
        – Select a contiguous list of profiles by selecting the first profile name, holding SHIFT, and selecting the last profile. You will select the first profile, last profile, and any profiles in the middle.

     b. Select **Customize/bind the selected SQLj profiles as a group** This option specifies that the application server will create a .grp file that contains the SQLj profiles that are processed. You can use the .grp file for other binding operations in the future. After you have completed this panel and click **OK**, you will be given an option to download the .grp file.
   - Select **Use a profile group file to specify profiles to customize/bind**. Select this to specify a profile group to process. Click **Browse...** to locate the file on the system.
6. Complete the necessary information to connect to the database. You need to complete the following fields:

   Database URL

   Specifies the URL of the database to which the profile/s will be bound. The typical syntax is:

   `jdbc:db2://<host name="">:<port>/<database name="">.</database></port></host> or`

   or

   `fully_qualified_host_name:port`

User Specifies the user ID for the database administrator on the server where the database is located.

Password
Specifies the password for the database administrator on the server where the database is located.

Additional options
Specifies additional options to use during the customization and bind processes. See the DB2 documentation for a complete list of customization options.

Class path
Specifies the class path where sqlj.zip, and db2jcc.jar or db2jcc4.jar are located.

7. Click **OK**.

Note: If you are processing a large enterprise application, or you are processing many SQLJ profiles, the process might take longer than the default timeout for the administrative console. The default connection timeout for the application server's administrative console is set to 30 minutes. If the default timeout is reached and you lose the connection to the server, you can check the system output log for the final results of the customization and bind process.

To prevent this disconnection, configure the console session timeout to a longer period of time. After a successful customization and binding process, check the system output log for the total processing time. Use that time period as a basis for the new timeout value. For information about how to configure the console timeout, see the topic on changing the console session expiration.

Results

After the application server finishes processing the SQLJ profiles, you will see the results from the customization and binding. The results panel displays messages from the database server, as well as summary results from the application server.

If the operation completed successfully, the following message will be printed to the system log:

```
ADMA0507I=ADMA0507I: The SQLJ operation on application {0} completed successfully.  Exit code: {1}
ADMA0507I.explanation=This informational message indicates the program status.
ADMA0507I.useraction=No user action is required.
```

If the operation did not complete successfully, the following message will be printed to the system out log:

```
ADMA0506I=ADMA0506I: The SQLJ operation on application {0} did not complete successfully.  Exit code: {1}
ADMA0506I.explanation=The SQLJ operation encountered a problem.  This informational message indicates
the program status. Prior messages in the command output give details of the problem.
ADMA0506I.useraction=Check the command output for the cause of the problem.
```

Customizing and binding SQLJ profiles with the db2sqljcustomize tool:

Customize and bind SQLJ profiles with the db2sqljcustomize tool before you install the SQLJ application in the application server.

Before you begin

To perform this task, you must have SQLJ application that has been deployed, but the application should not be installed in the application server. If the application is already installed in the application server, you will need to reinstall the application after you customize the profiles. You also need serialized profiles for the SQLJ application.

For SQLJ applications that use container-managed persistence, you can deploy the application in two ways:

- Deploy the SQLJ application in the application server. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) for more information.
- Deploy SQLJ applications with the ejbdeploy tool. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool.

For SQLJ application that use bean-managed persistence, see the topic on deploying SQLJ applications that use bean-managed persistence, servlets, or sessions beans.

About this task

To take advantage of SQLJ applications in the application server, you need to customize the SQLJ profiles. The customization process augments the profiles with information that is specific to the DB2 database. The database uses this information at run time. By default, four DB2 packages are created in the database; one package is created for each isolation level.

Note: The application server supports customizing and binding the SQLJ profiles in the administrative console or with scripting:
- For administrative console support, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
- For scripting support, see the topic on the application management command group for the AdminTask object.

1. Make sure the necessary database tables exist, as described in the topic on deploying data access applications.
2. Transfer the serialized profiles to the environment on which you installed your application. Alternatively, use the Java `jar` command to extract the serialized profiles from the JAR file in your installed EAR directory.
3. Add the location for the SQLJ profiles and the application's JAR file to your environment's class path.
4. Make sure the necessary database tables exist, as described in the topic on deploying data access applications.
5. Optional: If your application is not running in a clustered environment, you can use the Ant script to make customization easier. If you run a batch SQLJ customization against an EAR file with the ejbdeploy tool, the tool produces an Ant script that is named *application_name*.ear.xml. You can use this script file to run the DB2 customizer program against the serialized profiles in all of the enterprise bean JAR files for the associated EAR file. The script updates each enterprise bean's JAR file with a serialized profile and replaces the JAR files in the existing EAR file with the modified versions.
   a. Change the values of the database URL, and the database user and password properties in ejbdeploy.sqlj.properties. This file is a common file to all Ant scripts that are generated by the ejbdeploy command. The ejbdeploy.sqlj.properties script defines the global properties for:
      - Database URL - db.url
      - User - db.user
      - Password - db.password

      The Ant script uses the URL, user, and password properties in the serialized profile to customize the profile. By default, the properties for the serialized profile are created from the global properties.
   b. Run the Ant script, specifying the `properties` target. For example:

      ```
      ws_ant -buildfile application_name.ear.xml properties
      ```

      This script creates the properties file, *application_name*.ear.properties. The *application_name*.ear.properties file contains properties that specify the default names for the packages corresponding to each serialized profile in the EAR file. This is a sample properties file:

      ```
      url.MyEJB1.jar.DB2UDBNT_V8_1=jdbc:db2://localhost:50000/MyDB1
      user.MyEJB1.jar.DB2UDBNT_V8_1=dbuser
      password.MyEJB1.jar.DB2UDBNT_V8_1=dbpassword
      pkg.MyEJB1.jar.DB2UDBNT_V8_1=TEST
      ```

```
url.MyEJB2.jar.DB2UDBNT_V8_1=jdbc:db2://localhost:50000/MyDB2
user.MyEJB2.jar.DB2UDBNT_V8_1=dbuser
password.MyEJB2.jar.DB2UDBNT_V8_1=dbpassword
pkg.MyEJB2.jar.DB2UDBNT_V8_1=WORK
```

c. Use the DB2 Control Center to identify the packages that are installed in the database. The DB2 SQLJ customizer requires a type 4 database URL in the form of:

```
jdbc:db2://host-name:port/database-name
```

It also requires a user and password. The value of the port is 50000, unless you change it when you install DB2.

d. Change the names that are used by the script file to ensure that the names for each customization profile do not conflict with existing package names that are in the database. Ant scripts that are generated for different EAR files use the same package names by default, and the script will overwrite existing packages unless you change the names. Overwritten packages can cause errors at run time.

DB2 uses the first seven characters of the package name. The DB2 customizer uses this name to create four packages in the database. For example, if you specify the name TEST, the DB2 customizer will create packages called TEST1, TEST2, TEST3, and TEST4.

e. Run the Ant script. The Ant script updates the original EAR file with the modified serialized profiles.

Note: Verify that you have db2jcc.jar in the class path. This file should have been added to the class path environment variable when DB2 V8 FixPak1 was installed.

A sample Ant command looks like this:

```
ws_ant -Dwork.dir=tmp
       -Dscript.property.file=other.properties
       -buildfile application_name.ear.xml
```

where:

- `-buildfile` specifies the XML file to create.
- `-Dscript.property.file` specifies a different properties file. This parameter is optional. If you want your Ant script to use a another file instead of *application_name*.ear.properties, specify the Dscript.property.file property when you run the script.
- `-Dwork.dir` specifies a temporary working directory for the script. The script will create and delete files and subdirectories in this directory. If the working directory contains existing files and directories with the same name as the files and directories used by the script, the script will erase or overwrite the files and directories. This script creates and uses a directory called `tmp` as its working directory.

f. Proceed to installing the application in the application server..

6. Run the db2sqljcustomize tool to customize the SQLJ profiles that correspond to each enterprise bean's JAR file. When you generate your deployment code, serialized profiles (files with a .ser extension) that are specific to your application are created. These profiles exist in the same directory as your SQLJ files, and the files must be customized to the environment before they can be used. When you run the DB2 SQLJ customizer against the serialized profiles, you create static SQL in the database that DB2 will use at run time. The customization phase creates four database packages that contain static SQL, one for each isolation level.

a. Optional: Consider using the SQLJ customizer tool to enable context caching for your application's data source connections. DB2 V8.1 fix pack 6 provides the new caching option with the db2sqljcustomize tool called db2optimize. You can run this option if your application uses the explicit connection context instead of the default context.

Note:

- SQLJ context caching support requires the DB2 with IBM JCC driver or Version 2.2 or later of the DB2 Universal JDBC Driver with APAR PQ87786 applied.

- If you want to enable context caching for an application or BMP bean that caches connections across transaction boundaries, you cannot use shareable connections. Use the get/use/close pattern of connection usage when you invoke the db2optimize option, or an object closed exception occurs. The following code gives an example of incorrect connection usage for context caching:

```
utx.begin();
     cons =ds.getConnection(
     request.getParameter("db.user"),
     request.getParameter("db.password"));
         cmctx1 = new CM_context(cons);
         #sql [cmctx1] {DELETE FROM cmtest WHERE id=1};
 utx.commit();
       //The next statement verifies the result:
         #sql [cmctx1] cursor1 = {SELECT id, name FROM cmtest WHERE id=1};
```

  In this case, the `Select` statement elicits an object closed exception. To prevent the exception from occurring, close the connection before committing the transaction. Then get a new connection and a new context before running the `Select` statement.

The following example code demonstrates proper syntax for running the option on the serialized profile:

```
sqlj -db2optimize SQLJTransactionTest.sqlj
db2sqljcustomize -url jdbc:db2://localhost:50000/dbname -user USER_NAME -password PASSWORD
SQLJTransactionTest_SJProfile0.ser
```

b. Run the db2sqljcustomize tool to customize the SQLJ profiles. After you successfully run the db2sqljcustomize command, customized profiles exist in the directory from which you issued the command. If you run the db2sqljcustomize command from the directory that contains the serialized profiles that were not customized, the customized versions will overwrite previous versions that have the same file names.

   The recommended syntax for running the db2sqljcustomize command is:

```
db2sqljcustomize -url JDBC_URL -user USER_NAME  -password PASSWORD
[-rootpkgname PACKAGE_NAME] SERIALIZED_PROFILE1 SERIALIZED_PROFILE2 ...
```

   where:

   - *JDBC_URL* is the JDBC URL that is used to access the DB2 system where your tables reside.
   - *USER_NAME* is a valid user name for the DB2 system where your tables reside.
   - *PASSWORD* is the password for the specified user name.
   - *PACKAGE_NAME* is a valid partitioned data set (PDS) member name, up to seven characters long. Each of the four packages that are created by the profile customizer begin with this name and are appended with a number from 1 to 4. If you customize only one serialized profile, this value defaults to a shortened version of the serialized profile name and the -rootpkgname parameter is not required. If you customize more than one serialized profile with the same command, there is no default value and the -rootpkgname parameter is required.
   - *SERIALIZED_PROFILE#* is the name of the serialized profile that you are customizing.
     – To customize more than one serialized profile with the same command, list multiple files, separated by spaces.
     – Alternatively, you can specify the `-rootpkgname` parameter to customize more than one serialized profile with the same command.

   Note: The following options provide more control over the customization process:

   - `-automaticbind yes` specifies to run the DB2 SQLJ customizer against the serialized profiles to create static SQL in the database that the database will use at run time. The customization phase creates four database packages that contain static SQL, one for each isolation level.

- `-onlinecheck NO and -bindoptions "VALIDATE RUN"` specifies settings to bypass errors during a profile customization and ensure a successful customization.

7. Update the JAR file for the enterprise beans with the serialized profiles.
8. Use the `jar` command to replace the serialized profiles in your JAR file with the customized profiles.

   Note: The customized files must be placed in a location that is part of the application class path, and they must exist ahead of the serialized profiles that are not customized in your JAR file. If you decide to replace the serialized profiles in your JAR file, maintain the directory structure in which the profiles exist.
9. Package the JAR file for the enterprise bean, servlets, and serialized profiles into an enterprise archive (EAR) file.
10. Install the application in the application server.

    Note: Do not select **Deploy enterprise beans** during the application installation process in the administrative console. If you redeploy the enterprise beans from the administrative console, you will lose the customization changes that you have made.

SQLj profiles and pureQuery bind files settings:

Use this panel to specify customization and binding settings for the Structured Query Language in Java (SQLJ) profiles and pureQuery bind files for DB2 that are included in this application. You can view SQLJ profiles for other database types, but you cannot change the these profiles. PureQuerybind files are only valid for DB2. Use SQLJ or pureQuery to develop data access applications that connect to DB2 databases. SQLJ is a set of programming extensions that enable a programmer to use the Java programming language to embed statements that provide SQL (Structured Query Language) database requests. PureQuery provides an alternate set of APIs that can be used instead of JDBC to access the DB2 database.

To view this administrative console page, click **Applications** → **Websphere enterprise applications** → ***app_name*** → **SQLj profiles and pureQuery bind files**.

Advantages of developing applications with SQLJ include improved performance and a shorter, more efficient development cycle. With SQLJ, you can:
- Improve performance by using static SQL statements.
- Reduce the development cycle by:
  - Writing less code with the simpler SQLJ syntax, which reduces the amount of code that is required to execute statements, and set and retrieve parameters.
  - Detecting programming errors earlier in the development phase with the online check function, which performs data type and schema validation. Activate this function by running it as an option with the **db2sqljcustomize** command. See the DB2 documentation for a complete description of the SQLJ customize command.

DB2 pureQuery runtime is an alternative set of apis to JDBC or SQLj. Advantages of developing applications with pureQuery include allowing SQL execution to be either dynamic or static. In addition to improved performance by using static SQL statements, pureQuery has better problem determination and diagnosis because it allows for errors at the DB2 server to be related back to application artifacts rather than to SQL that was generated by an application generator.

Customize and bind profiles:

Specifies that the application server will process the SQLJ profiles that you select from this application.

Note: This selection does not apply to pureQuery. If selected, this option will be ignored when processing pureQuery bind files.

By default, one DB2 package is created in the database for each isolation level. The customization process augments the profile or profiles with information that is specific for the DB2 database for use at run time. Typically, you should run the customization process after the SQLJ application has been translated and before the application is started. If you do not run the customization step, the SQLJ application will use dynamic SQL like a JDBC application.

Binding DB2 SQLj profiles involves the process of binding the customized SQLj profiles to the DB2 database.

Bind packages:

Specifies that the application server will bind the SQLJ profiles that you select to the DB2 database server.

Note: This selection does not apply to pureQuery. If selected, this option will be ignored when processing pureQuery bind files.

Bind packages from the SQLJ application that have already been customized.

Select and order the profiles to customize/bind:

Specifies the profiles to process from the list that is provided.

- Select a profile or group of profiles from the **Available profiles**, and click **Add** to add the profile that is selected to **Selected Profiles**.
- Select a profile or group of profiles from the **Selected Profiles**, and click **Remove** to add the profile that is selected to **Available profiles**.

When SQLJ or pureQuery profiles have been added to **Selected Profiles**, select profiles from that list and use **Move Up** or **Move Down** to change the order in which the profiles will be processed.

Customize/bind the selected SQLj profiles as a group:

Specifies that the application server will create a .grp file that contains the SQLJ profiles that you selected.

Note: This selection does not apply to pureQuery. If selected, this option will be ignored when processing pureQuery bind files.

When you click **OK**, there is an option on the next panel to download the .grp file.

Use a profile group file to specify profiles to customize/bind:

Specifies a profile group file from the local file system to customize or bind.

Database URL:

Specifies the URL of the database to which the profile or profiles will be bound.

The typical syntax is:

```
jdbc:db2://host_name:port_name/database_name
```

User:

Specifies the user ID for the database administrator on the server where the database is located.

Password:

Specifies the password for the database administrator on the server where the database is located.

Additional options:

Specifies additional options to use during the customization and bind processes.

Options for pureQuery binding uses the following syntax:

```
-bindoptions "BLOCKING NO"
```

For more information on pureQuery bind options, refer to the DB2 pureQuery Bind Utility topic.

Class path:

Specifies the class path where the sqlj.zip, and db2jcc.jar or db2jcc4.jar files for SQLj are located. Specifies the class path where the pdq.jar, pdqmgt.jar, db2jcc.jar, db2jcc_license_cisuz.jar files for pureQuery are located.

Download SQLj profile group:

Use this panel to download the group file for the Structured Query Language in Java (SQLJ) profiles that will be bound together as a single package on the DB2 database server. You can use the file when performing future customization or binding work on the application. Click the link that is provided to download the profile group to your local file system.

Note: This topic does not apply to pureQuery. PureQuery does not support binding pureQuery bind files as a group.

Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***app_name*** → **SQLj profiles**. When you are selecting the profiles to customize and bind, select **Customize/bind the selected SQLj profiles as a group** to view this console panel.

Review results:

Use this panel to review the results from the customization and binding process for the Structured Query Language in Java (SQLJ) profiles or pureQuery bind files. Use SQLJ or pureQuery to develop data access applications that connect to DB2 databases. SQLJ is a set of programming extensions that enable a programmer to use the Java programming language to embed statements that provide SQL (Structured Query Language) database requests. PureQuery provides an alternate set of APIs that can be used instead of JDBC to access the DB2 database.

Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **SQLj profiles and pureQuery bind**. Select profiles to customize and bind, complete the necessary fields, and click **OK** to view this console panel.

Review results:

Displays the results of the customization and bind process. The field shows information that is received from the database and summary statements from the application server.

Using embedded Structured Query Language in Java (SQLJ) with the DB2 for z/OS Legacy driver

Structured Query Language in Java (SQLJ) is a set of programming extensions that enable a programmer, using the Java programming language, to embed statements that provide SQL (Structured Query Language) database requests. You can use the DB2 for z/OS Legacy driver with your data access applications.

About this task

Note:

1. To use SQLJ with WebSphere Application Server for z/OS and the DB2 for z/OS Legacy Driver, you must install DB2 APAR PQ76442.
2. Container Managed Persistence (CMP) beans generated using SQLJ are not supported by the DB2 for z/OS Legacy Driver. Use the DB2 Universal Driver for CMPs that are generated using SQLJ.

Following are the steps required to develop applications with SQLJ that run on WebSphere Application Server for z/OS v6.0 using the DB2 for z/OS Legacy driver.

1. Design your application in Rational Application Developer (RAD) according to your requirements, using SQLJ when necessary. For example, if you develop a bean called Test that uses BMP, code TestBean.sqlj (instead of TestBean.java).
   a. From your DB2 for z/OS installation, copy the db2sqljclasses.zip file to a directory on your workstation, then modify the Java Build Path of your EJB jar project to include db2sqljclasses.zip.
   b. Translate your SQLJ code according to the following steps:
      1) Locate your SQLJ file, then use ASCII mode transfer to FTP it to an HFS in your z/OS environment.
      2) Use the sqlj command to translate your SQLJ code into Java code. This produces two files, one with a .java extension and the other with a .ser extension.

         ```
         sqlj -compile=false SQLJ_FILE_NAME
         ```
      3) Use ASCII mode transfer for the .java file and BINARY mode transfer for the .ser file to move these files back to the directory on your workstation where the SQLJ file resides.
      4) Refresh the project.
   c. Generate deployment code for your application.
   d. Export your EAR file.
2. Install your application
   a. Create a data source using the DB2 for zOS Local JDBC Provider (RRS). When you define your JDBC Provider and DataSource, the default values are sufficient for providing SQLJ support .
   b. Install your application into WebSphere Application Server.

      Use the data source you created in Step 1 to resolve your resource references.
3. Customize your serialized profiles When you generate your deployment code, serialized profiles (files with a .ser extension) that are specific to your application are created. These profiles must be customized in a z/OS environment before they can be used.
   a. Use binary transfer to transfer the serialized profiles to the z/OS environment on which you installed your application. Alternatively, use the Java jar command to extract the serialized profiles from the EJB jar file in your installed EAR directory.
   b. Use the db2profc command to customize your serialized profiles. You can get information on the various options associated with this command from the DB2 documentation; however, here are the minimum requirements to customize your profile:

      ```
      db2profc -pgmname=PROGRAM_NAME PROFILE_NAME
      ```
      - Where:
        - *PROGRAM_NAME* must be a valid MVS PDS member name, and can be up to seven characters.
        - *PROFILE_NAME* is the name of the serialized profile that you want to customize. You must run db2profc once for each profile.
      - The profile customizer creates four DBRM datasets in the PDS *USERNAME*.DBRMLIB.DATA. The member names of the DBRMs begin with what you specified as *PROGRAM_NAME*.
      - Ensure that your CLASSPATH environment variable includes:

- The location of the serialized profile
- The EJB jar file in your installed EAR directory

- Allocate a PDS to contain the DBRMs that are created. Name this PDS *USERNAME*.DBRMLIB.DATA, where *USERNAME* is the user who will execute the db2profc command.

  The following fields are an example:

```
Space units=TRACK
Primary quantity=15
Secondary quantity=5
Directory blocks=10
Record format=FB
Record length=80
Block size=27920
Data set name type=PDS
```

c. Place the existing serialized profiles, which are now customized, into a location that is part of the application classpath and that is ahead of the serialized profiles that exist in your EJB jar file.

   The output of the DB2 profile customizer and the input file have the same name. Move the output file ahead of the original serialized profile in the classpath. Alternatively, you can move the customized profile into the EJB jar file, replacing the original. We recommend that you replace the original file.

   IMPORTANT: If you run the db2profc command from the directory where the serialized profile exists, the profile customizer overwrites the serialized profile. Because you need only the customized version after the profile customizer has run, this is not a problem.

d. Bind your DBRMs into a package.

   Note: You must create your database tables before binding your DBRMs. If you do not, the bind job will fail.

   The db2profc customization command creates a series of DBRMs that must be bound into packages. For each customized profile, four DBRMs are created.

   These DBRMs:

   - Are located in *USERNAME*.DBRMLIB.DATA
   - All have names that begin with what you specified as *PROGRAM_NAME*
   - Are numbered from 1-through-4

     For example, if you log in as IBMUSER, and you specify -pgmname=TESTBMP, then run the db2profc command, the four datasets, TESTBMP1, TESTBMP2, TESTBMP3, AND TESTBMP4 are created and placed in the PDS IBMUSER.DBRMLIB.DATA.

     These datasets must be bound into packages with isolation of UR, CS, RS, and RR, respectively. You must run a bind for each serialized profile that you customize.

e. After you bind all of the DBRMs into packages, bind the packages into a plan. Name the plan whatever you like.

   IMPORTANT: You must also include the JDBC packages in the package list (PKLIST) of your new plan. The default names for the JDBC packages to include are DSNJDBC.DSNJDBC1, ..., DSNJDBC.DSNJDBC4. If your installation did not use the default names for the JDBC packages, contact your DB2 administrator to determine the names of the JDBC packages that you need to include.

   Following is a sample job used to bind a new plan.

   - One serialized profile was created while logged on as IBMUSER.
   - **-pgmname=TESTBMP** was specified to run db2profc.
   - The new plan is named SQLJPLAN.

```
//BBOOLS  JOB (516B,1025),'IBMUSER',MSGCLASS=H,CLASS=A,PRTY=14,
//         NOTIFY=&SYSUID,TIME=1440,USER=IBMUSER,PASSWORD=IBMUSER,
//         MSGLEVEL=(1,1)
```

```
//*******************************************************************
//BINDOLS  EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB  DD  DSN=IBMUSER.DBRMLIB.DATA,DISP=SHR
//*        DD  DSN=MVSDSOM.DB2710.SDSNDBRM,DISP=SHR
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *

 DSN SYSTEM(DB2)

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP1) -
     VALIDATE(BIND) -
     ISOLATION(UR) -
     SQLERROR(NOPACKAGE) -

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP2) -
     VALIDATE(BIND) -
     ISOLATION(CS) -
     SQLERROR(NOPACKAGE) -

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP3) -
     VALIDATE(BIND) -
     ISOLATION(RS) -
     SQLERROR(NOPACKAGE) -

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP4) -
     VALIDATE(BIND) -
     ISOLATION(RR) -
     SQLERROR(NOPACKAGE) -


 BIND PLAN(SQLJPLAN)                -
  QUALIFIER(IBMUSER)                -
  PKLIST(TESTBMP.*                  -
         DSNJDBC.*                ) -
  ACTION(REPLACE) RETAIN            -
  VALIDATE(BIND)

 END
 /*
```

f. Grant the proper authority to your new plan. Use an interface to DB2, such as SPUFI, to grant the authority. Issue this command:

```
GRANT EXECUTE ON PLAN PLANNAME TO APPSERVERID
```

Where:

- *PLANNAME* is the name of the plan that you bound.
- *APPSERVERID* is the ID under which WebSphere Application Server runs; for example, CBSYMSR1.

4. Configure your data source to use your new plan

a. From the WebSphere Application Server for z/OS Administrative Console, navigate to your Data Source and select Custom Properties.
b. Select the Custom Property **planName**.
c. Update the value of **planName** with what you named your plan when it was bound.
d. Set **enableSQLJ** to **true**.

5. Stop and restart your server.
6. Run your application.

Changing the error detection model to use the Exception Checking Model

The error detection model has been expanded and the data source has a configuration option that you can use to select the exception mapping model or the exception checking model for error detection. This configuration option allows the Error Detection Model to comply with Java Database Connectivity (JDBC) 4.0.

About this task

By default, the exception mapping Error Detection Model configuration is selected. The exception mapping Error Detection Model replaces some exceptions raised by the JDBC driver. Exception checking does not do this. If you want to use this configuration, no changes are needed. If you want to use the exception checking model, you need to configure the error detection model in the application server. If you previously changed the **Error Detection Model**, you can also use these steps to change the configuration back to using to the exception mapping model.

1. Open the administrative console.
2. Go to the **WebSphere Application Server Data Source properties** panel for the data source.
   a. Select **Resources** → **JDBC** → **Data Sources** → ***data_source***
   b. Select **WebSphere Application Server Data Source properties**.
3. In the **Error Detection Model** section, click **Use the WebSphere Application Server Exception Checking Model**.

Exceptions pertaining to data access

All enterprise bean container-managed persistence (CMP) beans under the EJB 2.x specification receive a standard EJB exception when an operation fails. JDBC applications receive a standard SQL exception if any JDBC operation fails. The product provides special exceptions for its relational resource adapter (RRA), to indicate that the connection currently held is no longer valid.

- The connection wait timeout exception indicates that the application has waited for the number of seconds specified by the connection timeout setting and has not received a connection. This situation can occur when the pool is at maximum size and all of the connections are in use by other applications for the duration of the wait. In addition, there are no connections currently in use that the application can share because either the connection properties do not match, or the connection is in a different transaction.

  For a Version 4.0 data source, the ConnectionWaitTimeout object creates an exception that is instantiated from the com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException class.

  For J2C connection factories, the ConnectionWaitTimeout object generates a resource exception of the com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException class.

  When the error detection model is configured to exception mapping, later versions of data sources issue an SQL exception of the com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException subclass. When the error detection model is configured to exception checking, later versions of data sources issue an SQL exception of the java.sql.SQLTransientConnectionException class with a chained exception of the com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException class.

- When the error detection model is configured to exception mapping, the stale connection exception indicates that the connection is no longer valid. When the error detection model is configured to exception checking, the JDBC driver raises a JDBC 4.0 exception, such as java.sql.SQLRecoverableException or java.sql.SQLNonTransientConnectionException, or the JDBC driver specifies an appropriate SQLState to indicate that the connection is no longer valid. Read “Stale connections” for more information on this type of exception.

Note: Version 7.0 includes a new custom property for your data sources called userDefinedErrorMap. This property overlays existing entries in the error map by invoking the DataStoreHelper.setUserDefinedMap method. The userDefinedErrorMap can be used to add, change, or remove entries from the error map.

- Entries are delimited by a ; (semicolon).
- Each entry consists of a key and value, where the key is an error code (numeric value) or SQLState, which is text enclosed in double quotes.
- Keys and values are are separated by = (equals sign).

For example, to remove the mapping of SQLState S1000, add a mapping of error code 1062 to duplicate key, and add a mapping of SQLState 08004 to stale connection, you can specify the following value for userDefinedErrorMap:

```
"S1000"=;1062=com.ibm.websphere.ce.cm.DuplicateKeyException;"08004"=
com.ibm.websphere.ce.cm.StaleConnectionException
```

userDefinedErrorMap can be located in the administrative console by selecting the data source and configuring the custom properties.

Stale connections

The product provides a special subclass of the java.sql.SQLException class for using connection pooling to access a relational database. This com.ibm.websphere.ce.cm.StaleConnectionException subclass exists in both a WebSphere 4.0 data source and in the most recent version data source that use the relational resource adapter. This class serves to indicate that the connection currently held is no longer valid.

This situation can occur for many reasons, including the following:

- The application tries to get a connection and fails, as when the database is not started.
- A connection is no longer usable because of a database failure. When an application tries to use a previously obtained connection, the connection is no longer valid. In this case, all connections currently in use by the application can get this error when they try to use the connection.
- The connection is orphaned (because the application had not used it in at most two times the value of the unused timeout setting) and the application tries to use the orphaned connection. This case applies only to Version 4.0 data sources.
- The application tries to use a JDBC resource, such as a statement, obtained on a stale connection.
- A connection is closed by the Version 4.0 data source auto connection cleanup feature and is no longer usable. Auto connection cleanup is the standard mode in which connection management operates. This mode indicates that at the end of a transaction, the transaction manager closes all connections enlisted in that transaction. This enables the transaction manager to ensure that connections are not held for excessive periods of time and that the pool does not reach its maximum number of connections prematurely.

  A negative ramification does ensue, however, when the transaction manager closes the connections and returns the connection to the free pool after a transaction ends. An application cannot obtain a connection in one transaction and try to use it in another transaction. If the application tries this, a stale connection exception occurs because the connection is already closed.

In the case of trying to use an orphaned connection or a connection that is made unavailable by auto connection cleanup, a stale connection exception indicates that the application has attempted to use a connection already returned to the connection pool. It does not indicate an actual problem with the connection. However, other cases of a stale connection exception indicate that the connection to the

database has gone bad, or stale. Once a connection has gone stale, you cannot recover it, and you must completely close the connection rather than returning it to the pool.

Detecting stale connections

When a connection to the database becomes stale, operations on that connection result in an SQL exception from the JDBC driver. Because an SQL exception is a rather generic exception, it contains state and error code values that you can use to determine the meaning of the exception. However, the meanings of these states and error codes vary depending on the database vendor. The connection pooling run time maintains a mapping of which SQL state and error codes indicate a stale connection exception for each database vendor supported. When the connection pooling run time catches an SQL exception, it checks to see if this SQL exception is considered a stale connection exception for the database server in use.

Recovering from stale connections

An application can catch a stale connection exception, depending on the type of error detection model that is configured on the data source:

- When the error detection model is configured to exception mapping, the application server replaces the exception that is raised by the JDBC driver with StaleConnectionException. In this case, the application might trap for a stale connection exception.
- When the error detection model is configured to exception checking, the application server still consults the error map in order to manage the connection pool, but it does not replace the exception. In this case, the application should not trap for a stale connection exception.

Because of the differences between error detection models, the application server provides an API that applications can use with either case to identify stale connections. The API is com.ibm.websphere.rsadapter.WSCallHelper.getDataStoreHelper(datasource).isConnectionError(sqlexception).

Applications are not required to explicitly identify a stale connection exception. Applications are already required to catch the java.sql.SQLException, and the stale connection exception or the exception that is raised by the JDBC driver, always inherits data from the java.sql.SQLException. The stale connection exception, which can result from any method that is declared to raise SQLException, is caught automatically in the general catch-block. However, explicitly identifying a stale connection exception makes it possible for an application to recover from bad connections. When application code identifies a stale connection exception, it should take explicit steps to recover, such as retrying the operation under a new transaction and new connection.

Example: Handling data access exception - stale connection:

These code samples demonstrate how to programmatically address stale connection exceptions for different types of data access clients in different transaction scenarios.

When an application receives a stale connection exception on a database operation, it indicates that the connection currently held is no longer valid. Although it is possible to get an exception for a stale connection on any database operation, the most common time to see a stale connection exception issued is after the first time the connection is retrieved. Because connections are pooled, a database failure is not detected until the operation immediately following its retrieval from the pool, which is the first time communication to the database is attempted. It is only when a failure is detected that the connection is marked stale. The stale connection exception occurs less often if each method that accesses the database gets a new connection from the pool.

Many stale connection exceptions are caused by intermittent problems with the network of the database server. Obtaining a new connection and retrying the operation can result in successful completion without exceptions to the end user. In some cases it is advantageous to add a small wait time between the retries

to give the database server more time to recover. However, applications should not retry operations indefinitely, in case the database is down for an extended period of time.

Note: If you are developing applications for the Application Server with an integrated development environment (IDE) like Eclipse, you might need to import the *app_server_root*/plugins/com.ibm.ws.runtime.jar file into the development environment to take advantage of code that is provided.

Before the application can obtain a new connection for a retry of the operation, roll back the transaction in which the original connection was involved and begin a new transaction. You can break down details on this action into two categories:

Objects operating in a bean-managed global transaction context begun in the same method as the database access.

A servlet or session bean with bean-managed transactions (BMT) can start a global transaction explicitly by calling *begin()* on a *javax.transaction.UserTransaction* object, which you can retrieve from naming or from the bean *EJBContext* object. To commit a bean-managed transaction, the application calls *commit()* on the *UserTransaction* object. To roll back the transaction, the application calls *rollback()*. Entity beans and non-BMT session beans cannot explicitly begin global transactions.

If an object that explicitly started a bean-managed transaction receives a stale connection exception on a database operation, close the connection and roll back the transaction. At this point, the application developer can decide to begin a new transaction, get a new connection, and retry the operation.

The following code fragment shows an example of handling stale connection exceptions in this scenario:

```
//get a userTransaction
javax.transaction.UserTransaction tran = getSessionContext().getUserTransaction();
//retry indicates whether to retry or not
//numOfRetries states how many retries have
// been attempted
boolean retry = false;
int numOfRetries = 0;
java.sql.Connection conn = null;
java.sql.Statement stmt = null;
do {
  try {
    //begin a transaction
    tran.begin();
    //Assumes that a datasource has already been obtained
    //from JNDI
    conn = ds.getConnection();
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    stmt.execute("INSERT INTO EMPLOYEES VALUES
              (0101, 'Bill', 'R', 'Smith')");
    tran.commit();
    retry = false;
  } catch(java.sql.SQLException sqlX)
  {
    // If the error indicates the connection is stale, then
    // rollback and retry the action
    if (com.ibm.websphere.rsadapter.WSCallHelper
        .getDataStoreHelper(ds)
        .isConnectionError(sqlX))
    {
      try {
        tran.rollback();
      } catch (java.lang.Exception e) {
        //deal with exception
        //in most cases, this can be ignored
      }
```

```
        if (numOfRetries < 2) {
          retry = true;
          numOfRetries++;
        } else {
          retry = false;
        }
      }
      else
      {
        //deal with other database exception
        retry = false
      }
    } finally {
      //always cleanup JDBC resources
      try {
        if(stmt != null) stmt.close();
      } catch (java.sql.SQLException sqle) {
        //usually can ignore
      }
      try {
        if(conn != null) conn.close();
      } catch (java.sql.SQLException sqle) {
        //usually can ignore
      }
    }
  } while (retry) ;
```

Objects operating in a global transaction context and transaction not begun in the same method as the database access.

When the object which receives the stale connection exception does not have direct control over the transaction, such as in a container-managed transaction case, the object must mark the transaction for rollback, and then indicate to its caller to retry the transaction. In most cases, you can do this by creating an application exception that indicates to retry that operation. However this action is not always allowed, and often a method is defined only to create a particular exception. This is the case with the ejbLoad() and ejbStore() methods on an enterprise bean. The next two examples explain each of these scenarios.

Example 1: Database access method creates an application exception

When the method that accesses the database is free to create whatever exception is required, the best practice is to catch the stale connection exception and create some application exception that you can interpret to retry the method. The following example shows an EJB client calling a method on an entity bean with transaction demarcation *TX_REQUIRED*, which means that the container begins a global transaction when *insertValue()* is called:

```
public class MyEJBClient
{
  //... other methods here ...

  public void myEJBClientMethod()
  {
    MyEJB myEJB = myEJBHome.findByPrimaryKey("myEJB");
    boolean retry = false;
    do
    {
      try
      {
        retry = false;
        myEJB.insertValue();
      }
      catch(RetryableConnectionException retryable)
      {
        retry = true;
      }
      catch(Exception e) { /* handle some other problem */ }
    }
    while (retry);
```

```
  }
}  //end MyEJBClient

public class MyEJB implements javax.ejb.EntityBean
{
  //... other methods here ...
  public void insertValue() throws RetryableConnectionException,
         java.rmi.EJBException
  {
    try
    {
      conn = ds.getConnection();
      stmt = conn.createStatement();
      stmt.execute("INSERT INTO my_table VALUES (1)");
    }
    catch(java.sql.SQLException sqlX)
    {
      // Find out if the error indicates the connection is stale
      if (com.ibm.websphere.rsadapter.WSCallHelper
          .getDataStoreHelper(ds)
          .isConnectionError(sqlX))
      {
        getSessionContext().setRollbackOnly();
        throw new RetryableConnectionException();
      }
      else
      {
        //handle other database problem
      }
    }
    finally
    {
      //always cleanup JDBC resources
      try
      {
        if(stmt != null) stmt.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
      try
      {
        if(conn != null) conn.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
    }
  }
}  //end MyEJB
```

MyEJBClient first gets a *MyEJB* bean from the home interface, assumed to have been previously retrieved from the Java Naming and Directory Interface (JNDI). It then calls *insertValue()* on the bean. The method on the bean gets a connection and tries to insert a value into a table. If one of the methods fails with a stale connection exception, it marks the transaction for *rollbackOnly* (which forces the caller to roll back this transaction) and creates a new *retryable connection* exception, cleaning up the resources before the exception is thrown. The retryable connection exception is simply an application-defined exception that tells the caller to retry the method. The caller monitors the retryable connection exception and, if it is caught, retries the method. In this example, because the container is beginning and ending the transaction; no transaction management is needed

in the client or the server. Of course, the client could start a bean-managed transaction and the behavior would still be the same, provided that the client also committed or rolled back the transaction.

Example 2: Database access method creates an onlyRemote exception or an EJB exception

Not all methods are allowed to throw exceptions defined by the application. If you use bean-managed persistence (BMP), use the *ejbLoad()* and *ejbStore()* methods to store the bean state. The only exceptions issued from these methods are the java.rmi.Remote exception or the javax.ejb.EJB exception, so you cannot use something similar to the previous example.

If you use container-managed persistence (CMP), the container manages the bean persistence, and it is the container that sees the stale connection exception. If a stale connection is detected, by the time the exception is returned to the client it is simply a remote exception, and so a simple catch-block does not suffice. There is a way to determine if the root cause of a remote exception is a stale connection exception. When a remote exception is created to wrap another exception, the original exception is usually retained. All remote exception instances have a detail property, which is of type *java.lang.Throwable*. With this detail, you can trace back to the original exception and, if it is a stale connection exception, retry the transaction. In reality, when one of these remote exceptions flows from one Java Virtual Machine API to the next, the detail is lost, so it is better to start a transaction in the same server as the database access occurs. For this reason, the following example shows an entity bean accessed by a session bean with bean-managed transaction demarcation.

```
public class MySessionBean extends javax.ejb.SessionBean
{
  ... other methods here ...
  public void mySessionBMTMethod() throws
    java.rmi.EJBException
  {
    javax.transaction.UserTransaction tran =
    getSessionContext().getUserTransaction();
    boolean retry = false;
    do
    {
      try
      {
        retry = false;
        tran.begin();
        // causes ejbLoad() to be invoked
        myBMPBean.myMethod();
        // causes ejbStore() to be invoked
        tran.commit();
      }
      catch(java.rmi.EJBException re)
      {
        try
        {
          tran.rollback();
        }
        catch(Exception e)
        {
          //can ignore
        }
        if (causedByStaleConnection(re))
          retry = true;
        else
          throw re;
      }
      catch(Exception e)
      {
        // handle some other problem
      }
```

```
      finally
      {
        //always cleanup JDBC resources
        try
        {
          if(stmt != null) stmt.close();
        }
        catch (java.sql.SQLException sqle)
        {
          //usually can ignore
        }
        try
        {
          if(conn != null) conn.close();
        }
        catch (java.sql.SQLException sqle)
        {
          //usually can ignore
        }
      }
    }
    while (retry);
  }

  public boolean causedByStaleConnection(java.rmi.EJBException re)
  {
    // Search the exception chain for errors
    // indicating a stale connection
    for (Throwable t = re; t != null; t = t.getCause())
      if (t instanceof RetryableConnectionException)
        return true;

    // Not found to be stale
    return false;
  }
}

public class MyEntityBean extends javax.ejb.EntityBean
{
  ... other methods here ...
  public void ejbStore() throws java.rmi.EJBException
  {
    try
    {
      conn = ds.getConnection();
      stmt = conn.createStatement();
      stmt.execute("UPDATE my_table SET value=1 WHERE
      primaryKey=" + myPrimaryKey);
    }
    catch(java.sql.SQLException sqlX)
    {
      // Find out if the error indicates the connection is stale
      if (com.ibm.websphere.rsadapter.WSCallHelper
          .getDataStoreHelper(ds)
          .isConnectionError(sqlX))
      {
        // rollback the tran when method returns
        getEntityContext().setRollbackOnly();
        throw new java.rmi.EJBException(
          "Exception occurred in ejbStore",
          new RetryableConnectionException(sqlX));
      }
      else
      {
        // handle some other problem
      }
    }
```

```
    finally
    {
      //always cleanup JDBC resources
      try
      {
        if(stmt != null) stmt.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
      try
      {
        if(conn != null) conn.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
    }
  }
}
```

In *mySessionBMTMethod()* of the previous example:

- The session bean first retrieves a *UserTransaction* object from the session context and then begins a global transaction.
- Next, it calls a method on the entity bean, which calls the *ejbLoad()* method. If *ejbLoad()* runs successfully, the client then commits the transaction, causing the *ejbStore()* method to be called.
- In *ejbStore()*, the entity bean gets a connection and writes its state to the database; if the connection retrieved is stale, the transaction is marked *rollbackOnly* and a new *EJBException* that wraps the *RetryableConnectionException* is thrown. That exception is then caught by the client, which cleans up the JDBC resources, rolls back the transaction, and calls *causedByStaleConnection()*, which determines if a stale connection exception is buried somewhere in the exception.
- If the method returns true, the retry flag is set and the transaction is retried; otherwise, the exception is re-issued to the caller.
- The *causedByStaleConnection()* method looks through the chain of detail attributes to find the original exception. Multiple wrapping of exceptions can occur by the time the exception finally gets back to the client, so the method keeps searching until it encounters stale connection exception and *true* is returned; otherwise, there is no stale connection exception in the list and *false* is returned.
- If you are talking to a CMP bean instead of to a BMP bean, the session bean is exactly the same. The CMP bean's *ejbStore()* method would most likely be empty, and the container after calling it would persist the bean with generated code.
- If a stale connection exception occurs during persistence, it is wrapped with a remote exception and returned to the caller. The *causedByStaleConnection()* method would again look through the exception chain and find the root exception, which would be stale connection exception.

Objects operating in a local transaction context.

When a database operation occurs outside of a global transaction context, a local transaction is implicitly begun by the container. This includes servlets or JSPs that do not begin transactions with the *UserTransaction* interface, as well as enterprise beans running in unspecified transaction contexts. As with global transactions, you must roll back the local transaction before the operation is retried. In these cases, the local transaction containment usually ends when the business method ends. The one exception is if you are using activity sessions. In this case the activity session must end before attempting to get a new connection.

When the local transaction occurs in an enterprise bean running in an unspecified transaction context, the enterprise bean client object, outside of the local transaction containment, could use the method described in the previous bullet to retry the transaction. However, when the local

transaction containment takes place as part of a servlet or JSP file, there is no client object available to retry the operation. For this reason, it is recommended to avoid database operations in servlets and JSP files unless they are a part of a user transaction.

Example: Handling servlet JDBC connection exceptions:

The following code sample demonstrates how to set transaction management and connection management properties, such as operation retries, to address stale connection exceptions within a servlet JDBC transaction.

This example code performs the following actions:

- initializes a servlet
- looks up a data source
- specifies error messages, connection retries, and transaction rollback requirements

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
// Import JDBC packages and naming service packages.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.transaction.*;
import com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException;
import com.ibm.websphere.rsadapter.WSCallHelper;

public class EmployeeListTran extends HttpServlet {
    private static DataSource ds   = null;
    private UserTransaction ut = null;
    private static String title    = "Employee List";

// ****************************************************************
// * Initialize servlet when it is first loaded.                  *
// * Get information from the properties file, and look up the    *
// * DataSource object from JNDI to improve performance of the    *
// * the servlet's service methods.                               *
// ****************************************************************
    public void init(ServletConfig config)
    throws ServletException
    {
        super.init(config);
        getDS();
```

```
    }

// *************************************************************
// * Perform the JNDI lookup for the DataSource and            *
// * User Transaction objects.                                 *
// * This method is invoked from init(), and from the service  *
// * method of the DataSource is null                          *
// *************************************************************
    private void getDS() {
        try {
            Hashtable parms = new Hashtable();
            parms.put(Context.INITIAL_CONTEXT_FACTORY,
 "com.ibm.websphere.naming.WsnInitialContextFactory");
            InitialContext ctx = new InitialContext(parms);
            // Perform a naming service lookup to get the DataSource object.
            ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
            ut = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
        } catch (Exception e) {
            System.out.println("Naming service exception: " + e.getMessage());
            e.printStackTrace();
        }
    }

// *************************************************************
// * Respond to user GET request                               *
// *************************************************************
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException
    {
        Connection conn  = null;
        Statement stmt = null;
        ResultSet rs = null;
        Vector employeeList = new Vector();
        // Set retryCount to the number of times you would like to retry after a
        // stale connection exception
        int retryCount = 5;
        // If the Database code processes successfully, we will set error = false
        boolean error = true;
        do
        {
            try
            {
                //Start a new Transaction
                ut.begin();
                // Get a Connection object conn using the DataSource factory.
                conn = ds.getConnection();
                // Run DB query using standard JDBC coding.
                stmt = conn.createStatement();
                String query   = "Select FirstNme, MidInit, LastName " +
                                 "from Employee ORDER BY LastName";
                rs   = stmt.executeQuery(query);
                while (rs.next())
                {
                    employeeList.addElement(rs.getString(3) + ", " + rs.getString(1) +
                }
                //Set error to false to indicate successful completion of the database work
                error=false;
            }
            catch (SQLException sqlX)
            {
              // Determine if the connection request timed out.
              // This code works regardless of which error detection
              // model is used. If exception mapping is enabled, then
              // we need to look for ConnectionWaitTimeoutException.
              // If exception checking is enabled, then look for
              // SQLTransientConnectionException with a chained
              // ConnectionWaitTimeoutException.
```

```
            if ( sqlX instanceof ConnectionWaitTimeoutException
              || sqlX instanceof SQLTransientConnectionException
                 && sqlX.getCause() instanceof ConnectionWaitTimeoutException)
            {
              // This exception is thrown if a connection can not be obtained from the
              // pool within a configurable amount of time.  Frequent occurrences of
              // this exception indicate an incorrectly tuned connection pool

              System.out.println("Connection Wait Timeout Exception during get connection or
process SQL: " + c.getMessage());

              //In general, we do not want to retry after this exception, so set retry count to 0
              //and roll back the transaction
              try
              {
                  ut.setRollbackOnly();
              }
              catch (SecurityException se)
              {
                  //Thrown to indicate that the thread is not allowed to roll back the transaction.
                  System.out.println("Security Exception setting rollback only! " + se.getMessage());
              }
              catch (IllegalStateException ise)
              {
                  //Thrown if the current thread is not associated with a transaction.
                  System.out.println("Illegal State Exception setting rollback only! " + ise.getMessage());
              }
              catch (SystemException sye)
              {
                  //Thrown if the transaction manager encounters an unexpected error condition
                  System.out.println("System Exception setting rollback only! " + sye.getMessage());
              }
              retryCount=0;
            }
            else if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(sqlX))
            {
              // This exception indicates that the connection to the database is no longer valid.
              //Roll back the transaction, then retry several times to attempt to obtain a valid
              //connection, display an error message if the connection still can not be obtained.

              System.out.println("Connection is stale: " + sc.getMessage());

              try
              {
                  ut.setRollbackOnly();
              }
              catch (SecurityException se)
              {
                  //Thrown to indicate that the thread is not allowed to roll back the transaction.
                  System.out.println("Security Exception setting rollback only! " + se.getMessage());
              }
              catch (IllegalStateException ise)
              {
                  //Thrown if the current thread is not associated with a transaction.
                  System.out.println("Illegal State Exception setting rollback only! " + ise.getMessage());
              }
              catch (SystemException sye)
              {
                  //Thrown if the transaction manager encounters an unexpected error condition
                  System.out.println("System Exception setting rollback only! " + sye.getMessage());
              }
              if (--retryCount == 0)
              {
                  System.out.println("Five stale connection exceptions, displaying error page.");
              }
            }
            else
```

```
  {
    System.out.println("SQL Exception during get connection or process SQL: " + sq.getMessage());

    //In general, we do not want to retry after this exception, so set retry count to 0
    //and rollback the transaction
    try
    {
        ut.setRollbackOnly();
    }
    catch (SecurityException se)
    {
        //Thrown to indicate that the thread is not allowed to roll back the transaction.
        System.out.println("Security Exception setting rollback only! " + se.getMessage());
    }
    catch (IllegalStateException ise)
    {
        //Thrown if the current thread is not associated with a transaction.
        System.out.println("Illegal State Exception setting rollback only! " + ise.getMessage());
    }
    catch (SystemException sye)
    {
        //Thrown if the transaction manager encounters an unexpected error condition
        System.out.println("System Exception setting rollback only! " + sye.getMessage());
    }
    retryCount=0;
  }
}
catch (NotSupportedException nse)
{
    //Thrown by UserTransaction begin method if the thread is already associated with a
    //transaction and the Transaction Manager implementation does not support nested
    //transactions.
    System.out.println("NotSupportedException on User Transaction begin: " + nse.getMessage());
}
catch (SystemException se)
{
    //Thrown if the transaction manager encounters an unexpected error condition
    System.out.println("SystemException in User Transaction: " +se.getMessage());
}
catch (Exception e)
{
    System.out.println("Exception in get connection or process SQL: " + e.getMessage());
    //In general, we do not want to retry after this exception, so set retry count to 5
    //and roll back the transaction
    try
    {
        ut.setRollbackOnly();
    }
    catch (SecurityException se)
    {
        //Thrown to indicate that the thread is not allowed to roll back the transaction.
        System.out.println("Security Exception setting rollback only! " + se.getMessage());
    }
    catch (IllegalStateException ise)
    {
        //Thrown if the current thread is not associated with a transaction.
        System.out.println("Illegal State Exception setting rollback only! " + ise.getMessage());
    }
    catch (SystemException sye)
    {
        //Thrown if the transaction manager encounters an unexpected error condition
        System.out.println("System Exception setting rollback only! " + sye.getMessage());
    }
    retryCount=0;
}
finally
{
```

```
            // Always close the connection in a finally statement to ensure proper
            // closure in all cases. Closing the connection does not close and
            // actual connection, but releases it back to the pool for reuse.

            if (rs != null)
            {
               try
               {
                  rs.close();
               }
               catch (Exception e)
               {
                  System.out.println("Close Resultset Exception: " + e.getMessage());
               }
            }
            if (stmt != null)
            {
               try
               {
                  stmt.close();
               }
               catch (Exception e)
               {
                  System.out.println("Close Statement Exception: " + e.getMessage());
               }
            }
            if (conn != null)
            {
               try
               {
                  conn.close();
               }
               catch (Exception e)
               {
                  System.out.println("Close connection exception: " + e.getMessage());
               }
            }
            try
            {
                ut.commit();
            }
            catch (RollbackException re)
            {
                //Thrown to indicate that the transaction has been rolled back rather than committed.
                System.out.println("User Transaction Rolled back! " + re.getMessage());
            }
            catch (SecurityException se)
            {
                //Thrown to indicate that the thread is not allowed to commit the transaction.
                System.out.println("Security Exception thrown on transaction commit: " + se.getMessage());
            }
            catch (IllegalStateException ise)
            {
                //Thrown if the current thread is not associated with a transaction.
                System.out.println("Illegal State Exception thrown on transaction commit: " + ise.getMessage());
            }
            catch (SystemException sye)
            {
                //Thrown if the transaction manager encounters an unexpected error condition
                System.out.println("System Exception thrown on transaction commit: " + sye.getMessage());
            }
            catch (Exception e)
            {
                System.out.println("Exception thrown on transaction commit: " + e.getMessage());
            }
        }
    }
```

```
        while ( error==true && retryCount > 0 );

        // Prepare and return HTML response, prevent dynamic content from being cached
        // on browsers.
        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setHeader("Cache-Control", "no-cache");
        res.setDateHeader("Expires", 0);
        try
        {
            ServletOutputStream out = res.getOutputStream();
            out.println("<HTML>");
            out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>");
            out.println("<BODY>");
            if (error==true)
            {
                out.println("<H1>There was an error processing this request.</H1>" +
                  "Please try the request again, or contact " +
                  " the <a href='mailto:sysadmin@my.com'>System Administrator</a>");
            }
            else if (employeeList.isEmpty())
            {
               out.println("<H1>Employee List is Empty</H1>");
            }
            else
            {
               out.println("<H1>Employee List </H1>");
               for (int i = 0; i < employeeList.size(); i++)
               {
                  out.println(employeeList.elementAt(i) + "<BR>");
               }
            }
            out.println("</BODY></HTML>");
            out.close();
        }
        catch (IOException e)
        {
           System.out.println("HTML response exception: " + e.getMessage());
        }
    }
}
```

Example: Handling connection exceptions for session beans in container-managed database transactions:

The following code sample demonstrates how to roll back transactions and issue exceptions to the bean client in cases of stale connection exceptions.

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================
```

```
package WebSphereSamples.ConnPool;

import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException;
import com.ibm.websphere.rsadapter.WSCallHelper;

/*****************************************************************************
* This bean is designed to demonstrate Database Connections in a
* Container Managed Transaction Session Bean.  Its transaction attribute               *
* should be set to TX_REQUIRED or TX_REQUIRES_NEW.                                     *
*****************************************************************************
*/
public class ShowEmployeesCMTBean implements SessionBean {
   private javax.ejb.SessionContext mySessionCtx = null;
   final static long serialVersionUID = 3206093459760846163L;

   private javax.sql.DataSource ds;

//*****************************************************************************
//* ejbActivate calls the getDS method, which does the JNDI lookup for the DataSource.
//* Because the DataSource lookup is in a separate method, we can also invoke it from
//* the getEmployees method in the case where the DataSource field is null.
//*****************************************************************************
public void ejbActivate() throws java.rmi.EJBException {
   getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.EJBException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.EJBException {}
/**
 * ejbPassivate method
 * @exception java.rmi.EJBException
 */
public void ejbPassivate() throws java.rmi.EJBException {}
/**
 * ejbRemove method
 * @exception java.rmi.EJBException
 */
public void ejbRemove() throws java.rmi.EJBException {}

//*****************************************************************************
//* The getEmployees method runs the database query to retrieve the employees.
//* The getDS method is only called if the DataSource variable is null.
//* Because this session bean uses Container Managed Transactions, it cannot retry the
//* transaction on a StaleConnectionException.  However, it can throw an exception to
//* its client indicating that the operation is retriable.
//*****************************************************************************

public Vector getEmployees() throws ConnectionWaitTimeoutException, SQLException,
       RetryableConnectionException
{
   Connection conn  = null;
   Statement stmt = null;
   ResultSet rs = null;
   Vector employeeList = new Vector();

   if (ds == null) getDS();

   try
   {
```

```
   // Get a Connection object conn using the DataSource factory.
   conn = ds.getConnection();
   // Run DB query using standard JDBC coding.
   stmt = conn.createStatement();
   String query   = "Select FirstNme, MidInit, LastName " +
                    "from Employee ORDER BY LastName";
   rs   = stmt.executeQuery(query);
   while (rs.next())
   {
      employeeList.addElement(rs.getString(3) + ", " + rs.getString(1) + " " +
   }
}
catch (SQLException sqlX)
{
  // Determine if the connection is stale
  if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(sqlX))
  {
    // This exception indicates that the connection to the database is no longer valid.
    // Roll back the transaction, and throw an exception to the client indicating they
    // can retry the transaction if desired.

    System.out.println("Connection is stale: " + sqlX.getMessage());
    System.out.println("Rolling back transaction and throwing  RetryableConnectionException");

    mySessionCtx.setRollbackOnly();
    throw new RetryableConnectionException(sqlX.toString());
  }
  // Determine if the connection request timed out.
  else if ( sqlX instanceof ConnectionWaitTimeoutException
         || sqlX instanceof SQLTransientConnectionException
            && sqlX.getCause() instanceof ConnectionWaitTimeoutException)
  {
    // This exception is thrown if a connection can not be obtained from the
    // pool within a configurable amount of time.  Frequent occurrences of
    // this exception indicate an incorrectly tuned connection pool

    System.out.println("Connection Wait Timeout Exception during get connection or process SQL: " +
    sqlX.getMessage());
    throw sqlX instanceof ConnectionWaitTimeoutException ?
          sqlX :
          (ConnectionWaitTimeoutException) sqlX.getCause();
  }
  else
  {
    //Throwing a remote exception will automatically roll back the container managed
    //transaction

    System.out.println("SQL Exception during get connection or process SQL: " +
         sqlX.getMessage());
    throw sqlX;
  }
}
finally
{
  // Always close the connection in a finally statement to ensure proper
  // closure in all cases. Closing the connection does not close and
  // actual connection, but releases it back to the pool for reuse.

  if (rs != null)
  {
    try
    {
      rs.close();
    }
    catch (Exception e)
    {
      System.out.println("Close Resultset Exception: " +
```

```
                    e.getMessage());
      }
    }
    if (stmt != null)
    {
      try
      {
        stmt.close();
      }
      catch (Exception e)
      {
        System.out.println("Close Statement Exception: " +
                    e.getMessage());
      }
    }
    if (conn != null)
    {
      try
      {
        conn.close();
      }
      catch (Exception e)
      {
        System.out.println("Close connection exception: " + e.getMessage());
      }
    }
  }
  return employeeList;
}

/**
 * getSessionContext method
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
  return mySessionCtx;
}
//*****************************************************************************
//* The getDS method performs the JNDI lookup for the data source.
//* This method is called from ejbActivate, and from getEmployees if the data source
//* object is null.
//*****************************************************************************

private void getDS() {
  try {
     Hashtable parms = new Hashtable();
     parms.put(Context.INITIAL_CONTEXT_FACTORY,
           "com.ibm.websphere.naming.WsnInitialContextFactory");
     InitialContext ctx = new InitialContext(parms);
     // Perform a naming service lookup to get the DataSource object.
     ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
  }
      catch (Exception e) {
     System.out.println("Naming service exception: " + e.getMessage());
     e.printStackTrace();
  }
}
/**
 * setSessionContext method
 * @param ctx javax.ejb.SessionContext
 * @exception java.rmi.EJBException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws java.rmi.EJBException {
  mySessionCtx = ctx;
}
}
```

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Session Bean
 */
public interface ShowEmployeesCMTHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
* @return WebSphereSamples.ConnPool.ShowEmployeesCMT
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
WebSphereSamples.ConnPool.ShowEmployeesCMT create() throws javax.ejb.CreateException,
   java.rmi.RemoteException;
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface ShowEmployeesCMT extends javax.ejb.EJBObject {

/**
 *
 * @return java.util.Vector
```

```
 */
java.util.Vector getEmployees() throws java.sql.SQLException, java.rmi.RemoteException,
 ConnectionWaitTimeoutException, WebSphereSamples.ConnPool.RetryableConnectionException;
}
//==================START_PROLOG=====================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=======================================

package WebSphereSamples.ConnPool;

/**
 * Exception indicating that the operation can be retried
 * Creation date: (4/2/2001 10:48:08 AM)
 * @author: Administrator
 */
public class RetryableConnectionException extends Exception {
/**
 * RetryableConnectionException constructor.
 */
public RetryableConnectionException() {
   super();
}
/**
 * RetryableConnectionException constructor.
 * @param s java.lang.String
 */
public RetryableConnectionException(String s) {
   super(s);
}
}
```

Example: Handling connection exceptions for session beans in bean-managed database transactions:

The following code sample demonstrates your options for addressing stale connection exceptions. You can set different transaction management and connection management parameters, such as the number of operation retries, and the connection timeout interval.

```
//==================START_PROLOG=====================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
```

```
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import javax.transaction.*;
import com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException;
import com.ibm.websphere.rsadapter.WSCallHelper;

/*****************************************************************************
* This bean is designed to demonstrate Database Connections in a                *
* Bean-Managed Transaction Session Bean.  Its transaction attribute             *
* should be set to TX_BEANMANAGED.
*****************************************************************************/
public class ShowEmployeesBMTBean implements SessionBean {
   private javax.ejb.SessionContext mySessionCtx = null;
   final static long serialVersionUID = 3206093459760846163L;

   private javax.sql.DataSource ds;

   private javax.transaction.UserTransaction userTran;

//****************************************************************************
//* ejbActivate calls the getDS method, which makes the JNDI lookup for the DataSource
//* Because the DataSource lookup is in a separate method, we can also invoke it from
//* the getEmployees method in the case where the DataSource field is null.
//****************************************************************************
public void ejbActivate() throws java.rmi.EJBException {
   getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.EJBException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.EJBException {}
/**
 * ejbPassivate method
 * @exception java.rmi.EJBException
 */
public void ejbPassivate() throws java.rmi.EJBException {}
/**
 * ejbRemove method
 * @exception java.rmi.EJBException
 */
public void ejbRemove() throws java.rmi.EJBException {}

//****************************************************************************
//* The getEmployees method runs the database query to retrieve the employees.
//* The getDS method is only called if the DataSource or userTran variables are null.
//* If a stale connection occurs, the bean retries the transaction 5 times,
//* then throws an EJBException.
//****************************************************************************

public Vector getEmployees() throws EJBException {
   Connection conn  = null;
   Statement stmt = null;
   ResultSet rs = null;
   Vector employeeList = new Vector();
```

```
// Set retryCount to the number of times you would like to retry after a
// stale connection
int retryCount = 5;

// If the Database code processes successfully, we will set error = false
boolean error = true;

if (ds == null || userTran == null) getDS();
do
{
      try
      {
         //try/catch block for UserTransaction work
         //Begin the transaction
         userTran.begin();
         try
         {
            //try/catch block for database work
            //Get a Connection object conn using the DataSource factory.
            conn = ds.getConnection();
            // Run DB query using standard JDBC coding.
            stmt = conn.createStatement();
            String query   = "Select FirstNme, MidInit, LastName " +
          "from Employee ORDER BY LastName";
            rs   = stmt.executeQuery(query);
            while (rs.next())
            {
               employeeList.addElement(rs.getString(3) + ", " + rs.getString(1) + " " + rs.getString(2));
            }
            //Set error to false, as all database operations are successfully completed
            error = false;
         }
         catch (SQLException sqlX)
         {
           if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(sqlX))
           {
             // This exception indicates that the connection to the database is no longer valid.
             // Rollback the transaction, and throw an exception to the client indicating they
             // can retry the transaction if desired.

             System.out.println("Stale connection: " +
             se.getMessage());
             userTran.rollback();
             if (--retryCount == 0)
             {
               //If we have already retried the requested number of times, throw an EJBException.
               throw new EJBException("Transaction Failure: " + sqlX.toString());
             }
             else
             {
               System.out.println("Retrying transaction, retryCount = " +
                      retryCount);
             }
           }
           else if (sqlX instanceof ConnectionWaitTimeoutException
                 || sqlX instanceof SQLTransientConnectionException
                    && sqlX.getCause() instanceof ConnectionWaitTimeoutException)
           {
             // This exception is thrown if a connection can not be obtained from the
             // pool within a configurable amount of time.  Frequent occurrences of
             // this exception indicate an incorrectly tuned connection pool

             System.out.println("Connection request timed out: " +
             sqlX.getMessage());
             userTran.rollback();
             throw new EJBException("Transaction failure: " + sqlX.getMessage());
           }
```

```
              else
              {
       // This catch handles all other SQL Exceptions
                System.out.println("SQL Exception during get connection or process SQL: " +
            sqlX.getMessage());
                userTran.rollback();
                throw new EJBException("Transaction failure: " + sqlX.getMessage());
              }
            finally
            {
              // Always close the connection in a finally statement to ensure proper
              // closure in all cases. Closing the connection does not close and
              // actual connection, but releases it back to the pool for reuse.

              if (rs != null) {
               try {
                  rs.close();
               }
               catch (Exception e) {
                  System.out.println("Close Resultset Exception: " + e.getMessage());
               }
            }
            if (stmt != null) {
               try {
                  stmt.close();
 }
            catch (Exception e) {
               System.out.println("Close Statement Exception: " + e.getMessage());
            }
         }
         if (conn != null) {
            try {
               conn.close();
            }
            catch (Exception e) {
               System.out.println("Close connection exception: " + e.getMessage());
            }
         }
      }
      if (!error) {
         //Database work completed successfully, commit the transaction
         userTran.commit();
      }
      //Catch UserTransaction exceptions
      }
      catch (NotSupportedException nse) {

//Thrown by UserTransaction begin method if the thread is already associated with a
//transaction and the Transaction Manager implementation does not support nested transactions.
 System.out.println("NotSupportedException on User Transaction begin: " +
                                nse.getMessage());
         throw new EJBException("Transaction failure: " + nse.getMessage());
      }
      catch (RollbackException re) {
//Thrown to indicate that the transaction has been rolled back rather than committed.
         System.out.println("User Transaction Rolled back! " + re.getMessage());
         throw new EJBException("Transaction failure: " + re.getMessage());
      }
      catch (SystemException se) {
     //Thrown if the transaction manager encounters an unexpected error condition
         System.out.println("SystemException in User Transaction: "+ se.getMessage());
         throw new EJBException("Transaction failure: " + se.getMessage());
      }
      catch (Exception e) {
         //Handle any generic or unexpected Exceptions
         System.out.println("Exception in User Transaction: " + e.getMessage());
         throw new EJBException("Transaction failure: " + e.getMessage());
```

```
      }
   }
   while (error);
   return employeeList;
}
/**
 * getSessionContext method comment
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
   return mySessionCtx;
}

//*****************************************************************************
//* The getDS method performs the JNDI lookup for the DataSource.
//* This method is called from ejbActivate, and from getEmployees if the DataSource
//* object is null.
//*****************************************************************************
private void getDS() {
   try {
      Hashtable parms = new Hashtable();
parms.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.ibm.websphere.naming.WsnInitialContextFactory");
      InitialContext ctx = new InitialContext(parms);

      // Perform a naming service lookup to get the DataSource object.
      ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
      //Create the UserTransaction object
      userTran = mySessionCtx.getUserTransaction();
   }
   catch (Exception e) {
      System.out.println("Naming service exception: " + e.getMessage());
      e.printStackTrace();
}
}
/**
 * setSessionContext method
 * @param ctx javax.ejb.SessionContext
 * @exception java.rmi.EJBException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws java.rmi.EJBException {
   mySessionCtx = ctx;
}
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Session Bean
```

```
 */
public interface ShowEmployeesBMTHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
 * @return WebSphereSamples.ConnPool.ShowEmployeesBMT
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
WebSphereSamples.ConnPool.ShowEmployeesBMT create() throws javax.ejb.CreateException,
    java.rmi.RemoteException;
}
//==================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface ShowEmployeesBMT extends javax.ejb.EJBObject {

/**
 *
 * @return java.util.Vector
 */
java.util.Vector getEmployees() throws java.rmi.RemoteException, javax.ejb.EJBException;
}
```

Example: Handling connection exceptions for BMP beans in container-managed database transactions:

The following code sample demonstrates how to roll back transactions and issue exceptions to the bean client in cases of stale connection exceptions.

```
//==================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2005,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
```

```
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import com.ibm.websphere.rsadapter.WSCallHelper;

/**
 * This is an Entity Bean class with five BMP fields
 * String firstName, String lastName, String middleInit
 * String empNo, int edLevel
 */
public class EmployeeBMPBean implements EntityBean {
       private javax.ejb.EntityContext entityContext = null;
       final static long serialVersionUID = 3206093459760846163L;

       private java.lang.String firstName;
       private java.lang.String lastName;
       private String middleInit;
       private javax.sql.DataSource ds;
       private java.lang.String empNo;
       private int edLevel;
/**
 * ejbActivate method
 * ejbActivate calls getDS(), which performs the
 * JNDI lookup for the datasource.
 */
public void ejbActivate() {
       getDS();
}
/**
 * ejbCreate method for a BMP entity bean
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception javax.ejb.CreateException
 */
public WebSphereSamples.ConnPool.EmployeeBMPKey ejbCreate(String empNo,
String firstName, String lastName, String middleInit, int edLevel) throws
javax.ejb.CreateException {

   Connection conn = null;
   PreparedStatement ps = null;

   if (ds == null) getDS();

   this.empNo = empNo;
   this.firstName = firstName;
   this.lastName = lastName;
   this.middleInit = middleInit;
   this.edLevel = edLevel;

   String sql = "insert into Employee (empNo, firstnme, midinit, lastname,
        edlevel) values (?,?,?,?,?)";

   try {
      conn = ds.getConnection();
      ps = conn.prepareStatement(sql);
      ps.setString(1, empNo);
      ps.setString(2, firstName);
      ps.setString(3, middleInit);
      ps.setString(4, lastName);
```

```
      ps.setInt(5, edLevel);

 if (ps.executeUpdate() != 1){
         System.out.println("ejbCreate Failed to add user.");
         throw new CreateException("Failed to add user.");
      }
   }
   catch (SQLException se)
   {
      if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(se))
      {
         // This exception indicates that the connection to the database is no longer valid.
         // Rollback the transaction, and throw an exception to the client indicating they
         // can retry the transaction if desired.

         System.out.println("Connection is stale: " +  se.getMessage());
         throw new CreateException(se.getMessage());
      }
      else
      {
         System.out.println("SQL Exception during get connection or process SQL: " +
                            se.getMessage());
         throw new CreateException(se.getMessage());
      }
   }
   finally
   {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close an
      // actual connection, but releases it back to the pool for reuse.
      if (ps != null)
      {
         try
         {
            ps.close();
         }
         catch (Exception e)
         {
            System.out.println("Close Statement Exception: " + e.getMessage());
         }
      }
      if (conn != null)
      {
         try
         {
            conn.close();
         }
         catch (Exception e)
         {
            System.out.println("Close connection exception: " + e.getMessage());
         }
      }
   }
   return new EmployeeBMPKey(this.empNo);
}
/**
 * ejbFindByPrimaryKey method
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param primaryKey WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception javax.ejb.FinderException
 */
public WebSphereSamples.ConnPool.EmployeeBMPKey
     ejbFindByPrimaryKey(WebSphereSamples.ConnPool.EmployeeBMPKey primaryKey)
                 javax.ejb.FinderException {
       loadByEmpNo(primaryKey.empNo);
       return primaryKey;
}
```

```
/**
 * ejbLoad method
 */
public void ejbLoad() {
       try {
              EmployeeBMPKey pk = (EmployeeBMPKey) entityContext.getPrimaryKey();
              loadByEmpNo(pk.empNo);
       } catch (FinderException fe) {
              throw new EJBException("Cannot load Employee state from database.");
       }
}
/**
 * ejbPassivate method
 */
public void ejbPassivate() {}
/**
 * ejbPostCreate method for a BMP entity bean
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 */
public void ejbPostCreate(String empNo, String firstName, String lastName, String middleInit,
    int edLevel) {}
/**
 * ejbRemove method
 * @exception javax.ejb.RemoveException
 */
public void ejbRemove() throws javax.ejb.RemoveException
{

   if (ds == null)
      GetDS();

   String sql = "delete from Employee where empNo=?";
   Connection con = null;
   PreparedStatement ps = null;
   try
   {
      con = ds.getConnection();
      ps = con.prepareStatement(sql);
      ps.setString(1, empNo);
      if (ps.executeUpdate() != 1)
      {
         throw new EJBException("Cannot remove employee: " + empNo);
      }
   }
   catch (SQLException se)
   {
      if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(se))
      {
         // This exception indicates that the connection to the database is no longer valid.
         // Rollback the transaction, and throw an exception to the client indicating they
         // can retry the transaction if desired.

         System.out.println("Connection is stale: " + se.getMessage());
         throw new EJBException(se.getMessage());
      }
      else
      {
         System.out.println("SQL Exception during get connection or process SQL: " +
                         se.getMessage());
         throw new EJBException(se.getMessage());
      }
   }
   finally
   {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close an
      // actual connection, but releases it back to the pool for reuse.
```

```
      if (ps != null)
      {
         try
         {
            ps.close();
         }
         catch (Exception e)
         {
            System.out.println("Close Statement Exception: " + e.getMessage());
         }
      }
      if (con != null)
      {
         try
         {
            con.close();
         }
         catch (Exception e)
         {
            System.out.println("Close connection exception: " + e.getMessage());
         }
      }
   }
}
/**
 * Get the employee's edLevel
 * Creation date: (4/20/2001 3:46:22 PM)
 * @return int
 */
public int getEdLevel() {
       return edLevel;
}
/**
 * getEntityContext method
 * @return javax.ejb.EntityContext
 */
public javax.ejb.EntityContext getEntityContext() {
       return entityContext;
}
/**
 * Get the employee's first name
 * Creation date: (4/19/2001 1:34:47 PM)
 * @return java.lang.String
 */
public java.lang.String getFirstName() {
       return firstName;
}
/**
 * Get the employee's last name
 * Creation date: (4/19/2001 1:35:41 PM)
 * @return java.lang.String
 */
public java.lang.String getLastName() {
       return lastName;
}
/**
* get the employee's middle initial
 * Creation date: (4/19/2001 1:36:15 PM)
 * @return char
 */
public String getMiddleInit() {
       return middleInit;
}
/**
 * Lookup the DataSource from JNDI
 * Creation date: (4/19/2001 3:28:15 PM)
 */
```

```
private void getDS() {
       try {
              Hashtable parms = new Hashtable();
              parms.put(Context.INITIAL_CONTEXT_FACTORY,
                           "com.ibm.websphere.naming.WsnInitialContextFactory");
              InitialContext ctx = new InitialContext(parms);
              // Perform a naming service lookup to get the DataSource object.
              ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
       }
       catch (Exception e) {
              System.out.println("Naming service exception: " + e.getMessage());
              e.printStackTrace();
       }
}
/**
 * Load the employee from the database
 * Creation date: (4/19/2001 3:44:07 PM)
 * @param empNo java.lang.String
 */
private void loadByEmpNo(String empNoKey) throws javax.ejb.FinderException
{
       String sql = "select empno, firstnme, midinit, lastname, edLevel from employee where empno = ?";
       Connection conn = null;
       PreparedStatement ps = null;
       ResultSet rs = null;

       if (ds == null) getDS();

       try
       {
              // Get a Connection object conn using the DataSource factory.
              conn = ds.getConnection();
              // Run DB query using standard JDBC coding.
              ps = conn.prepareStatement(sql);
              ps.setString(1, empNoKey);
              rs = ps.executeQuery();
              if (rs.next())
              {
                     empNo= rs.getString(1);
                     firstName=rs.getString(2);
                     middleInit=rs.getString(3);
                     lastName=rs.getString(4);
                     edLevel=rs.getInt(5);
              }
              else
              {
                throw new ObjectNotFoundException("Cannot find employee number " +
                                                  empNoKey);
              }
       }
       catch (SQLException se)
       {
         if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(se))
         {
           // This exception indicates that the connection to the database is no longer valid.
           // Roll back the transaction, and throw an exception to the client indicating they
           // can retry the transaction if desired.

           System.out.println("Connection is stale: " + se.getMessage());
           throw new FinderException(se.getMessage());
         }
         else
         {
           System.out.println("SQL Exception during get connection or process SQL: " +
                              se.getMessage());
           throw new FinderException(se.getMessage());
         }
```

```
        }
        finally
        {
          // Always close the connection in a finally statement to ensure
          // proper closure in all cases. Closing the connection does not
          // close an actual connection, but releases it back to the pool
          // for reuse.
          if (rs != null)
          {
             try
             {
                rs.close();
             }
             catch (Exception e)
             {
                System.out.println("Close Resultset Exception: " + e.getMessage());
             }
          }
          if (ps != null)
          {
             try
             {
                ps.close();
             }
             catch (Exception e)
             {
                System.out.println("Close Statement Exception: " + e.getMessage());
             }
          }
          if (conn != null)
          {
             try
             {
                conn.close();
             }
             catch (Exception e)
             {
                System.out.println("Close connection exception: " + e.getMessage());
             }
          }
        }
}
/**
 * set the employee's education level
 * Creation date: (4/20/2001 3:46:22 PM)
 * @param newEdLevel int
 */
public void setEdLevel(int newEdLevel) {
        edLevel = newEdLevel;
}
/**
 * setEntityContext method
 * @param ctx javax.ejb.EntityContext
 */
public void setEntityContext(javax.ejb.EntityContext ctx) {
        entityContext = ctx;
}
/**
 * set the employee's first name
 * Creation date: (4/19/2001 1:34:47 PM)
 * @param newFirstName java.lang.String
 */
public void setFirstName(java.lang.String newFirstName) {
        firstName = newFirstName;
}
/**
 * set the employee's last name
```

```
 * Creation date: (4/19/2001 1:35:41 PM)
 * @param newLastName java.lang.String
 */
public void setLastName(java.lang.String newLastName) {
       lastName = newLastName;
}
/**
 * set the employee's middle initial
 * Creation date: (4/19/2001 1:36:15 PM)
 * @param newMiddleInit char
 */
public void setMiddleInit(String newMiddleInit) {
       middleInit = newMiddleInit;
}
/**
 * unsetEntityContext method
 */
public void unsetEntityContext() {
       entityContext = null;
}
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface EmployeeBMP extends javax.ejb.EJBObject {

/**
 *
 * @return int
  */
int getEdLevel() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getFirstName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getLastName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getMiddleInit() throws java.rmi.RemoteException;
```

```
/**
 *
 * @return void
 * @param newEdLevel int
 */
void setEdLevel(int newEdLevel) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newFirstName java.lang.String
 */
void setFirstName(java.lang.String newFirstName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newLastName java.lang.String
 */
void setLastName(java.lang.String newLastName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newMiddleInit java.lang.String
 */
void setMiddleInit(java.lang.String newMiddleInit) throws java.rmi.RemoteException;
}
//==================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface EmployeeBMP extends javax.ejb.EJBObject {

/**
 *
 * @return int
  */
int getEdLevel() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getFirstName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getLastName() throws java.rmi.RemoteException;
/**
```

```
 *
 * @return java.lang.String
 */
java.lang.String getMiddleInit() throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newEdLevel int
 */
void setEdLevel(int newEdLevel) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newFirstName java.lang.String
 */
void setFirstName(java.lang.String newFirstName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newLastName java.lang.String
 */
void setLastName(java.lang.String newLastName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newMiddleInit java.lang.String
 */
void setMiddleInit(java.lang.String newMiddleInit) throws java.rmi.RemoteException;
}
//==================START_PROLOG=====================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=======================================

package WebSphereSamples.ConnPool;

/**
* This is a Primary Key Class for the Entity Bean
**/
public class EmployeeBMPKey implements java.io.Serializable {
   public String empNo;
   final static long serialVersionUID = 3206093459760846163L;

/**
* EmployeeBMPKey() constructor
*/
public EmployeeBMPKey()  {
}
/**
* EmployeeBMPKey(String key) constructor
*/
public EmployeeBMPKey(String key)  {
   empNo = key;
```

```
}
/**
* equals method
* - user must provide a proper implementation for the equal method. The generated
*   method assumes the key is a String object.
*/
public boolean equals (Object o)  {
   if (o instanceof EmployeeBMPKey)
      return empNo.equals(((EmployeeBMPKey)o).empNo);
   else
      return false;
}
/**
* hashCode method
* - user must provide a proper implementation for the hashCode method. The generated
*    method assumes the key is a String object.
*/
public int hashCode ()  {
   return empNo.hashCode();
```

Example: Handling data access exception - ConnectionWaitTimeoutException (for the JDBC API)

This code sample demonstrates how you specify the conditions under which the application server issues the ConnectionWaitTimeoutException for a JDBC application.

In all cases in which the ConnectionWaitTimeoutException is caught, there is very little that can be done to recover.

```
public void test1() {
      java.sql.Connection conn = null;
      java.sql.Statement stmt = null;
      java.sql.ResultSet rs = null;

      try {
         // Look for datasource
         java.util.Properties props = new java.util.Properties();
         props.put(
            javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
         ic = new javax.naming.InitialContext(props);
         javax.sql.DataSource ds1 = (javax.sql.DataSource) ic.lookup(jndiString);

         // Get Connection.
         conn = ds1.getConnection();
         stmt = conn.createStatement();
         rs = stmt.executeQuery("select * from mytable where this = 54");
      }
      catch (java.sql.SQLException sqlX) {
     if (sqlX instanceof com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException
      || sqlX instanceof java.sql.SQLTransientConnectionException
         && sqlX.getCause() instanceof com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException)
     {
         //notify the user that the system could not provide a
         //connection to the database.  This usually happens when the
         //connection pool is full and there is no connection
         //available for to share.
      }
     else
     {
         // handle other database problems.
     }
      }
      finally {
         if (rs != null)
            try {
               rs.close();
```

```
            }
            catch (java.sql.SQLException sqle1) {
            }
         if (stmt != null)
            try {
               stmt.close();
            }
            catch (java.sql.SQLException sqle1) {
            }
         if (conn != null)
            try {
               conn.close();
            }
            catch (java.sql.SQLException sqle1) {
            }
      }
   }
```

Example: Handling data access exception - ConnectionWaitTimeoutException (for Java EE Connector Architecture)

This code sample demonstrates how you specify the conditions under which WebSphere Application Server issues the ConnectionWaitTimeout exception for a JCA application.

In all cases in which the `ConnectionWaitTimeout` exception is caught, there is very little to do for recovery.

The following code fragment shows how to use this exception in Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA):

```
/**
 * This method does a simple Connection test.
 */
public void testConnection()
   throws javax.naming.NamingException, javax.resource.ResourceException,
       com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException {
   javax.resource.cci.ConnectionFactory factory = null;
   javax.resource.cci.Connection conn = null;
   javax.resource.cci.ConnectionMetaData metaData = null;
   try {
      // lookup the connection factory
      if (verbose) System.out.println("Look up the connection factory...");
try {
factory =
  (javax.resource.cci.ConnectionFactory) (new InitialContext()).lookup("java:comp/env/eis/Sample");
  }
   catch (javax.naming.NamingException ne) {
      // Connection factory cannot be looked up.
      throw ne;
   }
    // Get connection
      if (verbose) System.out.println("Get the connection...");
      conn = factory.getConnection();
      // Get ConnectionMetaData
      metaData = conn.getMetaData();
      // Print out the metadata Informatin.
      System.out.println("EISProductName is " + metaData.getEISProductName());
   }
   catch (com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException cwtoe) {
      // Connection Wait Timeout
      throw cwtoe;
   }
   catch (javax.resource.ResourceException re) {
      // Something wrong with connections.
      throw re;
   }
   finally {
      if (conn != null) {
```

```
         try {
            conn.close();
         }
         catch (javax.resource.ResourceException re) {
         }
      }
   }
}
```

Example: Handling data access exception - error mapping in DataStoreHelper

The application server provides a DataStoreHelper interface for mapping different database SQL error codes to the appropriate exceptions in the application server.

Error mapping is necessary because various database vendors can provide different SQL errors and codes that represent that same issue. For example, the stale connection exception has different codes in different databases. The **DB2** SQLCODEs of *1015, 1034, 1036* , and so on indicate that the connection is no longer available because of a temporary database problem. The **Oracle** SQLCODEs of *28, 3113, 3114*, and so on indicate the same situation.

Mapping these error codes to standard exceptions provides the consistency that makes applications portable across different installations of the application server. The following code segment illustrates how to add two error codes into the error map:

```
public class NewDSHelper extends GenericDataStoreHelper
{
  public NewDSHelper(java.util.Properties dataStoreHelperProperties)
  {
    super(dataStoreHelperProperties);
    java.util.Hashtable myErrorMap = null;
    myErrorMap = new java.util.Hashtable();
    myErrorMap.put(new Integer(-803), myDuplicateKeyException.class);
    myErrorMap.put(new Integer(-1015), myStaleConnectionException.class);
    myErrorMap.put("S1000", MyTableNotFoundException.class);
    setUserDefinedMap(myErrorMap);
    ...
  }
}
```

Note: Version 7.0 includes a new custom property for your data sources called userDefinedErrorMap. This property overlays existing entries in the error map by invoking the DataStoreHelper.setUserDefinedMap method. The userDefinedErrorMap can be used to add, change, or remove entries from the error map.

- Entries are delimited by a ; (semicolon).
- Each entry consists of a key and value, where the key is an error code (numeric value) or SQLState, which is text enclosed in double quotes.
- Keys and values are are separated by = (equals sign).

For example, to remove the mapping of SQLState S1000, add a mapping of error code 1062 to duplicate key, and add a mapping of SQLState 08004 to stale connection, you can specify the following value for userDefinedErrorMap:

```
"S1000"=;1062=com.ibm.websphere.ce.cm.DuplicateKeyException;"08004"=
com.ibm.websphere.ce.cm.StaleConnectionException
```

userDefinedErrorMap can be located in the administrative console by selecting the data source and configuring the custom properties.

A configuration option referred to as the Error Detection Model controls how the error map is used. At V6 and earlier, Exception Mapping was the only option available for the Error Detection Model. At V7 and later, another option called Exception Checking is also available. Under the Exception Mapping model, the application server consults the error map and replaces exceptions with the corresponding exception type

listed in the error map. Under the Exception Checking model, the application server still consults the error map for its own purposes but does not replace exceptions. If you wish to continue to use Exception Mapping, you do not need to change anything. Exception Mapping is the default Error Detection Model. If you wish to use the Exception Checking Model, refer to the topic ″Changing the Error Detection Model to use the Exception Checking Model″ in the related links section.

Database deadlock and foreign key conflicts

Repetition of certain SQL error messages indicate problems, such as database referential integrity violations, that you can prevent by using the CMP sequence grouping feature.

Exceptions resulting from foreign key conflicts due to violations of database referential integrity

A database *referential integrity* (RI) policy prescribes rules for how data is written to and deleted from the database tables to maintain relational consistency. Run-time requirements for managing bean persistence, however, can cause an EJB application to violate RI rules, which can cause database exceptions.

Your EJB application is violating database RI if you see an exception message in your WebSphere Application Server trace or log file that is similar to one of the following messages (which were produced in an environment running DB2):

```
The insert or update value of the FOREIGN KEY table1.name_of_foreign_key_constraint
is not equal to any value of the parent key of the parent table.
```

or

```
A parent row cannot be deleted because the relationship table1.name_of_foreign_key_constraint
is not equal to any value of the parent key of the parent table.
```

To prevent these exceptions, you must designate the order in which entity beans update relational database tables by defining sequence groups for the beans.

Exceptions resulting from deadlock caused by optimistic concurrency control schemes

Additionally, sequence grouping can minimize transaction rollback exceptions for entity beans that are configured for optimistic concurrency control. Optimistic concurrency control dictates that database locks be held for minimal amounts of time, so that a maximum number of transactions consistently have access to the data. In such a highly available database, concurrent transactions can attempt to lock the same table row and create deadlock. The resulting exceptions can generate messages similar to the following (which was produced in an environment running DB2):

```
Unsuccessful execution caused by deadlock or timeout.
```

Use the sequence grouping feature to order bean persistence so that database deadlock is less likely to occur.

CMP connection factories collection

Use this page to view existing CMP connection factories settings.

These connection factories are used by a container-managed persistence (CMP) bean to access any backend data store. A CMP connection factory is used by EJB model 2.x Entities with CMP version 2.x. Connection factories listed on this page are created automatically under the WebSphere Relational Resource Adapter when you check the box *Use this Data Source in container managed persistence (CMP)* in the General Properties area on the Data Source page. You cannot modify the settings for a CMP connection factory, and you cannot delete CMP connection factories from this collection. To remove the CMP connection factory object, you must navigate to the data source associated with the CMP connection factory and uncheck the *Use this Data Source for CMP* check box.

To view this administrative console page:

1. Click **Resources** → **Resource Adapters** → **Resource adapters**.
2. View the built-in resources. Click **Preferences**, and select **Show built-in resources**.
3. Click **WebSphere Relational Resource Adapter** → **CMP connection factories**

Name

Specifies a list of the display names for the resources.

Data type String

JNDI Name

Specifies the JNDI name of the resource.

Data type String

Description

Specifies a description for the resource.

Data type String

Category

Specifies a category string which can be used to classify or group the resource.

Data type String

CMP connection factory settings

Use this page to view the settings of a connection factory that is used by a container-managed persistence (CMP) bean to access any database server. This connection factory is only in ″read″ mode. It cannot be modified or deleted.

To view this administrative console page:

1. Click **Resources** → **Resource Adapters** → **Resource adapters**.
2. Click **Preferences**, and select **Show built-in resources**.
3. Click **WebSphere Relational Resource Adapter** → **CMP Connection Factories** → ***connection_factory***.

Name:

Specifies the display name for the resource.

Data type String

JNDI name:

Specifies the JNDI name of the resource.

Data type String

Description:

Specifies a description for the resource.

Data type String

Category:

Specifies a category string which can be used to classify or group the resource.

Data type String

Authentication preference:

Specifies which of the authentication mechanisms that are defined for the corresponding resource adapter applies to this connection factory. This property is deprecated starting with version 6.0.

For example, if two authentication mechanism entries are defined for a resource adapter (*KerbV5* and *Basic Password*), this specifies one of those two types. If the authentication mechanism preference specified is not an authentication mechanism available on the corresponding resource adapter, it is ignored.

Data type String

Component-managed authentication alias:

References authentication data for component-managed signon to the resource.

Data type Drop-down list

Container-managed authentication alias:

References authentication data for container-managed signon to the resource.

Data type Drop-down list

Assembling data access applications

When you assemble enterprise bean code into files that can be deployed onto an application server, you configure properties that define how the application accesses an enterprise information system (EIS), such as a database.

Before you begin

This topic assumes that you have created an enterprise application containing an EJB module that must transact with a database.

About this task

A data access application uses resources, such as data sources or connection factories, to connect with a database.

An application component uses a *connection factory* to access a connection instance, which the component then uses to connect to the underlying enterprise information system (EIS). Examples of connections include database connections, Java Message Service connections, and SAP R/3 connections.

During application assembly you perform activities that enable the application to use these resources. The process typically requires an assembly tool.

1. Identify the logical names that are used by the EJB module to reference application resources. These logical names are called *resource references*.

   For further explanation, consult “The benefits of using resource references” on page 986 topic.
2. Start an assembly tool.
3. If you have not done so already, configure the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules. Ensure that **Java EE** capability is enabled.
4. Define mapping and security properties for the resource references. This process includes the following activities:
   a. Bind the resource references to the application resources that provide database connectivity.

      See the “Data source lookups for enterprise beans and Web modules” on page 989 topic for more information on the concept of binding. At deployment time you can alter your bindings if necessary.
   b. For each resource define an authentication type, which is the security configuration through which database connections are granted. There are two authentication types:

      Component-managed
      The enterprise bean code performs EIS signon for data source or connection factory connections.

      Container-managed
      The product performs EIS signon.

      Consult the J2EE connector security topic for detailed reference on resource authentication.
5. Configure access intent assembly settings for your enterprise beans.
   a. Right-click your EJB module in a Project Explorer view and click **Open With > Deployment Descriptor Editor**.
   b. In an EJB Deployment Descriptor editor, select the **Access** tab.
   c. Under **Isolation Level**, click **Add**.
   d. Select the isolation level, enterprise beans, and method elements. For information on isolation levels, press **F1**.
   e. Click **Finish**.
6. Map enterprise beans to database tables.

Results

Files for the updated application are shown in the Project Explorer view.

What to do next

After testing your application, you are ready to deploy your application to an application server.

Creating or changing a resource reference

A resource reference supports application access to a resource (such as a data source, URL, or mail provider) using a logical name rather than the actual name in the runtime environment. This capability eliminates the necessity to alter application code when you change the resource runtime configurations.

Before you begin

This topic guides you through updating the resource references of an enterprise application that you assembled previously. The topic Assembling applications details the assembly procedure.

About this task

Resource references are declared in the deployment descriptor by the application provider. At some point in the application deployment process, you must bind the resource reference to the actual name of the resource in the run time environment. When you create a connection factory or data source in the application server, the application server provides a JNDI name that a component can use to access that connection factory or data source. The application server uses an indirect name with the java:comp/env prefix. For example:

- When you create a data source, the default JNDI name is set to jdbc/*data_source_name*.
- When you create a connection factory, its default name is eis/*j2c_connection_factory_name*.

If you override these values by specifying your own, retain the java:comp/env prefix. An indirect JNDI name allows the connection management infrastructure to access to any data from the resource reference that is associated with the application. This allows you to better manage resources based on the settings for authentication, isolation level, sharing scope, and resolution control.

This topic describes how to update the resource references of an enterprise application using an assembly tool. After you define the resource reference, you can perform an indirect JNDI lookup using the java:comp/env context.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules.
3. Import the enterprise application (EAR file) that you want to change into the EJB project.
4. Display the resource references for the type of module:
   - If an enterprise bean uses the resource reference:
     a. Expand the name of the EAR file.
     b. Expand **EJB Modules**.
     c. Expand the EJB module wanted.
     d. Expand the section for the appropriate type of enterprise bean (**Session Beans** or **Entity Beans**).
     e. Expand the enterprise bean.
   - If a servlet uses the resource reference:
     a. Expand the name of the EAR file.
     b. Expand **Web Modules**.
     c. Expand the Web module wanted.
   - If an application client uses the resource reference:
     a. Expand the name of the EAR file.
     b. Expand **Application Clients**.
     c. Expand the application client module wanted.
5. Right-click the module whose resource references you want to change and click **Open With > Deployment Descriptor Editor**.
6. For servlets and application clients, click **Add**. For EJB modules, select the particular bean and click **Add**.
7. Select the resource reference option and click **Next**.
8. Specify the settings for the resource reference, and click **Finish**.
9. Optional: Select the **References** tab and, under **WebSphere Extensions**, select an isolation level. If you choose to forego this step, the isolation level defaults to TRANSACTION_NONE.
10. Optional: Under **WebSphere Bindings**, specify a JNDI name. If you choose to forego this step you can set (or override) the binding when the application is deployed.
11. Close the deployment descriptor editor and save your changes.

Results

Files for the updated module are shown in the Project Explorer view.

What to do next

Verify the contents of the updated enterprise application in the Project Explorer view. Then, deploy your enterprise application.

You can generate EJB deployment code and deploy an EJB module to a target server in one step. In the Project Explorer view, right-click on the EJB project and click **Deploy**. See also the article "Deploying EJB modules" on page 215.

Assembling resource adapter (connector) modules

A resource adapter archive (RAR) file contains code that implements a library for connecting with a backend Enterprise Information System (EIS).

Before you begin

This topic assumes that you have created and unit tested a resource adapter RAR file that you want to assemble in an enterprise application and deploy onto an application server.

A Resource Adapter Archive (RAR) file is a Java archive (JAR) file used to package a resource adapter for the Java 2 Connector (J2C) Architecture for the product.

A RAR file can contain the following:
- Enterprise information system (EIS) supplied resource adapter implementation code in the form of JAR files or other runnable components, such as dynamic link lists.
- Utility classes.
- Static documents, such as HTML files, images, and sound files.

The standard file extension of a RAR file is *.rar*.

About this task

In an assembly tool, RAR files are called *connectors* and assembled resource adapters are called *connector modules*.

A *connector* is a Java Platform, Enterprise Edition (Java EE) component that provides access to Enterprise Information Systems (EIS), and must comply with the Java EE Connector Architecture (JCA). An example of an EIS is a transaction manager such as the Customer Information Control System (CICS).

You might see the terms resource adapter *modules*, resource adapter *connectors* and resource adapter *archive files* used interchangeably.

Use an assembly tool to assemble a *connector* in either of the following ways:
- Import an existing RAR file.
- Create a new connector module.

For information on assembling connectors, refer to the online documentation or the information center for your assembly tool.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that **Java EE** and **EJB** capabilities are enabled.

3. Migrate RAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an assembly tool. To migrate files, import your RAR files to the assembly tool.
4. Create a new connector module.

Results

A connector project is migrated or created. Files for the connector project are shown in the Project Explorer view under **Enterprise Applications** and **Connector Projects**.

What to do next

After creating a connector project, you can edit the connector deployment descriptor if default properties are not sufficient. In the Connector Deployment Descriptor editor, you can view and edit source code.

After assembling the connector project, deploy the module or its application onto a server. After deployment, to ensure that the connector module finds the classes and resources that it needs, check the **Classpath** setting for the RAR on the console Resource adapter settings page.

Migrating applications to use data sources of the current Java EE Connector Architecture (JCA)

Migrate your applications that use Version 4 data sources, or data sources (WebSphere Application Server V4), to use data sources that support more advanced connection management features, such as connection sharing.

About this task

To use the connection management infrastructure in the application server, you must package your application as a Java EE 1.3 or later application. This process involves repackaging your Web modules to the 2.3 specification and your EJB modules to the 2.1 specification before installing them onto WebSphere Application Server.

Applications left at the Java EE 1.2 level will continue to run fine using the connection management support that was available at V4.0; simply create the JDBC provider and data source, and install the 4.0 application as-is. If you choose to repackage your application for version 6.0 or above you can not use a Version 4.0 data source. You must use the other data source option, which supports applications coded to the Java EE 1.3 specifications, at minimum.

- Convert a 2.2 Web module to a 2.3 Web module
  1. Open an assembly tool.
  2. Create a new Web module by selecting **File > New > Web Module**.
  3. Add any required class files to the new module.
     a. Expand the **Files** portion of the tree.
     b. Right-click **Class Files** and select **Add Files**.
     c. In the Add Files window, click **Browse**.
     d. Navigate to your WebSphere Application Server 4.0 EAR file and click **Select**.
     e. In the upper left pane of the Add Files window, navigate to your WAR file and expand the `WEB-INF` and `classes` directories.
     f. Select each of the directories and files in the classes directory and click **Add**.
     g. After you add all of the required class files, click **OK**.
  4. Add any required JAR files to the new module.
     a. Expand the **Files** portion of the tree.
     b. Right-click **Jar Files** and select **Add Files**.

c. Navigate to your WebSphere 4.0 EAR file and click **Select**.
d. In the upper left pane of the Add Files window, navigate to your WAR file and expand the `WEB-INF` and `lib` directories.
e. Select each JAR file and click **Add**.
f. After you add all of the required JAR files, click**OK**.

5. Add any required resource files, such as HTML files, JSP files, GIFs, and so on, to the new module.
   a. Expand the **Files** portion of the tree.
   b. Right-click **Resource Files** and select **Add Files**.
   c. Navigate to your WebSphere Application Server 4.0 EAR file and click **Select**.
   d. In the upper left pane of the Add Files window, navigate to your WAR file.
   e. Select each of the directories and files in the WAR file, excluding `META-INF` and `WEB-INF`, and click **Add**.
   f. After you add all of the required resource files, click**OK**.
6. Import your Web components.
   a. Right-click **Web Components** and select **Import**.
   b. In the Import Components window click **Browse**.
   c. Navigate to your WebSphere Application Server 4.0 EAR file and click **Open**.
   d. In the left top pane of the **Import Components** window, highlight the WAR file that you are migrating.
   e. Highlight each of the components that display in the right top pane and click **Add**.
   f. When all of your Web components display in the Selected Components pane of the window, click **OK**.
   g. Verify that your Web components are correctly imported under the Web Components section of your new Web module.
7. Add servlet mappings for each of your Web components.
   a. Right-click **Servlet Mappings** and select **New**.
   b. Identify a URL pattern for the Web component.
   c. Select the web component from the Servlet drop-down box.
   d. Click **OK**.
8. Add any necessary resource references by following the instructions in the Creating a resource reference article in the information center.
9. Add any other Web module properties that are required. Click **Help** for a description of the settings.
10. **Save** the Web module.

- Convert a 1.1 EJB module to a 2.1 EJB module (or later)
  1. Open an assembly tool.
  2. Create a new EJB Module by selecting **File > New > EJB Module**.
  3. Add any required class files to the new module.
     a. Right-click **Files object** and select **Add Files**.
     b. In the Add Files window click **Browse**.
     c. Navigate to your WebSphere Application Server 4.0 EAR file and click **Select**.
     d. In the upper left pane of the Add Files window, navigate to your enterprise bean JAR file.
     e. Select each of the directories and class files and click **Add**.
     f. After you add all of the required class files, click **OK**
  4. Create your session beans and entity beans. To find help on this subject, see the information center article Migrating enterprise bean code to the supported specification.

5. Add any necessary resource references by following the instructions in the Creating a resource reference article in the information center.
6. Add any other EJB module properties that are required. Click **Help** for a description of the settings.
7. **Save** the EJB module.
8. Generate the deployed code for the EJB module by clicking **File > Generate Code for Deployment**.
9. Fill in the appropriate fields and click **Generate Now**.

- Add the EJB modules and Web modules to an EAR file
  1. Open an assembly tool.
  2. Create a new Application by selecting **File > New > Application**.
  3. Add each of your EJB modules.
     a. Right-click **EJB Modules** and select **Import**.
     b. Navigate to your converted EJB module and click **Open**.
     c. Click **OK**.
  4. Add each of your Web modules.
     a. Right-click **Web Modules** and select **Import**.
     b. Navigate to your converted Web module and click **Open**.
     c. Fill in a **Context root** and click **OK**.
  5. Identify any other application properties. Click **Help** for a description of the settings.
  6. Save the EAR file.
- Install the application on the application server.
  1. Install the application following the instructions in the topic on installing a new application, and bind the resource references to the data source that you created.
  2. Perform the necessary administrative task of creating a JDBC provider and a data source object following the instructions in the topic on creating a JDBC provider and data source.

Connection considerations when migrating servlets, JavaServer Pages, or enterprise session beans

If you plan to upgrade to WebSphere Application Server Version 6.x, and migrate applications from version 1.2 of the Java 2 Platform, Enterprise Edition (J2EE) specification to a later version, such as 1.4 or Java Platform, Enterprise Edition (Java EE), be aware that the product allocates shareable and unshareable connections differently for post-version 1.2 application components. For some applications, that difference can result in performance degradation.

Adverse behavior changes

Because WebSphere Application Server provides backward compatibility with application modules coded to the J2EE 1.2 specification, you can continue to use Version 4 style data sources when you migrate to Application Server Version 6.x. As long as you configure Version 4 data sources *only* for J2EE 1.2 modules, the behavior of your data access application components does not change.

If you are adopting a later version of the J2EE specification along with your migration to Application Server Version 6.x, however, the behavior of your data access components can change. Specifically, this risk applies to applications that include servlets, JavaServer Page (JSP) files, or enterprise session beans that run inside local transactions over shareable connections. A behavior change in the data access components can adversely affect the use of connections in such applications.

This change affects all applications that contain the following methods:
- `RequestDispatcher.include()`
- `RequestDispatcher.forward()`
- JSP includes (`<jsp:include>`)

Symptoms of the problem include:
- Session hang
- Session timeout
- Running out of connections

Note: You can also experience these symptoms with applications that contain the components and methods described previously if you are upgrading from J2EE 1.2 modules *within* Application Server Version 6.x.

The switch in allocating shareable and unshareable connections

For J2EE 1.2 modules using Version 4 data sources, WebSphere Application Server issues non-sharable connections to JSP files, servlets, and enterprise session beans. All of the other application components are issued shareable connections. However, for J2EE 1.3 and later modules, Application Server issues shareable connections to *all* logically named resources (resources bound to individual references) unless you specify the connections as unshareable in the individual resource-references. Using shareable connections in this context has the following effects:
- All connections that are received and used outside the scope of a user transaction are *not* returned to the free connection pool until the encapsulating method returns, even when the connection handle issues a `close()` call.
- All connections that are received and used outside the scope of a user transaction are *not* shared with other component instances (that is, other servlets, JSP files, or enterprise beans). For example, session bean 1 gets a connection and then calls session bean 2 that also gets a connection. Even if all properties are identical, each session bean receives its own connection.

If you do not anticipate this change in the connection behavior, the way you structure your application code can lead to excessive connection use, particularly in the cases of JSP includes, session beans that run inside local transactions over shareable connections, `RequestDispatcher.include()` routines, `RequestDispatcher.forward()` routines, or calls from these methods to other components. Consequently, you can experience session hang, session timeout, or connection deficiency.

Example scenario

Servlet A gets a connection, completes the work, commits the connection, and calls **`close()`** on the connection. Next, servlet A calls the `RequestDispatcher.include()` to include servlet B, which performs the same steps as servlet A. Because the servlet A connection does not return to the free pool until it returns from the current method, two connections are now busy. In this way, more connections might be in use than you intended in your application. If these connections are not accounted for in the **Max Connections** setting on the connection pool, this behavior might cause a lack of connections in the pool, which results in `ConnectionWaitTimeOut` exceptions. If the **connection wait timeout** is not enabled, or if the **connection wait timeout** is set to a large number, these threads might appear to hang because they are waiting for connections that are never returned to the pool. Threads waiting for new connections do not return the ones they are currently using if new connections are not available.

Resolution

To resolve these problems:
1. Use unshared connections.

   If you use an unshared connection and are not in a user transaction, the connection is returned to the free pool when you issue a `close()` call (assuming you commit or roll back the connection).
2. Increase the maximum number of connections.

   To calculate the number of required connections, multiply the number of configured threads by the deepest level of component call nesting (for those calls that use connections). See the Examples section for a description of call nesting.

Deploying data access applications

Frequently, deploying data access applications involves more than installing your WAR or EAR file onto a server. Deployment can include tasks for configuring your application to use the data access resources of the server and overall run-time environment.

Before you begin

You can only deploy application code that is assembled into the appropriate modules. The topic "Assembling data access applications" on page 1096 provides guidelines for this process.

About this task

Perform the following steps if your application requires access to a relational database (RDB). If your application requires access to a different type of enterprise information system (EIS), such as an object-oriented database or the Customer Information Control System (CICS), consult the topics "Resource adapters" on page 949 and "Accessing data using Java EE Connector Architecture connectors" on page 997.

1. If your RDB configuration does not already exist:
   a. Create a database to hold the data.
   b. Create tables required by your application.
      If your application uses CMP entity beans to access the data
      You can create the tables using the data definition language (DDL) generated from the enterprise bean configuration. For more information, see Recreating database tables from the exported table data definition language.
      If your application uses BMP entity beans, or *does not* **use entity beans**
      You must use your database server interfaces to create the tables.

      You can also use the EJB to RDB Mapping wizard of an assembly tool to create your database tables for either type of entity bean. Select the top-down mapping option in the wizard. Keep in mind, however, that this option does not give you direct control in naming the RDB elements or choosing column types. Additionally, because the top-down process is automatic, it might not provide mappings to reflect the precise relationships that you intend.

      If you use Rational Application Developer, consult the information center about the mapping wizard. To learn about all of your assembly tool options, consult the Assembly tools article in this information center.
   c. Check Data source minimum required settings, by vendor to see any database vendor requirements for connecting to an application server.
2. If necessary, map your entity beans to the database tables through the meet-in-the-middle mapping option of an assembly tool. This step is necessary only if you did not create your database schema through the top-down mapping option, did not generate your mapping relationships through bottom-up mapping, or did not generate mappings during the application assembly process. For information on the top-down mapping option refer to the information center for Rational Application Developer.
3. Install your application onto the application server. Consult Installing enterprise application files. When you install the application, you can alter data access settings that were made during application assembly, or set them for the first time if they were omitted from the assembly process. These settings include resource bindings and resource authentication aliases, which are addressed in the following substeps:
   a. Bind application resource references to the data sources, or other resource objects, that provide database connectivity. For details on the concept of binding, see the "Data source lookups for enterprise beans and Web modules" on page 989 topic.

      Note: After deployment, you can use the WebSphere Application Server administrative console to alter resource bindings. Click **Applications** → **Application Types** → **Webphere enterprise applications** → ***application_name***, and select the link to the appropriate mapping page. For

example, if you want to alter the binding of an EJB module resource, you might click **Map data sources for all 2.x CMP beans** . For a Web module resource, click **Resource references**.

b. Define authentication alias data for resources that must be authenticated with the backend through *container-managed* authorization. In this security configuration, WebSphere Application Server performs EIS signon for data source or connection factory connections. Consult the J2EE connector security topic for detailed reference on resource authentication.

4. Start the deployed application files using the administrative console , the wsadmin startApplication command, or your own Java program.
5. Save the changes to your administrative configuration.
6. Test the application. For example, point a Web browser at the URL for a deployed application and examine the performance of the application.

What to do next

If the application does not perform as desired, update the application, then save and test it again.

Available resources

Use this page to select configured resources that you want to bind to the resource references of the enterprise beans or web modules in your application.

To view this administrative console page:

1. Click **Applications** → **Application Types** → **Websphere enterprise applications** → ***application_name***.
2. Click the link for any of these resource configuration pages:
   - **Resource references**
   - **Map data sources for all 2.x CMP beans**
   - **Provide default data source mapping for modules containing 2.x entity beans**
3. Locate the table row of the EJB or web module that you want to map to a different resource.
4. Within the row, locate the JNDI name of the resource that is currently bound to the EJB or web module.
5. Click **Browse**.

   You now see **Available resources**.

Each table row corresponds to a resource that you can bind to your enterprise bean or web module.

Select

Select the resource that you want to bind to the resource reference of your module.

JNDI name

The Java Naming and Directory Interface (JNDI) name of the resource that you want to bind to the resource reference of your module.

Data type String

Scope

The scope of the resource. Note that this administrative console page displays only resources that are configured for a scope at which your application operates.

Description

The text description of the resource.

Map data sources for all 1.x CMP beans

Use this page to designate how the container-managed persistence (CMP) 1.x beans of an application map to data sources that are available to the application.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Map data sources for all 1.x CMP beans**.

Guidelines for using this administrative console page:

- The table depicts the 1.x CMP bean contents of your application.
- Each table row corresponds to a CMP bean within a specific EJB module. A row shows the JNDI name of the data source mapping target of the bean *only* if you bound them together during application assembly or installation. For every data source that is displayed, you see the corresponding security configuration.
- To set your mappings:
  1. Select a row. Be aware that if you check multiple rows on this page, the data source mapping target that you select in step 2 applies to all of those CMP beans.
  2. Click **Browse** to select a data source from the new page that is displayed, the Available Resources page. The Available Resources page shows all data sources that are available mapping targets for your CMP beans.
  3. Click **Apply**. The console displays the 1.x CMP bean data sources page again. In the rows that you previously selected, you now see the JNDI name of the new resource mapping target.
  4. *Before* you click **OK** to save your new configuration, set the security parameters for the data source. Use the following steps.
- To specify data source security settings:
  1. Select one or more rows in the table.
  2. Type in a user name and password that comprise the authentication alias for signing on to the data source. If these entries are not listed in the application Java Platform, Enterprise Edition (Java EE) Connector (J2C) authentication data list, you must input them into the list after saving your settings on this page. Read the information center topic on managing Java EE Connector Architecture authentication data entries for more information.
  3. Click **Apply** that immediately follows the user name and password input fields.
- Repeat all of the previous steps as necessary.
- Click **OK** to save your settings.

Select

Select the check boxes of the rows that you want to edit.

EJB

The name of an enterprise bean in the application.

EJB Module

The name of the module that contains the enterprise bean.

URI

Specifies location of the module relative to the root of the application EAR file.

JNDI name

The Java Naming and Directory Interface (JNDI) name of the data source that is configured for the enterprise bean.

Data type String

User name
The user name and password that comprise the authentication alias for securing the data source.

Map default data sources for modules containing 1.x entity beans
Use this page to set the default data source mapping for EJB modules that contain 1.x container-managed persistence (CMP) beans. Unless you configure individual data sources for your 1.x CMP beans, this default mapping applies to all beans within the module.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Map default data sources for modules containing 1.x entity beans**.

Guidelines for using this administrative console page:

- The page displays a table that depicts the EJB modules in your application that contain 1.x CMP beans.
- Each table row corresponds to a module. A row shows the JNDI name of the data source mapping target of the EJB module *only* if you bound them together during application assembly. For every data source that is displayed, you see the corresponding security configuration.
- To set your default data source mappings:
  1. Select a row. Be aware that if you check multiple rows on this page, the data source mapping target that you select in step 2 applies to all of those EJB modules.
  2. Click **Browse** to select a data source from the new page that is displayed, the Available Resources page. The Available Resources page shows all data sources that are available mapping targets for your EJB modules.
  3. Click **Apply**. The console displays the 1.x entity bean data sources page again. In the rows that you previously selected, you now see the JNDI name of the new resource mapping target.
  4. *Before* you click **OK** to save your new configuration, set the security parameters for the data source. Use the following steps.
- To specify security settings for the default data source:
  1. Select a row. Be aware that if you check multiple rows on this page, the security settings that you select later apply to all of those data sources.
  2. Type in a user name and password that comprise the authentication alias for signing on to the data source. If these entries are not listed in the application Java Platform, Enterprise Edition (Java EE) Connector (J2C) authentication data list, you must input them into the list after saving your settings on this page. Read the information center topic on managing Java EE Connector Architecture authentication data entries for more information.
  3. Click **Apply** that immediately follows the user name and password input fields.
- Repeat all of the previous steps as necessary.
- Click **OK** to save your work.

Select
Select the check boxes of the rows that you want to edit.

EJB Module
The name of the module that contains the 1.x enterprise beans.

URI
Specifies location of the module relative to the root of the application EAR file.

JNDI name
The Java Naming and Directory Interface (JNDI) name of the default data source for the EJB module.

Data type String

User name
The user name and password that comprise the authentication alias for securing the data source.

Map data sources for all 2.x CMP beans settings
Use this page to map container-managed persistence (CMP) 2.x beans of an application to data sources that are available to the application.

To view this administrative console page, click **Applications** → **Application Types** → **Websphere enterprise applications** → ***application_name*** → **Map data sources for all 2.x CMP beans**.

Each table row corresponds to a CMP bean within a specific EJB module. A row shows the JNDI name of the data source mapping target of the bean only if you bound them together during application assembly. For every data source that is displayed, you see the corresponding security configuration.

Set Multiple JNDI names
Specify the Java Naming and Directory Interface (JNDI) name for multiple EJB modules. Select one or more EJB modules from the table, and select a JNDI name from this list to configure the EJB modules with that JNDI name.

Data type Drop-down list

Set Authorization Type
Specify the authorization type for securing the data source. Select one or more EJB modules from the table to set the authorization type.

Select either **Container** or **Application** from the displayed list. Container-managed authorization indicates that WebSphere Application Server performs signon to the data source. Application-managed authorization indicates that the enterprise bean code performs signon.

Modify Resource Authentication Method
Specify the authorization type and the authentication method for securing the data source. Select one or more EJB modules from the table to modify the resource authentication method.

You can choose between the following authentication methods:

- **None**:
  1. Determine which data source configurations to designate with no authentication method.
  2. Select the appropriate table rows.
  3. Select **None** from the list of authentication method options that precede the table.
  4. Click **Apply**.
- **Use default method (many-to-one mapping)**:
  1. Determine which data source configurations to designate with the WebSphere Application Server DefaultPrincipalMapping login configuration. Apply this option to each data source individually if you want to designate different authentication data aliases. See the information center topic on J2EE Connector security for more information on the default mapping configuration.
  2. Select the appropriate table rows.
  3. Select **Use default method (many-to-one mapping)** from the list of authentication method options that precede the table.
  4. Select an authentication data entry or alias from the list.
  5. Click **Apply**.
- **Use Kerberos authentication**: Specifies to use the Kerberos authentication method.
  1. Ensure that you have configured the Kerberos authentication mechanism in the application server.

2. Select the appropriate table row.
3. Select **Use Kerberos authentication** from the list of authentication method options that precede the table.
4. Select an application login configuration from the list.
5. Click **Apply**.
6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

The application server will attempt to verify that you are connecting to the correct type of database when you select this option.

- **Use trusted connections (one-to-one mapping)**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Ensure that the database to which the modules will connect is configured for trusted connections.
  4. Select **Use trusted connections (one-to-one mapping)** from the list of authentication method options that precede the table.
  5. Select an application login configuration from the list.
  6. Click **Apply**.

  The application server will attempt to verify that you are connecting to the correct type of database when you select this option.
- **Custom login configuration**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Select **Use custom login configuration** from the list of authentication method options that precede the table.
  4. Select an application login configuration from the list.
  5. Click **Apply**.
  6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

Select
Select the check boxes of the rows that you want to edit.

EJB
The name of an enterprise bean in the application.

EJB Module
The name of the module that contains the enterprise bean.

URI
Specifies location of the module relative to the root of the application EAR file.

Target resource JNDI name
Specifies the resource to which the CMP bean is bound.

Resource authorization
Specifies the current setting for the resource authorization type.

Modify this setting with **Set authorization type**.

Map data sources for all 2.x CMP beans

Use this page to set the default data source mapping for EJB modules that contain 2.x container-managed persistence (CMP) beans. Unless you configure individual data sources for your 2.x CMP beans, this default mapping applies to all beans within the module.

To view this administrative console panel, click **Applications** → **Application Types** → **Websphere enterprise applications** → ***application_name*** → **Map data sources for all 2.x CMP beans** .

This panel displays a table that depicts the EJB modules in your application that contain 2.x CMP beans. Each table row corresponds to a module. A row shows the JNDI name of the data source mapping target of the EJB module only if you bound them together during application assembly. For every data source that is displayed, you see the corresponding security configuration.

Set Multiple JNDI Names

Specifies the JNDI name to bind to one or more modules. Select one or more modules, click **Set Multiple JNDI Names**, and select the JNDI name for the resource to which you would like to bind the module.

Set Authorization Type

Specifies the authorization type that you to use for the modules. Select one or more modules, click **Set Authorization Type**, and select the authorization type.

You can choose:

- Per application - indicates that the enterprise bean code performs signon.
- Container - indicates that the application server performs signon to the data source.

Modify Resource Authentication Method

Specifies the resource authentication method for the modules that you have configured with container-managed authorization. Select one or more modules, click **Modify Resource Authentication Method**, and select the authentication method.

You can choose between the following authentication methods:

- **None**:
  1. Determine which data source configurations to designate with no authentication method.
  2. Select the appropriate table rows.
  3. Select **None** from the list of authentication method options that precede the table.
  4. Click **Apply**.
- **Use default method (many-to-one mapping)**:
  1. Determine which data source configurations to designate with the WebSphere Application Server DefaultPrincipalMapping login configuration. Apply this option to each data source individually if you want to designate different authentication data aliases. See the information center topic on J2EE Connector security for more information on the default mapping configuration.
  2. Select the appropriate table rows.
  3. Select **Use default method (many-to-one mapping)** from the list of authentication method options that precede the table.
  4. Select an authentication data entry or alias from the list.
  5. Click **Apply**.
- **Use Kerberos authentication**: Specifies to use the Kerberos authentication method.
  1. Ensure that you have configured the Kerberos authentication mechanism in the application server.
  2. Select the appropriate table row.
  3. Select **Use Kerberos authentication** from the list of authentication method options that precede the table.

4. Select an application login configuration from the list.
5. Click **Apply**.
6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

The application server will attempt to verify that you are connecting to the correct type of database when you select this option.

- **Use trusted connections (one-to-one mapping)**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Ensure that the database to which the modules will connect is configured for trusted connections.
  4. Select **Use trusted connections (one-to-one mapping)** from the list of authentication method options that precede the table.
  5. Select an application login configuration from the list.
  6. Click **Apply**.

  The application server will attempt to verify that you are connecting to the correct type of database when you select this option.
- **Custom login configuration**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Select **Use custom login configuration** from the list of authentication method options that precede the table.
  4. Select an application login configuration from the list.
  5. Click **Apply**.
  6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

Select
Select the check boxes of the rows you want to edit.

EJB Module
Specifies the name of the module that contains the 2.x enterprise beans.

URI
Specifies location of the module relative to the root of the application EAR file.

JNDI name
Specifies the Java Naming and Directory Interface (JNDI) name of the default data source for the EJB module.

Data type String

Resource authorization
Specifies the authorization type and the authentication method for securing the data source.

Extended Datasource Properties
When selected, you will be directed to a panel on which you can specify extended properties that the module can use for the DB2 data source.

The application server will attempt to verify that you are connecting to the correct type of database when you select this option.

Task overview: Storing and retrieving persistent data with the Java Persistence API (JPA)

The Java Persistence API (JPA) for the application server defines the management of persistence and object/relational mapping within Java Enterprise Edition (Java EE) and Java Standard Edition (Java SE) environments.

About this task

The Java Persistence API (JPA) represents a simplification of the persistence programming model. JPA functions within the Java EE specification for Enterprise Java Beans (EJB) 3.0 requirements, managing persistence and object/relational mapping. The JPA specification defines the object/relational mapping within its own guidelines instead of relying on vendor-specific mapping implementations. These features make applications that use JPA easier to implement and manage.

In a nutshell, JPA combines the best features from previous persistence mechanisms such as Java Database Connectivity (JDBC) APIs, Object Relational Mapping (ORM) frameworks, and Java Data Objects (JDO). Creating entities under JPA is as simple as Plain Old Java Objects (POJOs). JPA supports the features provided by JDBC without requiring the knowledge of the specific programming models defined by the various JDBC implementations. Like object-relational software and object databases, JPA allows the use of advanced object-oriented concepts such as inheritance. JPA avoids vendor lock-in because it does not rely on a strict specification like JDO and EJB 2.x entities.

The JPA implementation does not mandate that you migrate existing applications. Existing EJB 2.x Container Managed Persistence applications continue to execute without changes. JPA may not be ideal for every application, however, for many applications it provides a better alternative to other persistence implementations.

Use the topics listed below for detailed information about aspects of JPA:

- Learn about **persistence and JPA**
- Learn about the differences in **Apache OpenJPA and JPA for WebSphere Application Server**
- Find information on **developing JPA applications for a Java EE environment** or **developing JPA applications for a Java SE environment**
- A guide to **configuring persistence providers**
- Set up a datasource by **configuring JDBC for use with JPA for WebSphere Application Server**
- Learn about **Configuring caching to improve performance**
- Monitor your applications by **logging your application's behavior with JPA for WebSphere Application Server**
- Troubleshoot JPA problems: **Troubleshooting JPA**
- Learn about **JPA Access Intent**

What to do next

Product support for JPA

The implementation of Java Persistence API for the application server can be used on all platforms that are supported for the application server, including iSeries and z/OS. Java Persistence API for the application server functions with all databases supported in WebSphere Application Server. In addition to these, Java Persistence API for WebSphere Application Server can support databases that are supported by the OpenJPA implementation of JPA.

Note: Databases supported by OpenJPA but not supported by the product have not been tested extensively by IBM and might contain unknown compatibility issues. For a list of supported databases, refer to the OpenJPA user guide.

Additional information

For additional information about OpenJPA, see the OpenJPA User Guide. For information about Java Persistence API specifications, see the link listed below. The information resides on both IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

- For information on the Java Persistence API specification, consult: http://java.sun.com/javaee/technologies/persistence.jsp.

Related tasks

Configuring Persistence Provider support in the application server
Persistence providers are implementations of the Java Persistence API (JPA) specification and can be deployed in the Java EE compliant application server that supports JPA persistence.

Associating persistence units and data sources
Java Persistence API (JPA) applications allow you to specify the underlying data source used by the persistence provider to access the database.

Configuring OpenJPA caching to improve performance
The OpenJPA implementation allows users the option of storing frequently used data in the memory to improve performance. OpenJPA provides concurrent data and concurrent query caches that allow applications to save persistent object data and query results in memory to share among threads and for use in future queries.

Troubleshooting Java Persistence API (JPA) applications
Use this information to find various known problems with JPA applications.

Related information

../../../../com.ibm.websphere.wim.doc.en/supporteddatabases.html

Java Persistence API (JPA) Architecture

Data persistence, the ability to maintain data between application executions, is vital to enterprise applications because the required access to relational databases. Applications that are developed for this environment must manage persistence themselves or make use of third-party solutions to handle database updates and retrievals with persistence. The Java Persistence API (JPA) provides a mechanism for managing persistence and object-relational mapping and functions for the EJB 3.0 specifications.

The JPA specification defines the object-relational mapping internally, rather than relying on vendor-specific mapping implementations. JPA is based on the Java programming model that applies to Java EE environments, but JPA can function within a Java SE environment for testing application functions.

JPA represents a simplification of the persistence programming model. The JPA specification explicitly defines the object-relational mapping, rather than relying on vendor-specific mapping implementations. JPA standardizes the important task of object-relational mapping by using annotations or XML to map objects into one or more tables of a database. To further simplify the persistence programming model:

- The EntityManager API can persist, update, retrieve, or remove objects from a database
- The EntityManager API and object-relational mapping meta-data handle most of the database operations without requiring you to write JDBC or SQL code to maintain persistence
- JPA provides a query language, extending the independent EJB querying language (also known as JPQL), that you can use to retrieve objects without writing SQL queries specific to the database you are working with.

JPA is designed to operate both inside and outside of a Java Enterprise Edition (Java EE) container. When you run JPA inside a container, the applications can use the container to manage the persistence context. If there is no container to manage JPA, the application must handle the persistence context management itself. Applications that are designed for container-managed persistence do not require as much code implementation to handle persistence, but these applications cannot be used outside of a container. Applications that manage their own persistence can function in a container environment or a Java SE environment.

Elements of a JPA Persistence Provider

Java EE containers that support the EJB 3.0 programming model must support a JPA implementation, also called a persistence provider. A JPA persistence provider uses the following elements to allow for easier persistence management in an EJB 3.0 environment:

- **Persistence unit:** consists of the declarative meta-data that describes the relationship of entity class objects to a relational database. The EntityManagerFactory uses this data to create a persistence context that can be accessed through the EntityManager.
- **EntityManagerFactory:** used to create an EntityManager for database interactions. The application server containers typically supply this function, but the EntityManagerFactory is required if you are using JPA application-managed persistence.
- **Persistence context:** defines the set of active instances that the application is manipulating currently. The persistence context can be created manually or through injection.
- **EntityManager:** the resource manager that maintains the active collection of entity objects that are being used by the application. The EntityManager handles the database interaction and meta-data for object-relational mappings. An instance of an EntityManager represents a persistence context. An application in a container can obtain the EntityManager through injection into the application or by looking it up in the Java component name-space. If the application manages its persistence, the EntityManager is obtained from the EntityManagerFactory.

  Note: Injection of the EntityManager is only supported for the following artifacts:
  - EJB 3.0 session beans
  - EJB 3.0 message-driven beans
  - Servlets, Servlet Filters, and Listeners
  - JSP tag handlers which implement interfaces javax.servlet. jsp.tagext.Tag and javax.servlet.jsp.tagext.SimpleTag
  - Java Server Faces (JSF) managed beans
  - the main class of the application client.
- **Entity objects:** a simple Java class that represents a row in a database table in its simplest form. Entities objects can be concrete classes or abstract classes. They maintain states by using properties or fields.

For more information about persistence, see the section on Java Persistence API Architecture and the section on Persistence in the Apache OpenJPA User's Guide. For more information and examples on specific elements of persistence, refer to the sections on the EntityManagerFactory, and the EntityManager in the Apache OpenJPA User's Guide.

JPA for WebSphere Application Server

JPA for WebSphere Application Server is built on the Apache OpenJPA open source project.

Apache OpenJPA and JPA for WebSphere Application Server

Apache OpenJPA is a compliant implementation of the Sun Microsystems JPA specification. Using OpenJPA as a base implementation, WebSphere Application Server employs extensions to provide additional features and utilities for WebSphere Application Server customers. Because JPA for WebSphere

Application Server is built from OpenJPA, all OpenJPA functionality, extensions and configurations are unaffected by the WebSphere Application Server extensions. Users running OpenJPA applications do not need to make any changes to use their applications in WebSphere Application Server.

JPA for WebSphere Application Server provides more than compatibility with OpenJPA. JPA for WebSphere Application Server contains a set of tools for application development and deployment. Other features of JPA for WebSphere Application Server include support for XML mapping, JPA Access Intent, enhanced tracing capabilities, command scripts and translated message files. The provider of JPA for WebSphere Application Server is `com.ibm.websphere.persistence.PersistenceProviderImpl`.

The properties for OpenJPA can be defined in one of two ways. You can either specify the property in the `persistence.xml` file or by using a Java virtual machine (JVM) command line argument on either client or server. See the following examples:

- Specify the OpenJPA property in the `persistence.xml` file.

```
<properties>
 <property name="openjpa.jdbc.SchemaFactory" value="native(ForeignKeys=true)" />
</properties>
```

- Specify the OpenJPA property using a JVM command line argument on the client or server.

```
-Dopenjpa.jdbc.SchemaFactory="native(ForeignKeys=true)"
```

For more information on OpenJPA properties, see Chapter 2 on Configuration and the Part 3 Reference sections in the OpenJPA users guide.

The Extension Properties of JPA for WebSphere Application Server may be specified with the openjpa or wsjpa prefix. You can mix the openjpa and wsjpa prefixes as you wish for a common set of properties. Exceptions to the rule are wsjpa specific configuration properties, which should use the wsjpa prefix only. In the event that a JPA for WebSphere Application Server specific property is used with the openjpa prefix, a warning message will be logged indicating that the offending property will be treated as a wsjpa property. The reverse does not hold true for the openjpa prefix. In that case, the offending property will merely be ignored.

Developing and packaging JPA applications for a Java EE environment

Containers in the application server can provide many of the necessary functions for the Java Persistence API (JPA) in a Java Enterprise Edition (Java EE) environment. The application server also provides JPA tools to assist you with developing applications in a Java EE environment.

About this task

JPA applications require different configuration techniques from applications that use container-managed persistence (CMP) or bean-managed persistence (BMP). They do not follow the typical deployment techniques that are associated with applications that implement CMP or BMP. In JPA applications, you must define a persistence unit and configure the appropriate properties to ensure that the applications can run in a Java EE environment.

The container supports all necessary injections to ensure that applications run in the Java EE environment. For example, the container can inject the @PersistenceUnit and @PersistenceContext for your applications.

1. Generate your entities classes. Depending upon your development model, you might use some or all of the JPA tools:
   - **Top-down mapping:** You start from scratch with the entity definitions and the object-relational mappings, and then you derive the database schemas from that data. If you use this approach, you are most likely concerned with creating the architecture of your object model and then writing your entity classes. These entity classes would eventually drive the creation of your database model. If you are using a top-down mapping of the object model to the relational model, develop the entity

classes and then use OpenJPA functionality to generate the database tables that are based on the entity classes. The wsmapping tool would help with this approach.

- **Bottom-up mapping:** You start with your data model, which are the database schemas, and then you work upwards to your entity classes. The wsreversemapping tool would help with this approach.
- **Meet in the middle mapping:** probably the most common development model. You have a combination of the data model and the object model partially complete. Depending on the goals and requirements, you will need to negotiate the relationships to resolve any differences. Both the wsmapping tool and the wsreversemapping tool would help with this approach.

The JPA solution for the application server provides several tools that help with developing JPA applications. Combining these tools with IBM Rational Application Developer provides a solid development environment for either Java EE or Java SE applications. Rational Application Developer includes GUI tools to insert annotations, a customized `persistence.xml` file editor, a database explorer, and other features. Another alternative is the Eclipse Dali project. More information on Rational Application Developer or the Eclipse Dali plugin can be found at their respective web sites.

2. Enhance the entity classes using the JPA enhancer tool, wsenhancer, for the application server. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. The enhancer post-processes the bytecode that is generated by the Java compiler and adds the fields and methods that are necessary to implement the persistence features. To use the wsenhancer tool, type the following at a command prompt:

   ```
   ${app_server_root}/bin/wsenhancer.sh [parameters][arguments]
   ```

   Although JPA for the application server and OpenJPA can automatically enhance the entities at run time, you will obtain better performance if you can enhance your entities when you build the application. The application will not attempt to enhance entities that are already enhanced.

3. Optional: If you are not using the development model for bottom-up mapping, generate or update your database tables automatically or by using the wsmapping tool.
   - By default, the object-relational mapping does not occur automatically, but you can configure the application server to provide that mapping with the openjpa.jdbc.SynchronizeMappings property. This property can accelerate development by automatically ensuring that the database tables match the object model. To enable automatic mapping, include the following line in the `persistence.xml` file:

     ```
     <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
     ```

     Note: To enable automatic object-relational mapping at run time, all of your persistent classes must be listed in the Java .class file, mapping file, and Java archive (JAR) file elements in XML format.
   - To manually update or generate your database tables, run the JPA mapping tool for the application server from the command line to create the tables in the database. For example:

     ```
     ${app_server_root}/bin/wsmapping.sh [parameters][arguments]
     ```

4. Optional: If you are using DB2 and want to use static SQL, run the wsdb2gen command. In order to use the wsdb2gen tool, pureQuery runtime must be installed. The wsdb2gen tool creates *persistence_unit_name*`.pdqxml` file under the same META-INF directory where your `persistence.xml` file is located. If you have multiple persistence units, wsdb2gen command must be run for each persistence unit. To use the wsdb2gen tool, type the following at a command prompt:

   ```
   ${app_server_root}/bin/wsdb2gen.sh [parameters]
   ```

5. Configure these properties.
   a. Specify the configuration options for your database. The application server manages access to data sources. You can configure the data sources, connection pooling, and JTA transaction service in the administrative console. If you have a specific data source for your application, configure the data source before you install your JPA application.
      1) Configure your data sources through the administrative console. See the topic on configuring the JDBC provider and data source for more information.

2) Specify the Java Naming and Directory Interface (JNDI) names for the <jta-data-source> and <non-jta-data-source> elements. If you use the component name space method for data source retrieval, ensure that your application defines these resource references so that you can use these JNDI names to access the data source. This configuration provides more flexibility if you need to alter the configuration for the data source. For more information on using the JNDI interface, refer to the topic on developing applications that use JNDI. For example, the `persistence.xml` file would have a entry like the following:

```
<jta-data-source>java:comp/env/jdbc/FooBarDataSourceJNDI</jta-data-source>
```

b. If you are using pureQuery, configure your data source to use pureQuery. Ensure the `pdq.jar` and `pdqmgmt.jar` files are included on the JDBC provider class path. The JPA provider implementation must be the IBM implementation of JPA provider of `com.ibm.websphere.persistence.PersistenceProviderImpl`. The OpenJPA persistence provider does not provide support for pureQuery. For more information, refer to the topic configuring a data source to use pureQuery.

6. Package the application. There are several packaging options for an application that uses JPA in a Java EE environment. Choose the packaging option that best suits the JPA usage and configuration within the modules of your application. These are some of the most common packaging options. For a definitive list of packaging options, see the Java Persistence API specification.

   Note: If you are using pureQuery, add the *persistence_unit_name*`.pdqxml` files created previous or the or *persistence_unit_name*`.pdqxml` files created in the above Step 4 to the JPA application JAR file. The files are located in same META-INF directory where your `persistence.xml` file is located.

   - For a standalone EJB module or a standalone application client module, package the EJB and application client modules in a standard JAR file. Ensure that you package the application with these conditions:
     – The JAR file must contain your EJB class files or the Java class files for the application client.
     – The META-INF directory of the archive must include your `persistence.xml` file.
     – If your application uses mapping files, orm.xml, or a custom mapping file, the JAR file must contain those files as well. If the location of the orm.xml file is not specified in the persistence unit, the default location is the META-INF directory of the JAR file.
   - For a standalone web module, package the application in a standard Web Application archive (WAR). Ensure that you package the application with these conditions:
     – The WAR file must contain your web application class files. The web application class files must be included in the WEB-INF/classes directory or in a JAR file that is located in the WEB-INF/lib directory of the WAR file.
     – Your `persistence.xml` file must be included in the WEB-INF/classes/META-INF directory or in the META-INF directory of a JAR file that is included in your WEB-INF/lib directory of your WAR file.
     – If your application uses mapping files, orm.xml, or a custom mapping file, the WAR file must also contain those files. Mapping files can reside in the WEB-INF/classes directory or in a JAR file that is contained within the WEB-INF/lib directory of the WAR file. Use the <mapping-file> element of the `persistence.xml` file to specify the location of mapping files. For example:

     ```
     <mapping-file>META-INF/JPAorm.xml</mapping-file>
     ```

   - For enterprise application that contains one or more modules, package the application in a standard Enterprise Application archive (EAR). An enterprise application can contain one ore more EJB module, web module, or application client module. Ensure that you package the application with these conditions:
     – If multiple modules use the same persistence unit, you can create a persistence archive and package the persistence archive within your EAR file.
     – Include your entity classes, any necessary supporting classes, your `persistence.xml` file, and additional mapping files in the persistence archive file. Follow the packaging rules for EJB and application client modules for the location of your `persistence.xml` file and mapping files.

– Each module that uses the persistence archive must have a class path entry in its META-INF/MANIFEST.MF file. Here is an example manifest file:

```
Manifest-Version: 1.0
Class-Path: MyJPAEntities.jar
```

– If your modules use separate persistence units and share entity classes, you can package the entity classes in a persistence archive and specify different `persistence.xml` file and mapping files for each module. If the modules do not share entity classes or a persistence configuration, package each module as a standalone EJB module, a standalone application client module, or a standalone web archive and then package them in the EAR file.

Example

This is a sample `persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="TheWildZooPU" transaction-type="JTA">
        <jta-data-source>jdbc/FooBarDataSourceJNDI</jta-data-source>
        <!-- additional Mapping file, in addition to orm.xml>
        <mapping-file>META-INF/JPAorm.xml</mapping-file>

        <class>com.company.bean.jpa.PersistebleObjectImpl</class>
        <class>com.company.bean.jpa.Animal</class>
        <class>com.company.bean.jpa.Dog</class>
        <class>com.company.bean.jpa.Cat</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
           <property name="openjpa.ConnectionFactoryProperties" value="PrettyPrint=true, PrettyPrintLineLength=72"/>
           <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
        </properties>

     </persistence-unit>
</persistence>
```

What to do next

For more information on any of the commands, classes or other OpenJPA information discussed here, refer to the Apache OpenJPA User's Guide.

Related tasks

"Mapping persistent properties to XML columns" on page 242
If your database supports Extensible Markup Language (XML) column types, you can use mapping tools to manage XML objects. The Java Persistence API (JPA) specification does not contain support for mapping XML columns to Java objects. You have the choice of mapping XML columns to a Java string or a Java byte array field. These mapping techniques make using the XML objects as strings or byte arrays difficult. JPA for the application server allows you to simplify the management of XML objects by using a third-party solution for mapping management.

Associating persistence units and data sources
Java Persistence API (JPA) applications allow you to specify the underlying data source used by the persistence provider to access the database.

Configuring a JDBC provider and data source
For access to relational databases, applications use the JDBC drivers and data sources that you configure for the application server.

"Developing applications that use JNDI" on page 1414
References to enterprise bean (EJB) homes and other artifacts such as data sources are bound to the WebSphere Application Server name space. These objects can be obtained through Java Naming and Directory Interface (JNDI). Before you can perform any JNDI operations, you need to get an initial context. You can use the initial context to look up objects bound to the name space.

Task overview: Data Studio pureQuery
Data Studio pureQuery provides Java Persistence API (JPA) users an alternative way to access a DB2 database. PureQuery supports static Structured Query Language (SQL).

Configuring to use pureQuery in a Java EE environment
Use this task to configure the application data source Java Database Connectivity (JDBC) provider to use pureQuery to access DB2.

Configuring pureQuery to use multiple package collections
Set up a pureQuery Java Persistence API (JPA) application to use multiple DB2 package collections.

Related reference

wsjpaversion command
Use this command line tool to find out information about the installed version of Java Persistence API (JPA) for WebSphere Application Server.

wsappid command

The Java Persistence API (JPA) specification allows an entity's primary key to be made up of more than one column. In this case, the primary key is referred to as a ″composite″ or ″compound″ primary key. You need to provide an ID class, which is specified by the @IdClass annotation, in order to manage a composite primary key. Use the identity tool for JPA to generate an ID class for entities that use composite primary keys.

Syntax

```
Before running the command, you must have a copy of persistence.xml on the classpath,
or specify it as a properties file through the -p [path_to_persistence.xml]
argument. Issue the command from the bin subdirectory of the app_install_root directory.
```

The command syntax is as follows:

```
wsappid.sh [parameters][arguments]
```

Parameters

The wsappid tool accepts the standard set of command-line arguments that are defined by the configuration framework along with the following:

- **-directory/-d** *<output_directory>*: The path to the output directory. If the directory does not match the generated output ID class package, the package structure will be created beneath the directory. If this

parameter is not specified, the wsappid tool will attempt to find the directory of the .java file for the class that is capable of persistence, and the wsappid tool uses the current directory if a .java file is not found.
- **-ignoreErrors/-i** <true/t | false/f>: If this parameter is set to false, an exception is thrown if the tool is run on any class that does not use the application identity or is not the base class in the inheritance hierarchy.
- **-token/-t** *<token>*: The token that is used to separate the values of stringed primary keys in the string form of the object ID. This option can be used only if there are multiple primary key fields. The default is ″**::**″.
- **-name/-n** *<id_class_name>*: The name of the identity class to generate. If this option is specified, the wsappid tool must be run on exactly one class. If the class meta-data already names an ID class for the object, this option will be ignored. If the name is not fully qualified, the package of the persistence class is appended to form the fully qualified name.
- **suffix** *<id_class_suffix>*: A string with which to suffix each persistent class name to form the identity class name. This option is overridden by the **-name/-n** parameter or by any object ID class that is specified in the meta-data.

Each additional argument to the wsappid tool must be one of the following:
- The full name of a persistent class.
- The `.java` name for a persistent class.
- The `.class` file of a persistent class.

Usage

The identity tool used with JPA for application server simplifies the task of creating an identity class for entities that use composite IDs. A composite ID refers to an identity that uses more than one field as its primary key. The entity class must be compiled, and primary keys need to be identified in the entity class. Run the wsappid tool from the command line in the *app_install_root*/bin/ directory. When you run this command, a new class representing the composite ID of the entity is generated. Messages and errors are logged to the console as specified.

Examples

Consider the following entity :

```
@Entity
public class Employee {

@Id
private int division;

@Id private int id;
// . . .
}
```

Before the entity can be used we need an ID class. For this example, assume that the entity is found in the src/main/java directory.

To generate an ID class for the Magazine entity run:

```
wsappid.sh -s Id src/main/java/Employee.java -d src/main/java
```

A new class, EmployeeId.java, will be generated in the src/main/java directory.

Additional information

You can refer to the Application identity tool in persistence classes in the Apache OpenJPA User's Guide.

wsenhancer command

The entity enhancer tool for Java Persistence API (JPA) applications in the application server inserts bytecode into an entity class file that allows the JPA provider to manage the state of an entity.

JPA with the application server requires that all entity classes be enhanced if you want to manage their state. In a container-managed environment, automated enhancement is provided by the containers. In a Java SE environment, though, there are no containers to manage persistence and you might use this command frequently before packaging application files for testing. After you have created the JPA entities, you can run the wsenhancer tool to inject bytecode into the entities before packaging the JAR file into the EAR file for the application.

Syntax

Before running the command, you must have a copy of persistence.xml on the classpath, or specify it as a properties file through the -p [*path_to_persistence.xml*] argument. Issue the command from the bin subdirectory of the *app_install_root* directory.

The command syntax is as follows:

```
wsenhancer.sh [parameters][arguments]
```

Parameters

The enhancer accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-directory/-d** *<output directory>*: specifies the path to the output directory. If the directory does not match the enhanced class's package, the package structure will be created beneath the directory. By default, the enhancer overwrites the original .class file.
- **-enforcePropertyRestrictions/-epr** <true/t | false/f>: specifies whether to generate an exception when it appears that a property access entity is not obeying the restrictions that are placed on property access. The default is set to false.
- **-addDefaultConstructor/-adc** <true/t | false/f>: specifies that all of the persistent classes define a no-argument constructor. This flag informs the enhancer if it should add a protected no-arg constructor to any persistent classes in which the constructor is not already present.
- **-tmpClassLoader/-tcl** <true/t | false/f>: specifies whether the enhancer should load persistent classes with a temporary class loader. This function allows other code to load the enhanced version of the class afterwards within the same Java Virtual Machine (JVM). The default is set to true.

  Note: If you are encountering class loading problems when running the enhancer, you can set this flag to false as a debugging step.
- For class name, specify one of the following:
  - The full name of a class.
  - The `.java` name for a class.
  - The `.class` file of a class.

If you do not supply any arguments to the enhancer, it will run on the classes in your persistent class list.

Usage

In order to use the wsenhancer tool you need entities defined to JPA specifications, and the entities need to be compiled. Run the wsenhancer tool against the entities before packaging them into a JAR file. If the entities are already packaged, you must extract the entity class files, run the enhancer, and recreate the JAR file.

To enhance your entities:

- Verify that your entities are in the class path, if they are not, add them.
- Run the wsenhancer command. It is found in ${app_server_root}/bin directory.

Messages and errors are logged to the console as specified in the log settings. After invoking the wsenhancer command, your files are enhanced.

Examples

To enhance all entities on the classpath:

```
$ cd build
/home/user/myproject/build $ ${app_server_root}/bin/wsenhancer.sh
```

All entities in myproject will be enhanced.

To enhance a specific entity when you have the source files:

```
$ cd build
/home/user/myproject/build $ ${app_server_root}/bin/wsenhancer.sh Magazine.java
```

To enhance a specific entity when you have the compiled class files:

```
$ export CLASSPATH=target/classes
$ ${app_server_root}/bin/wsenhancer.sh /bin/wsenhancer.sh target/classes/jpa/example/MyEntity.class
```

The entity, Magazine.java, located in myproject will be enhanced.

Additional information

For more information about enhancement tools, refer to the section on persistent classes in the Apache OpenJPA reference documentation.

wsmapping command

The wsmapping tool is used to provide top-down mapping of the entity object model to the database relational model. You can use the wsmapping tool to create database tables.

You can use the wsmapping tool to create database tables.

Syntax

Before running the command, you must have a copy of persistence.xml on the classpath, or specify it as a properties file through the -p [*path_to_persistence.xml*] argument. Issue the command from the bin subdirectory of the *app_install_root* directory.

The command syntax is as follows:

```
wsmapping.sh [options][arguments]
```

Parameters

The mapping tool accepts the standard set of command-line arguments defined by the configuration framework with the following options:

- **-schemaAction/-sa** <add | refresh | drop | build | reflect | retain | createDB | import | export | none>: The action to execute against the schema. These options correspond to the actions of the schema tool. **Add** is the default action if none is specified. Actions can be composed in a list separated by commas.

  Note: The wsmapping tool accepts the -action/-a flag to specify the action to take on individual classes. Unless you are running wsmapping on all of your persistent types at once, or dropping a mapping, you need to use the default **add** action or the **build** action. Otherwise, you might inadvertently drop schema components that are used by classes that you are not currently running the tool against.
- **-schemaFile/-sf** <true/t | false/f>: This option can be used to write the planned schema to an XML document rather than modify the database. The XML document can then be modified, manupulated and committed to the database with the schema tool.

- **-sqlFile/-sql** <stdout | output file>: This option can be used to write the planned schema modifications to an SQL script rather than modify the database. Combine this parameter with a **schemaAction** of build to generate a script that recreates the schema for the current mappings, even if the schema already exists.
- **-dropTables/-dt** <true/t | false/f>: When this option is set to true, schema drops tables that appear to be unused during **retain** and **refresh** actions. The default is true.
- **-dropSequences/-dsq** <true/t | false/f>: If this option is set to true, schema drops sequences that are unused during **retain** and **refresh** actions. The default is true.
- **-openjpatables/-ot** <true/t | false/f>: When reflecting the schema, this parameter determines whether to reflect on tables and sequences with names that start with OPENJPA_. Certain OpenJPA components can use these tables and sequences, such as the table schema factory. When using other actions, the openjpaTables parameter controls whether these tables can be dropped or not. The default setting is false.
- **-ignoreErrors/-i** <true/t | false/f>: If set to false, an exception is displayed if the tool encounters any database errors. The default is set to false.
- **-schemas/-s** *<schema list>*: Denotes a list of schema and table names the OpenJPA should access when running the wsschema tool. This is the equivalent to setting the openjpa.jdbc.Schemas property to run once. This parameter corresponds to the **-schemas/-s** parameter in the wsschema tool. This option is ignored if **-readSchema/-rs** is not set to true.
- **-readSchema/-rs** <true/t | false/f>: Set this option to true to read the entire existing schema when the mapping tool runs. Reading the existing schema ensures that OpenJPA does not generate any mappings that use the table, index, primary key or foreign key names that conflict with existing names.

  Note: Depending on the particular JDBC driver, selecting the **-readSchema/-rs** function can slow down the process for large schemas.
- **-primaryKeys/-pk** <true/t | false/f>: This flag determines if the primary keys can be manipulated on existing tables. The default is true.
- **-foreignKeys/fk** <true/t | false/f>: This flag determines if foreign keys can be manipulated on existing tables. The default is true. This means that to add any new foreign key to a class that has already been mapped, you must explicitly set this parameter flag to true.
- **-indexes/-ix** <true/t | false/f>: This flag determines if indexes can be manipulated on existing tables. The default is true. This means that to add any new indexes to a class that has already been mapped, you must explicitly set this parameter flag to true.
- **-sequences/-sq** <true/t | false/f>: This flag determines if sequences can be manipulated. The default is true.
- **-meta/-m** <true/t | false/f>: This flag determines whether or not a mapping applies to metadata rather than, or in addition to, standard mappings.
- The wsmapping tool accepts the **-action/-a** flag to specify the action to take on each class. Multiple actions can be composed in a list, separated by commas. The available actions are:
  - **buildSchema**: This is the default action. The **buildSchema** action makes the database schema match your existing mappings. If the provided mappings conflict with the class definitions, OpenJPA fails with an informative exception.
  - **validate**: Ensure that the mappings for the given classes are valid and that they match the schema of the database. No mappings of tables are changed as a result of this action. An exception is occurs if any mappings are invalid.

Each additional argument to the wsmapping tool must be one of the following:
- The full name of a persistent class.
- The `.java` name for a persistent class.
- The `.class` file of a persistent class.

If you do not supply any arguments to the wsmapping tool, it runs on the classes in the persistent classes list.

Usage

Before running the wsmapping tool, you need to configure the datasource information, including the URL, user, and password. It is required that the wsenhancer tool is run before the wsmapping tool to insert bytecode into the entity classes. Also, the compiled class files for your entities should be on the classpath (assume entity class files can be found in target/classes) , for example:

```
export CLASSPATH=${CLASSPATH}:target/classes

wsmapping.sh ...
```

To create tables, run the wsmapping command from the ${WAS_HOME}/bin directory. When completed, the database tables are created or updated. Messages and errors are logged to the console as specified by log settings.

wsmapping.sh . . . On Windows :

Note: By specifying the buildSchema parameter to the openjpa.jdbc.SynchronizeMappings property, the mapping tool provides the default mapping that matches with the database schema automatically. You are not required to run this mapping tool if the default mapping satisfies the necessary database schema.

Examples

To create the database tables needed for the Magazine.java file:

```
${WAS_HOME}/bin/wsmapping.sh Magazine.java
```

To drop the tables for Magazine.java:

```
C:\> %WAS_HOME%/bin/wsmapping.sh -sa dropDB Magazine.java
```

To validate the mappings for all classes on the classpath:

```
C:\> %WAS_HOME%/bin/wsmapping.sh -a validate
```

Additional information

Consult chapter 7, Mapping in the OpenJPA reference documentation for more information and examples.

wsreversemapping command

The wsreversemapping tool generates persistent class definitions and metadata from a database schema.

Syntax

Before running the command, you must have a copy of persistence.xml on the classpath, or specify it as a properties file through the -p [*path_to_persistence.xml*] argument. Issue the command from the bin subdirectory of the *app_install_root* directory.

The command syntax is as follows:

```
wsreversemapping.sh [parameters][arguments]
```

Parameters

The wsreversemapping tool accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-schemas/-s** *<schema and table names>*: A list of schema and table names, separated by commas, to run the reversmapping tool on if no XML schema file is supplied. Each element of the list must follow the naming conventions for the openjpa.jdbc.Schemas property. If this parameter flag is omitted, it defaults to the value of the **Schemas** property. If the **Schemas** property is not defined, then all schemas will be reverse mapped.

- **-package/-p** *<package name>*: The package name of the generated classes. If no package name is given, the generated code will not contain package declarations.
- **-directory/-d** *<output directory>*: All generated code and metadata will be written to the directory at this path. If the path does not match the package of a class, the package structure will be created beneath this directory. This parameter defaults to the current directory.
- **-useSchemaName/-sn** <true/t | false/f>: Set this parameter flag to true to include the schema as well as the table name in the name of each generated class. This can be useful when dealing with multiple schemas that have tables with identical names.
- **-useForeignKeyName/-fkn** <true/t | false/f>: Set this parameter flag to true if you want the field names for relations to be based on the database foreign key name. By default, relation field names are derived from the name of the related class.
- **-nullableAsObject/-no** <true/t | false/f>: By default, all non-foreign key columns are mapped to primitives. Set this parameter flag to true to generate primitive wrapper fields instead for columns that allow null values.
- **-blobAsObject/-bo** <true/t | false/f>: By default, all binary columns are mapped to the byte[] fields. Set this parameter flag to true to map them to Object fields instead.

  Note: When mapped this way, the column is presumed to contain a serialized Java object.
- **-primaryKeyOnJoin/pkj** <true/t | false/f>: The standard reverse mapping tool behavior is to map all tables with primary keys to persistent classes. If your schema has primary keys on many join tables as well, set this flag to true to avoid creating classes for those tables.
- **-inverseRelations/-ir** <true/t | false/f>: Set this parameter flag to false to prevent the creation of inverse one-to-many/one-to-one relations for every many-to-one/one-to-one relation detected.
- **-useDatastoreIdentity/-ds** <true/t | false/f>: Set to true to use datastore identity for tables that have single numeric primary key columns. The tool typically uses application identity for all generated classes.
- **-useBuiltinIdentityClass/-bic** <true/t | false/f>: Set this parameter flag to false to prevent the reversemapping tool from using built-in application identity classes when possible. This will force the tool to create custom application identity classes even when there is only one primary key column.
- **-innerIdentityClasses/-inn** <true/t | false/f>: Set this parameter flag to true to have any generated application identity classed be created as static inner classes within the persistent classes. The default setting is false.
- **-identityClassSuffix/-is** *<suffix>*: Suffix to append to the class names to form application identity class names, or for inner identity classes, the inner class name. The default suffix is `Id`.
- **-typeMap/-typ** <type mapping>: A string that specifies the default Java classes to generate for each SQL type that is seen in the schema. The format is SQLTYPE1=JavaClass1, SQLTYPE2=JavaClass2. The SQL type name first looks for a customization that is based on SQLTYPE(SIZE,PRECISION), then SQLTYPE(SIZE), and then SQLTYPE. If a column with type CHAR is found, it will first look for the CHAR(50,0) type name specification, then it will look for the CHAR(50), and finally it will look for the CHAR. For example, to generate a char array for every char column whose size is exactly 50 characters, and to generate a short for every type name of INTEGER, you might specify: CHAR(50)=char[],INTEGER=short.

  Note: Various databases report different type names differently, one database type may not work for another database. Enable TRACE level logging on the MetaData channel to track which type names JPA for WebSphere Application Server is examining.
- **-customizerClass/-cc** *<class name>*: The full class name of an org.apache.openjpa.jdbc.meta.ReverseCustomizer customization plugin. If you do not specify a reverse customizer of your own, the system defaults to a PropertiesReverseCustomizer. This customizer allows you to specify simple customization options in the properties file given with the -customizerProperties flag.
  - **-customizerProperties/-cp***<properties file or resource>*: The path or resource name of a properties file to pass to the reverse customizer on initialization.
  - **-customizer/-c** *<property name> <property value>*: The given property name will be matched with the corresponding Java bean property in the specified reverse customizer, and set to the given value.

Usage

The wsreversemapping tool is used to perform reverse (bottom-up) mappings of database tables to entity source files. This is useful if developers want to generate Java files from a database for use in other JPA applications. To run this tool:

- You need to have database tables and your database connection configured.
- Run the wsreversemapping tool from the command line in the $ {WAS_HOME}/bin directory.
- The tool will generate .java files for every class along with a XML descriptor file orm.xml

The generated Java files from the wsreversemapping tool might require some editing before they can be used in an application. Also, generated files will not contain annotations. Annotations can be added manually if desired. Messages and errors are logged to the console as specified by the configuration.

Examples

Generate entities based on the information saved in the schema.xml file. Schema.xml was created by running the schema tool. The Java files are created in the **src** directory and use the package com.xyz:

```
${WAS_HOME}/bin/wsreversemapping.sh -pkg com.xyz -d ./src schema.xml
```

Generate entities based on information in a DB2 database. Entities are created in the **src** directory, and use the package com.reversemapped:

```
C:\> %WAS_HOME%/bin/wsreversemapping.bat -sa dropDB Magazine.javapkg com.reversemapped -d src
-connectionDriverName=com.ibm.db2.jcc.DB2Driver -connectionURL=jdbc:db2:localhost:50000/TEST
-connectionUser=db2User -connectionPassword=db2Password
```

Additional information

For more information, consult the mapping section in the Apache OpenJPA User's Guide.

wsschema command

The schema tool can be used to view the database schema in XML form or match an XML schema to an existing database.

Developers may find that they need the wsschema tool for its powerful functions. The wsschema tool can reflect on the current database schema, optionally translating it into an XML representation for further manipulation. Also, the schema tool can take an XML schema definition, calculate the differences between the XML and the existing database schema, and apply the necessary changes to make the databases correspond to the XML schema. The XML format used by the schema tool is abstract from the differences in SQL dialects used by different vendors. The tool also automatically adapts its SQL to meet foreign dependencies, thus the schema tool is useful as a general way to manipulate the schemas.

Syntax

The command syntax is as follows:

```
wsschema.sh [parameters][arguments]
```

Issue the command from the bin subdirectory of the *app_install_root* directory.

Parameters

The wsschema tool accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-ignoreErrors/-i** <true/t | false/f>: If set to false, an exception will be thrown if the tool encounters any database errors. The default is set to false.
- **-file/-f** *<stdout | output file>*: Use this option to write a SQL script for the planned schema modifications, rather than committing them to the database. When this is used in conjunction with the export or reflect

actions, the named file will be used to write the exported schema XML. If the file names a resource in the CLASSPATH, data will be written to that resource. Use stdout to write to standard output. The default setting is stdout.

- **-openjpatables/-ot** <true/t | false/f>: When reflecting the schema, this parameter determines whether to reflect on tables and sequences whose names start with OPENJPA_. Certain OpenJPA components may use such tables and sequences, such as the table schema factory. When using other actions, openjpaTables controls whether these tables can be dropped or not. The default setting is false.
- **-dropTables/-dt** <true/t | false/f>: When this option is set to true, schema drops tables that appear to be unused during retain and refresh actions. The default is true.
- **-dropSequences/-dsq** <true/t | false/f>: If this option is set to true, schema drops sequences that appear to be unused during retain and refresh actions. The default is true.
- **-sequences/-sq** <true/t | false/f>: This flag determines if sequences may be manipulated. The default is true.
- **-indexes/-ix** <true/t | false/f>: This flag determines if indexes may be manipulated on existing tables. The default is true.
- **-primaryKeys/-pk** <true/t | false/f>: This flag determines if primary keys may be manipulated on existing tables. The default is true.
- **-foreignKeys/fk** <true/t | false/f>: This flag determines if foreign keys may be manipulated on existing tables. The default is true.
- **-record/-r** <true/t | false/f>: This flag permits or prevents writing schema changes made by the schema tool to the current schema factory. Select true to permit writing schema changes or false to prevent writing schema changes. The default is set to true.
- **-schemas/-s** <*schema list*>: Denotes a list of schema and table names the OpenJPA should access when running the schema tool. This is the equivalent to setting the openjpa.jdbc.Schemas property to run once.

Note: The schema tool accepts the **-action/-a** flag. Multiple actions may be composed in a list, separated by commas. The available actions are:

- **add**: This is the default action if no other actions are specified. It updated the schema with the given XML documents by adding tables, columns, indexes, or other components. This action never drops any schema components.
- **retain**: This action keeps all schema components in the given XML definition but drops the rest from the database. This action never adds any schema components.
- **drop**: Drops all schema components in the schema XML. This action will drop tables only if they would have 0 columns after dropping all columns listed in the XML.
- **refresh**: This action is the equivalent of the **retain** and the **add** functions.
- **build**: Generates SQL to build a schema matching the one in the supplied XML file. Unlike the**add** action, this option does not take into account the fact that part of the schema defined in the XML file may already exist in the database. This action is typically used in conjunction with the **-file/-f** parameter flag to write a SQL script. This script may be used later to recreate the schema in the XML.
- **reflect**: Generates a XML representation of the current database schema.
- **createDB**: This action generates SQL to recreate the current database. This action is typically used in conjunction with the **-file/-f** parameter flag to write a SQL script that can be used to recreate the current schema on a new database.
- **dropDB**: Generates SQL to drop the current database. Like the **createDB** action this may be used with the **-file/-f** parameter flag to script a database drop rather than manually perform it.
- **import**: Imports the given XML schema definition into the current schema factory.

  Note: This action will do nothing if the schema factory does not store a record of the schema.
- **export**: Exports the current schema factory's stored schema definition to a XML file.

  Note: This may produce an empty file if the schema factory does not store a record of the schema.
- **deleteTableContents**: This action executes SQL to delete all rows from all tables that OpenJPA finds.

Usage

The wsschema tool is used to obtain a XML file that describes the schema of your database. To generate a XML schema file:

- You need to have database tables and your database connection configured.
- Run the wsschema tool from the command line in the $ {WAS_HOME}/bin directory.
- The tool will generate a XML file that describes the database schema.

Messages and errors are logged to the console as specified by the configuration.

Examples

Add the necessary schema components to the database to match the given XML document without dropping any data:

```
$ wsschema.sh targetSchema.xml
```

Repeat the same action as the previous example, this time not changing the database but instead writing any planned changes to a SQL script:

```
 wsschema.sh -f script.sql targetSchema.xml
```

Write an SQL script that will recreate the current database:

```
$ wsschema.sh -a createDB -f script.sql
```

Refresh the schema and delete all the contents of all the tables that OpenJPA knows about:

```
$ wsschema.bat -a refresh,deleteTableContents
```

Drop the current database:

```
$ wsschema.sh -a dropDB
```

Write a XML representation of the current schema to the file *schema.xml*:

```
$ wsschema.sh -a reflect -f schema.xml
```

Additional information

For more information, refer to chapter 4 JDBC, in the OpenJPA reference documentation.

wsdb2gen command

The command allows users to utilize the pureQuery feature in Java Persistence API (JPA) applications.

Syntax

The command syntax is as follows:

```
wsdb2gen.sh [parameters]
```

Before running the command, your `persistence.xml` file must be in the META-INF directory and the META-INF directory must be in the class path.

Parameters

- **-help** : This parameter displays the help information.
- **-pu** : The name of the persistence unit defined in `persistence.xml` file.
- **-collection** : The collection-id which is assigned to package names. The default is `NULLID`.
- **-url** : The url of the target database. This is used to validate the generated SQL. A url must be specified either in the `persistence.xml` file or as a command option. If both are specified, the url specified in the command option will be used.
- **-user** : The user id

- **-pw** : The corresponding password to connect to target database. If this parameter is not specified, the value found in the `persistence.xml` file will be used.
- **-package** : If this parameter is specified, then the -**package** parameter takes the string value package name and a single DB2 package with the specified name is generated. If the -**package** parameter is not specified, then one package is generated for each entity class. The name consists of seven or fewer letters. For each entity class, the first seven characters of the entity class is used for the package name. If the first seven characters are not unique, then the package name is changed to create a unique name.

Usage

The `persistence.xml` file must be included in the application Java archive (JAR) file and is also used as input in the DB2 bind to create the DB2 package. The wsdb2gen command requires a connection to a database in order to validate generated SQL. The database does not have to be the same as the run time database, but it should be at the same version and release level.

Ensure the following JAR files are on the class path:

- pdq.jar
- pdqmgmt.jar
- db2jcc.jar
- db2jcc_licence_cu.jar.

If the database URL specifies a DB2 for zOS database, then the following JAR file must also be on the class path: db2jcc_licence_cisuz.jar

Examples

```
wsdb2gen.sh  -pu payroll -collection prod1 -url jdbc:db2://myhostname:50000/proddb  -user produser -pw secret
```

ANT Task WsJpaDB2GenTask

The ANT task `WsJpaDB2GenTask` provides an alternative to the wsdb2gen command.

The `WsJpaDB2GenTask` ANT task utility allows users to utilize the pureQuery feature in Java Persistence API (JPA) applications. Instead of executing the wsdb2gen from the command line, you can use the example code in your ANT build XML file to use the `WsJpaDB2GenTask` in your build process.

```
Both the PDQ runtime Java archive (JAR) files, pdq.jar and pdqmgmt.jar, must be specified using the ANT -lib option.
```

Example

The example listed below could be run with the ANT command using the following:

```
<!-- invoke this build using the ANT command
ant jar -noclasspath -lib c:/was7/lib/j2ee.jar
-lib c:/was7/plugins/com.ibm.ws.jpa.jar
-lib c:/sqllib/java/db2jcc.jar
-lib c:/sqllib/java/db2jcc_license_cu.jar
-lib c:/sqllib/java/pdq.jar
-lib c:/sqllib/java/pdqmgmt.jar
-->
```

When calling the ANT command, the JAR files for pureQuery, JPA, and the JDBC driver must be on the library list.

```
<?xml version="1.0"?>


<project name="sample" default="jar">

<taskdef name="enhancer" classname="org.apache.openjpa.ant.PCEnhancerTask" />
```

```
<taskdef name="wsdb2gen" classname="com.ibm.websphere.persistence.pdq.ant.WsJpaDB2GenTask" />

<target name="clean" description="remove intermediate files">
<delete dir="classes"/>
<delete dir="enhanced" />
<delete>
<fileset dir="." includes="META-INF/*.pdqxml" />
<fileset dir="." includes="sample.jar" />
</delete>
</target>

<target name="compile"
description="compile the Java source code to class files">
<mkdir dir="classes"/>
<javac srcdir="." destdir="classes">
<classpath>
<pathelement location="c:/was7/lib/j2ee.jar"/>
<pathelement location="c:/was7/plugins/com.ibm.ws.jpa.jar" />
</classpath>
</javac>
</target>

<target name="enhance" depends="compile" >
<mkdir dir="enhanced" />
<enhancer directory="./enhanced" >
<config propertiesFile="META-INF/persistence.xml" />
<classpath>
<pathelement location="." />
<pathelement location="classes" />
</classpath>
</enhancer>
</target>


<target name="wsdb2gen" depends="enhance" >
<wsdb2gen pu="MyAntTest" url="jdbc:db2://localhost:50000/demodb" user="user1" pw="secret" >
<classpath>
<pathelement location="."/>
<pathelement location="enhanced" />
</classpath>
</wsdb2gen>
</target>


<target name="jar" depends="wsdb2gen"
description="create a Jar file for the application">
<jar destfile="sample.jar">
<fileset dir="classes" includes="**/*.class"/>
<fileset dir="." includes="META-INF/*.xml" />
</jar>
</target>
</project>
```

Developing and packaging JPA applications for a Java SE environment

Prepare and package persistence applications to test outside of the application server container in a Java SE environment.

About this task

JPA applications require different configuration techniques from applications that use container-managed persistence (CMP) or bean-managed persistence (BMP). They do not follow the typical deployment techniques that are associated with applications that implement CMP or BMP. In JPA applications, you must define a persistence unit and configure the appropriate properties in the `persistence.xml` file to ensure that the applications can run in a Java SE environment.

There are some considerations for running JPA applications in a Java SE environment:

- Resource injection is not available. You must configure these services specifically or programmatically.
- The life cycle of the EntityManagerFactory and EntityManager are managed by the application. Applications control the creation, manipulation, and deletion of these constructs programmatically.

For this task, you will need to specify the com.ibm.ws.jpa.thinclient_7.0.0.jar standalone Java archive (JAR) file in your class path. This standalone JAR file is available from the client and server install images. The location of this file on the client install image is ${*app_client_root*}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar.
The location of this file on the server install image is ${*app_server_root*}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar

1. Generate your entities classes. Depending upon your development model, you might use some or all of the JPA tools:
   - **Top-down mapping:** You start from scratch with the entity definitions and the object-relational mappings, and then you derive the database schemas from that data. If you use this approach, you are most likely concerned with creating the architecture of your object model and then writing your entity classes. These entity classes would eventually drive the creation of your database model. If you are using a top-down mapping of the object model to the relational model, develop the entity classes and then use OpenJPA functionality to generate the database tables that are based on the entity classes. The wsmapping tool would help with this approach.
   - **Bottom-up mapping:** You start with your data model, which are the database schemas, and then you work upwards to your entity classes. The wsreversemapping tool would help with this approach.
   - **Meet in the middle mapping:** probably the most common development model. You have a combination of the data model and the object model partially complete. Depending on the goals and requirements, you will need to negotiate the relationships to resolve any differences. Both the wsmapping tool and the wsreversemapping tool would help with this approach.

   The JPA solution for the application server provides several tools that help with developing JPA applications. Combining these tools with IBM Rational Application Developer provides a solid development environment for either Java EE or Java SE applications. Rational Application Developer includes GUI tools to insert annotations, a customized `persistence.xml` file editor, a database explorer, and other features. Another alternative is the Eclipse Dali project. More information on Rational Application Developer or the Eclipse Dali plugin can be found at their respective web sites.
2. Optional: Enhance the entity classes using the JPA enhancer tool, or specify the Java agent to perform dynamic enhancement at run time.
   - Use the wsenhancer tool. The enhancer post-processes the bytecode that is generated by the Java compiler and adds the fields and methods that are necessary to implement the persistence features. For example:

     ${*app_client_root*}/bin/wsenhancer.sh [parameters][arguments]
   - You can specify the Java agent mechanism to perform the dynamic enhancement at run time. For example, type the following at the command prompt:

     java -javaagent:${*app_client_root*}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar *com.xyz.Main*
3. Optional: If you are not using the development model for bottom-up mapping, generate or update your database tables automatically or by using the wsmapping tool.
   - By default, the object-relational mapping does not occur automatically, but you can configure the application server to provide that mapping with the openjpa.jdbc.SynchronizeMappings property. This property can accelerate development by automatically ensuring that the database tables match the object model. To enable automatic mapping, include the following line in the `persistence.xml` file:

     ```
     <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
     ```

     Note: To enable automatic object-relational mapping at run time, all of your persistent classes must be listed in the Java .class file, mapping file, and Java archive (JAR) file elements in XML format.

- To manually update or generate your database tables, run the JPA mapping tool for the application server from the command line to create the tables in the database. For example:

  ${*app_server_root*}/bin/wsmapping.sh [parameters][arguments]

4. Optional: If you are using DB2 and want to use static SQL, run the wsdb2gen command. In order to use the wsdb2gen tool, pureQuery runtime must be installed. The wsdb2gen tool creates *persistence_unit_name*.pdqxml file under the same META-INF directory where your persistence.xml file is located. If you have multiple persistence units, wsdb2gen command must be run for each persistence unit. To use the wsdb2gen tool, type the following at a command prompt:

   ${*app_server_root*}/bin/wsdb2gen.sh [parameters]

5. Optional: If you are using application-managed identity, generate an application-managed identity class with the wsappid tool. When you use an application-managed identity, one or more of the fields must be an identity field. Use an identity class if your entity has multiple identity fields and at least one of the fields is related to another entity. The application-managed identity tool generates Java code that uses the identity class for any persistent type that implements application-managed identity. For example, type the following at a prompt:

   ${*app_client_root*}/bin/wsappid.sh [parameters][arguments]

6. Configure the properties of the persistence unit in the persistence.xml file that will be used in the JPA application.
   a. Specify your data source. Use the openjpa.Connection property to obtain a connection to the database. When you run a JPA application in a Java SE environment, a JTA data source will be treated as a data source that is not JTA compliant.
   b. Select com.ibm.websphere.persistence.PersistenceProviderImpl as the persistence provider.

      Note: Include the persistence provider in the classpath if you run the JPA application as a standalone application.

   c. Specify your database configuration options. If you are using pureQuery, configure your data source to use pureQuery, ensure the pdq.jar and pdqmgmt.jar files are included on the JDBC provider classpath. For more information, see topic ″Configuring a data source to use pureQuery″. Indicate the database type and method of connection in the persistence.xml file.

      ```
      <property name="openjpa.ConnectionDriverName" value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="openjpa.ConnectionURL" value="jdbc:derby:target/database/jpa-test-database;create=true"/>
      ```

   d. Specify the transaction type to RESOURCE_LOCAL. For example, the following entry should be in the persistence.xml file:

      ```
      <persistence-unit name="persistence_unit" transaction-type="RESOURCE_LOCAL">
      ```

   e. Include the location of the object relationship mapping file, orm.xml, and any additional mapping files. For example, the following entry should be in the persistence.xml file:

      ```
      <mapping-file>META-INF/JPAorm.xml</mapping-file>
      ```

   f. Add any vendor specific properties to the persistence unit.
7. Package the application.

   Note: Package the persistence units in separate JAR files to make them more accessible and reusable. If you package the persistence units this way, they can be tested outside the container both with and without the occurrence of database persistence. The persistence units can be included in standalone applications or they can be packaged into EAR files as persistence archive files. If you package the persistence unit into a persistence archive file, all of the application components must be able to access the persistence archive. The application that uses the persistence units must declare a dependency on the persistence archive using the MANIFEST.MF Class-Path: declaration.

   Note: If you are using pureQuery, add the *persistence_unit_name*.pdqxml files created previous or the or *persistence_unit_name*.pdqxml files created in the above Step 4 to the JPA application JAR

file. The files are located in same META-INF directory where your `persistence.xml` file is located.

To package the application:

```
jar -cvf ${jar_Name} ${entity_Path}
```

where ${*jar_Name*} represents the name of the JAR file to create, and ${*entityPath*} represents the root location where the entities reside, which is where you compiled them.

8. When you run your standalone application, specify the com.ibm.ws.jpa.thinclient_7.0.0.jar standalone JAR file in your class path when executing your application. For example, use the following Java call to run the com.xyz.Main standalone application:

```
java -classpath ${app_client_root}/runtimes/com.ibm.ws.jpa.thinclient_7.0.0.jar other_jar_file.jar com.xyz.Main
```

Example

The following is a sample `persistence.xml` file for the Java SE Environment:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="TheWildZooPU" transaction-type="RESOURCE_LOCAL">

        <!-- additional Mapping file, in addition to orm.xml>
        <mapping-file>META-INF/JPAorm.xml</mapping-file>

        <class>com.company.bean.jpa.PersistebleObjectImpl</class>
        <class>com.company.bean.jpa.Animal</class>
        <class>com.company.bean.jpa.Dog</class>
        <class>com.company.bean.jpa.Cat</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
            <property name="openjpa.ConnectionDriverName"
                      value="org.apache.derby.jdbc.EmbeddedDriver" />
            <property name="openjpa.ConnectionURL"
                      value="jdbc:derby:target/database/jpa-test-database;create=true" />
            <property name="openjpa.Log"
                      value="DefaultLevel=INFO,SQL=TRACE,File=./dist/jpaEnhancerLog.log,Runtime=INFO,Tool=INFO" />
            <property name="openjpa.ConnectionFactoryProperties"
                      value="PrettyPrint=true, PrettyPrintLineLength=72" />
            <property name="openjpa.jdbc.SynchronizeMappings"
                      value="buildSchema(ForeignKeys=true)" />
            <property name="openjpa.ConnectionUserName"
                      value="user" />
            <property name="openjpa.ConnectionPassword"
                      value="password"/>
        </properties>

     </persistence-unit>

</persistence>
```

What to do next

For more information on any of the commands, classes or other OpenJPA information discussed here, refer to the Apache OpenJPA User's Guide.

Related tasks

“Task overview: Storing and retrieving persistent data with the Java Persistence API (JPA)” on page 218
The Java Persistence API (JPA) for the application server defines the management of persistence and object/relational mapping within Java Enterprise Edition (Java EE) and Java Standard Edition (Java SE) environments.

Associating persistence units and data sources
Java Persistence API (JPA) applications allow you to specify the underlying data source used by the persistence provider to access the database.

Task overview: Data Studio pureQuery
Data Studio pureQuery provides Java Persistence API (JPA) users an alternative way to access a DB2 database. PureQuery supports static Structured Query Language (SQL).

Configuring to use pureQuery in a Java EE environment
Use this task to configure the application data source Java Database Connectivity (JDBC) provider to use pureQuery to access DB2.

Configuring pureQuery to use multiple package collections
Set up a pureQuery Java Persistence API (JPA) application to use multiple DB2 package collections.

Related reference

“wsappid command” on page 225
The Java Persistence API (JPA) specification allows an entity’s primary key to be made up of more than one column. In this case, the primary key is referred to as a ″composite″ or ″compound″ primary key. You need to provide an ID class, which is specified by the @IdClass annotation, in order to manage a composite primary key. Use the identity tool for JPA to generate an ID class for entities that use composite primary keys.

“wsenhancer command” on page 226
The entity enhancer tool for Java Persistence API (JPA) applications in the application server inserts bytecode into an entity class file that allows the JPA provider to manage the state of an entity.

wsjpaversion command
Use this command line tool to find out information about the installed version of Java Persistence API (JPA) for WebSphere Application Server.

“wsmapping command” on page 228
The wsmapping tool is used to provide top-down mapping of the entity object model to the database relational model. You can use the wsmapping tool to create database tables.

“wsreversemapping command” on page 230
The wsreversemapping tool generates persistent class definitions and metadata from a database schema.

“wsschema command” on page 232
The schema tool can be used to view the database schema in XML form or match an XML schema to an existing database.

Enabling SQL statement batching

SQL statement batching can improve the performance of your application server. Java Persistence API (JPA) for WebSphere Application Server uses the Java Database Connectivity (JDBC) addBatch and executeBatch APIs to batch statements.

About this task

By default, statement batching is enabled for DB2 and Oracle databases. To enable SQL statement batching and to set the batch limit for JPA applications, you need to configure the persistence.xml file. The following steps review how to enable and disable statement batching, as well as set the batch limit:

1. Define the UpdateManager property in the persistence.xml file. For example:

```
<property name="openjpa.jdbc.UpdateManager"
value="com.ibm.ws.persistence.jdbc.kernel.OperationOrderUpdateManager(batchLimit=100)"/>
```

Note: The example shows that the SQL statement batch limit is set to 100.

Note: If you are using a DB2 or an Oracle database, by default the SQL statement batching is enabled and set to batchLimit=100. However, if you are using DB2 or Oracle, you are not required to specify this property in the `persistence.xml` file.

2. If you need to disable SQL statement batching, set the batchLimit value to 0 (zero) or remove the property. However, if you are using a DB2 or an Oracle database, you must specify the DBDictionary property, database, and set the defaultBatchLimit to 0 (zero). For example:

```
<property name="openjpa.jdbc.DBDictionary" value="db2(defaultBatchLimit=0)"/>
```

Results

You have now updated the `persistence.xml` file to enable or disable statement batching and set the batch limit.

Database generated version ID

Java Persistence API (JPA) for WebSphere Application Server has extended OpenJPA to work with database generated version IDs. These generated version fields (timestamp or token) can be used to efficiently detect changes to a given row.

Trigger based version ID generation is supported for all databases that WebSphere Application Server supports. Support is based on two Version Strategies in JPA for WebSphere Application Server.

- @VersionStrategy(″com.ibm.websphere.persistence.RowChangeTimestampStrategy″), if the entity version field type is Timestamp, and
- @VersionStrategy(″com.ibm.websphere.persistence.RowChangeVersionStrategy″), if the entity version field type is Long

Database generated version ID example

In this example, the Entity class is defined with the new Version Strategy annotation. The Entity has a surrogate version column.

```
@Entity(name="Item")
@VersionColumn(name="versionField")
@VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy")
public class Item implements Serializable
{
 @Id
 private int id2;

 private String name;

 private double price;

   @OneToOne
   private Owner master;
}
```

The create table statement for this would be:

```
CREATE TABLE ITEM
(ID2  INT  NOT NULL,
 NAME  VARCHAR(50) ,
 PRICE   DOUBLE,
 OWNER_ID INT,
             VERSIONFIELD GENERATED ALWAYS FOR EACH ROW ON
              UPDATE AS ROW CHANGE TIMESTAMP
         PRIMARYKEY(ID2));
```

During any updates to Item (insert or update) the VersionColumn value will be updated in the database. After the update, the value for VersionColumn is retrieved from the database and updated in the in memory object. Thereby the objects in the data cache reflect the correct version value. Here the Entity is using the @VersionColumn which produces a Surrogate Version Id rather than defining an explicit field in the entity .

The Entity could also use @Version annotation to define an explicit version field. The explicit version field could be of type Long or Timestamp corresponding to the @VersionStrategy. During any updates to Item (insert or update) the Version value will be updated in the database. After the update the value for Version would be retrieved from the database and updated in the in memory object. Thereby the objects in the data cache would reflect the right version value.

This is an example where the Entity has a version field defined, and the type Timestamp matches the RowChangeTimestampStrategy in the @VersionStrategy (if the version field type is using type long, then the RowChangeVersionStrategy should be annotated to match) :

```
@Entity(name="Item")
@VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy")
public class Item implements Serializable

{
 @Id
 private int id2;

 private String name;

 private double price;

   @Version
   private Timestamp versionField;

   @OneToOne
   private Owner master;
}
```

For z/OS DB2 V9 and Linux, Unix, and Windows DB2 V9.5, the generated database column must be of type timestamp but we support both the RowChangeTimestampStrategy and the RowChangeVersionStrategy. MS SqlServer only supports a non timestamp generated version ID that goes with the RowChangeVersionStrategy . To use the RowChangeTimestampStrategy, you must use a trigger on a timestamp field in the database. For other databases you can use triggers to simulate database version generation and use either strategy.

Mapping persistent properties to XML columns

If your database supports Extensible Markup Language (XML) column types, you can use mapping tools to manage XML objects. The Java Persistence API (JPA) specification does not contain support for mapping XML columns to Java objects. You have the choice of mapping XML columns to a Java string or a Java byte array field. These mapping techniques make using the XML objects as strings or byte arrays difficult. JPA for the application server allows you to simplify the management of XML objects by using a third-party solution for mapping management.

About this task

DB2, Oracle, and SQLServer databases support XML column types, XPath queries, and indices over these columns.

Persistent properties to XML mapping

An embedded class with XML column support needs to use XML marshalling to write the data to the XML column and unmarshalling to retrieve the data from the XML column. The path expressions and predicates over the embedded class are converted to XML predicates, XPATH expressions, or XQuery expressions and are written to the database.

WebSphere Application Server allows JPA applications to use a third-party tool for XML mapping. This is done through the extension points for custom field mappings. The third-party mapping tool uses the extension points by providing a custom value handler for the persistent fields that are mapped to the XML columns. In OpenJPA, this value handler is named org.apache.openjpa.xmlmapping.XmlValueHandler and this handler requires the @Strategy annotation on the Java field that is mapped to the XML column.

1. Annotate the entity property using the XML value handler strategy. The mapping of persistent properties to XML columns requires the @Strategy and the @Persistent annotation.

   ```
   @Persistent
   @Strategy("org.apache.openjpa.xmlmapping.XmlValueHandler")
   ```

   The XML value handler for the persistent property is set to org.apache.openjpa.xmlmapping.XmlValueHandler.
2. Change the default for fetch type if it is necessary. For example:

   ```
   @Persistence(fetch=FetchType.LAZY)
   ```

   The fetch type is now LAZY. If a value for the fetch type is not entered, the default is set to EAGER.
3. Annotate your embedded classes with the binding annotations for Java API for XML Binding (JAXB). These bindings can be created from an XML schema by using the Java Architecture for XML Binding Compiler (XJC).
4. Make sure that the class that maps to the root of the XML document is annotated with @XmlRootElement, in addition to the other annotations.
5. Compile your Java sources.
6. Run the enhancer tool on the entities. Refer to the topic on the entity enhancer tool for more information.

Example

For example, shipAddress, a property of Order Entity, is mapped to XML column shipaddr:

```
@Entity
public class Order {
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
  int oid;
 @Persistent
 @Strategy("org.apache.openjpa.xmlmapping.XmlValueHandler")
      @Column(name="shipaddr")
      Address shipAddress;
...
```

The OpenJPA mapping tool generates a SHIPADDR column with XML type in the table definition for ORDER table.

Related tasks

“Developing and packaging JPA applications for a Java EE environment” on page 221
Containers in the application server can provide many of the necessary functions for the Java Persistence API (JPA) in a Java Enterprise Edition (Java EE) environment. The application server also provides JPA tools to assist you with developing applications in a Java EE environment.

“Developing and packaging JPA applications for a Java SE environment” on page 236
Prepare and package persistence applications to test outside of the application server container in a Java SE environment.

Related reference

“wsenhancer command” on page 226
The entity enhancer tool for Java Persistence API (JPA) applications in the application server inserts bytecode into an entity class file that allows the JPA provider to manage the state of an entity.

JPA Access Intent

Java Persistence API (JPA) Access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the JDBC isolation level and whether read, update or exclusive locks are acquired when retrieving data.

About this task

Note: For a JPA persistence provider on the application server, the application can specify isolation and ReadLockMode based on a TaskName. The TaskName provides a better control over applying these characteristics. The application defines a set of entity types and their corresponding access intent for each TaskName defined in a persistence unit.

Note:

- Access intent is available for application in the Java EE server environment
- Access intent is applicable to non-query entity manager interface methods. Query should use query hint interface to set its isolation and read lock values.
- Access intent is only available for DB2 databases.
- Access intent is in effect only when pessimistic lock manager is used. Add the following to the persistence unit’s property list. <property name=″openjpa.LockManager″ value=″pessimistic″/>

The following table compares the Enterprise JavaBeans (EJB) 2.x entity bean access intent with the JPA access intent properties:

Table 27. Access intent Properties and Descriptions

WebSphere EJB 2.x entity bean access intent	JPA access intent	Description
optimistic	isolation: Read Committed	Data is read but no lock is held. Version id is used on update to insure data integrity. Other transactions can read and update data.
	lockManager: Optimistic	
	query Hint: ReadLockMode: READ	
pessimistic read	isolation: Repeatable Read	Data is read with shared locks. Other transactions attempting to update data are blocked.
	lockManager: Optimistic	
	query Hint: ReadLockMode: READ	
pessimistic update	isolation: Repeatable Read	Data is retrieved with update or exclusive lock. Other writes are blocked until commit. This access intent can be used to serialize update access to data when there are multiple writers.
	lockManager: Pessimistic	
	query Hint: ReadLockMode: WRITE	

Table 27. Access intent Properties and Descriptions (continued)

WebSphere EJB 2.x entity bean access intent	JPA access intent	Description
pessimistic exclusive	isolation: Serializable	Data is retrieved with update or exclusive lock. Other writes are blocked until commit. This access intent can be used to serialize update access to data when there are multiple writers.
	lockManager: Pessimistic	
	query Hint: ReadLockMode:WRITE	

A TaskName is set on a transaction context by one of the following:

- TaskName is automatically set via the EJB container when a transaction begins using WebSphere local transaction (EJB unspecifid transaction), JTA global transaction in a Container-Managed Transaction (CMT) or user-initiated global transaction in a Bean-Managed Transaction (BMT)
- TaskName is manually set in an application using the TaskNameAccessor API provided for JPA.

The advantage of using task names is that the access intent can be specified on a request scope rather than specifying it in the `persistence.xml` file, which has an application scope across all entities. Often a query is contained in a method or component which is used in many different transaction contexts. Some of these contexts may require repeatable-read and update lock intent but other contexts do not.

Isolation level and read locks can be specified on:

- An application scope in the `persistence.xml` file. These isolation level and read lock type are properties specified in the persistence.xml file. They apply to all entities define in the persistence unit.
- A transaction scope via task name. Transaction scoped hints will override application scope values.
- Query instance with a query hint. Query hint can be used to override isolation and ReadLockMode for a particular query instance. A query hint will override isolation level and read locks specified at the application or transaction scope.

1. “Setting a TaskName using TaskNameAccessor” on page 245 This task will show how to use the TaskNameAccessor API to set JPA TaskName at runtime.
2. “Specify TaskName in a JPA persistence unit” on page 247 This task will show how to specify a TaskName in JPA persistence unit

What to do next

For further information on the background and purpose of Access intent, see the topic on Access intent service.

Related concepts

“Access intent service” on page 201
Access intent is a WebSphere Application Server runtime service that enables you to precisely manage an application’s persistence.

Related tasks

“Task overview: Storing and retrieving persistent data with the Java Persistence API (JPA)” on page 218
The Java Persistence API (JPA) for the application server defines the management of persistence and object/relational mapping within Java Enterprise Edition (Java EE) and Java Standard Edition (Java SE) environments.

Setting a TaskName using TaskNameAccessor

Using the TaskNameAccessor API to set Java Persistence API (JPA) TaskName at runtime.

About this task

In the Enterprise JavaBeans (EJB) container, a task name is automatically set by default upon a transaction begins. This action is performed when a component or business method is invoked in a CMT session bean or when an application invoke the `sessionContext.getTransaction().begin()` in a BMT session bean. This TaskName consists of a concatenation of the fully package qualified session bean type, a dot character and the method name. For example: `com.acme.MyCmtSessionBean.methodABC`.

If using JPA in the context of the Web container, an application must use the TaskNameAccessor API to set the TaskName in the current thread of execution.

Note: Once a TaskName is set on a transaction context, application must not set the TaskName again in the same transaction. This will avoid problems with on the JDBC connection for different database access.

This example contains the TaskNameAccessor API definition

```
package com.ibm.websphere.persistence;

public abstract class TaskNameAccessor {

 /**
  * Returns the current task name attached in the current thread context.
  * @return current task name or null if none is found.
  */
 public static String getTaskName ();

 /**
  * Add a user-defined JPA access intent task name to the current thread
  * context.
  *
  * @param taskName
  * @return false if an existing task has already attached in the current
  *         thread or Transaction Synchronization Registry (TSR) is not
  *         available (i.e. in JSE environment).
  */
 public static boolean setTaskName(String taskName);

}
```

This code example shows how to set a TaskName using TaskNameAccessor.

```
package my.company;

@Remote
class Ejb1 {
    // assumer no tx from the caller
    @TransactionAttribute(Requires)
    public void caller_Method1() {

        // an implicit new transaction begins
        // TaskName "my.company.Ejb1.caller_Method1" set on TSR

        ejb1.callee_Method?();
    }

    @TransactionAttribute(RequiredNew)
    public void callee_Method2() {

        // an implicit new transaction begins i.e. TxRequiredNew.
        // TaskName "my.company.Ejb1.callee_Method2" set on TSR
    }

    @TransactionAttribute(Requires)
    public void callee_Method3() {
```

```
        // In caller's transaction, hence TaskName remains
        //     "my.company.Ejb1.caller_Method1"
    }

    @TransactionAttribute(NotSupported)
    public void callee_LocalTx () {

        // Unspecified transaction, a new local transaction implicitly started.
        // TaskName "my.company.Ejb1.callee_LocalTx" set on TSR
    }

}
```

Note: In the above example, an application must be aware of transaction boundary will be subtly changed if Ejb1 uses local interface (@Local). For example, when `caller_Method1()` calls `callee_Method3` or `callee_LocalTx`, this will be treated as a Java method call. No EJB transaction semantics are honored.

What to do next

Once you have completed this step, continue on with “Specify TaskName in a JPA persistence unit” on page 247.

Related tasks

“JPA Access Intent” on page 244
Java Persistence API (JPA) Access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the JDBC isolation level and whether read, update or exclusive locks are acquired when retrieving data.

“Specify TaskName in a JPA persistence unit” on page 247
Specifying a TaskName in Java Persistence API (JPA) persistence unit

Specify TaskName in a JPA persistence unit

Specifying a TaskName in Java Persistence API (JPA) persistence unit

About this task

A TaskName is defined in the `persistence.xml` file using the `wsjpa.AccessIntent` property name in a persistence unit. The property value is a list of TaskNames, entity types and access intent definitions. The following example shows the contents of the `wsjpa.AccessIntent` property name in a persistence unit.

```
<property name  = "wsjpa.AccessIntent"
         value = "Tasks=' <taskName> { <entityName> ( <isolationLockValue> ) } ' "/>
                         A            A              A                    | | |
                         |            |              +--------- , --------+ | |
                         |            |                                     | |
                         |            +----------------- , -----------------+ |
                         +---------------------- , --------------------------+
  Tasks                        ::= <task> [ ',' <task> ]*

  <task>                       ::= <taskName> '{' <entity> [ ',' <entity> ]* '}'

  <entity>                     ::= <entityName> '(' <isolationLockValues> ')'

  <taskName>                   ::= <fully_qualified_identifier>

  <entityName>                 ::= <fully_qualified_identifier>

  <fully_qualified_identifier> ::= <identifier> [ '.' <identifier> ]*

  <identifier>                 ::= <idStartCharacter> [ <idCharacter> ]*

  <idStartCharacter>           ::= Character.isJavaIdentifierStart | '?' | '*'
```

```
<idStartCharacter>          ::= Character.isJavaIdentifierPart  | '?' | '*'

<isolationLockValues>       ::= <isolationLockValue> [ ',' <isolationLockValue> ]

<isolationLockValue>        ::= <isolation> | <readLock>

<isolation>                 ::= "isolation" '=' <isolationValue>

<readLock>                  ::= "readlock" '=' <readlockValue>

<isolationValue>            ::= "read-uncommitted"|"read-committed"|"repeatable-read"|"serializable"

<readlockValue>             ::= "read" | "write"
```

Before setting the TaskName in a persistence unit, keep the following in mind:

- White spaces are ignored between tokens.
- Only <isolation> and <readLock> contents are not case sensitive.
- <TaskName> is in the form of a fully package qualified method name, such as com.acme.bean.MyBean.increment, or an arbitrary user-defined task name, such as MyProfile.
- <entityName> is in the form of a fully package qualified class name such as com.acme.bean.Entity1.
- The wild card characters '?' or '*' can be used in <TaskName> and <entityName>. ″?″ matches any single character and ″*″ matches zero or more sequence characters.
- Only hintNames isolation and readLock are allowed on a task definition and the order is not significant
- If readLock has the value `write`, then isolation must be `repeatable-read` or `serializable`
- If readLock has the value `read`, it has no effect if the isolation is `read-uncommitted`.

The following code example shows how to specify a TaskName in JPA persistence unit.

```
package my.company;

@Remote
class Ejb1 {
    // assumer no tx from the caller
    @TransactionAttribute(Requires)
    public void caller_Method1() {

        // an implicit new transaction begins
        // TaskName "my.company.Ejb1.caller_Method1" set on TSR

        ejb1.callee_Method?();
    }

    @TransactionAttribute(RequiredNew)
    public void callee_Method2() {

        // an implicit new transaction begins i.e. TxRequiredNew.
        // TaskName "my.company.Ejb1.callee_Method2" set on TSR
    }

    @TransactionAttribute(Requires)
    public void callee_Method3() {

        // In caller's transaction, hence TaskName remains
        //     "my.company.Ejb1.caller_Method1"
    }

    @TransactionAttribute(NotSupported)
    public void callee_LocalTx () {

        // Unspecified transaction, a new local transaction implicitly started.
```

```
        // TaskName "my.company.Ejb1.callee_LocalTx" set on TSR
    }

}
```

Since a wild card can be used to specify TaskName and entity type, multiple specification matches may occur at runtime. The order defined in the `wsjpa.AccessIntent` property will be used to search for task names and entity types.

```
<properties>
  <property name="wsjpa.AccessIntent" value="Tasks="
        *.Task1 { *.Employee1 ( isolation=read-uncommitted                ),
                  *.Employee? ( isolation=repeatable-read,  readlock=write ),
                },
        *       { *.Employee3 ( isolation=serializable,     readlock=write ) },
      '" />
</properties>
```

Related tasks

“JPA Access Intent” on page 244
Java Persistence API (JPA) Access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the JDBC isolation level and whether read, update or exclusive locks are acquired when retrieving data.

“Setting a TaskName using TaskNameAccessor” on page 245
Using the TaskNameAccessor API to set Java Persistence API (JPA) TaskName at runtime.

Chapter 9. Messaging resources

Choosing a messaging provider

For messaging between application servers, perhaps with some interaction with a WebSphere MQ system, you can use the default messaging provider. To integrate WebSphere Application Server messaging into a predominantly WebSphere MQ network, you can use the WebSphere MQ messaging provider. You can also use a third-party messaging provider. To choose the provider that is best suited to your needs, consider what the application needs to do, and the business need for the provider to integrate well with your enterprise infrastructure.

About this task

Enterprise applications in WebSphere Application Server can use asynchronous messaging through services based on Java Message Service (JMS) messaging providers and their related messaging systems. These messaging providers conform to the JMS Version 1.1 specification.

You can configure any of the following messaging providers:

- The default messaging provider (which uses service integration as the provider)
- The WebSphere MQ messaging provider (which uses your WebSphere MQ system as the provider)
- A third-party messaging provider (which uses another company's product as the provider)

The types of messaging providers that can be configured in WebSphere Application Server are not mutually exclusive:

- All types of provider can be configured within one cell.
- Different applications can use the same, or different, providers.
- One application can access multiple providers.

No one of these providers is necessarily better than another. The choice of provider depends on what your JMS application needs to do, and on other factors relating to your business environment and planned changes to that environment.

Note: For backwards compatibility with earlier releases, WebSphere Application Server also includes support for the "V5 default messaging provider". For more information, see "Maintaining Version 5 default messaging resources" on page 1182.

1. Determine the environment and application requirements.

   If you need to use a third-party messaging provider, or interoperate with WebSphere Application Server Version 5 resources, use the associated provider. For more information, see "Other ways of managing messaging" on page 1172.

   If your existing or planned messaging environment involves both WebSphere MQ and WebSphere Application Server systems, and it is not clear to you whether you should use the default messaging provider, the WebSphere MQ provider, or a mixture of the two, complete the task "Choosing messaging providers for a mixed environment" on page 1151.

2. Choose the messaging provider:
   - Choose the default messaging provider.

     If you mainly want to use messaging between applications in WebSphere Application Server, perhaps with some interaction with a WebSphere MQ system, the default messaging provider is the natural choice because this provider is fully integrated with the WebSphere Application Server runtime environment. For more information, see "Default messaging provider" on page 1158. To configure and manage messaging with the default messaging provider, see Managing messaging with the default messaging provider.
   - Choose the WebSphere MQ messaging provider.

If your business also uses WebSphere MQ, and you want to integrate WebSphere Application Server messaging applications into a predominantly WebSphere MQ network, the WebSphere MQ messaging provider allows you to define resources for connecting directly to the queues in a WebSphere MQ system. For more information, see “WebSphere MQ messaging provider” on page 1159. To configure and manage messaging with the WebSphere MQ messaging provider, see Managing messaging with the WebSphere MQ messaging provider.

- Choose a third-party messaging provider.

  You can use any third-party messaging provider that supports the JMS Version 1.1 unified connection factory. You might want to do this, for example, because of existing investments.

  Note:

  – To administer a third-party messaging provider, use the resource adaptor or client supplied by the third party. You can still use the WebSphere Application Server administrative console to administer the JMS connection factories and destinations that are within WebSphere Application Server, but you cannot use the administrative console to administer the JMS provider itself, or any of its resources that are outside of WebSphere Application Server.

  – To use message-driven beans (MDBs), third-party messaging providers must include Application Server Facility (ASF), an optional feature that is part of the JMS Version 1.1 specification, or use an inbound resource adapter that conforms to the Java EE Connector Architecture (JCA) Version 1.5 specification.

  To work with a third-party provider, see “Managing messaging with a third-party messaging provider” on page 1173.

JMS providers collection

In the administrative console page, to view this page click **Resources** → **JMS** → **JMS providers**

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS providers that are available to WebSphere applications. For each JMS provider in the list, the entry indicates the *scope* level at which JMS resource definitions are visible to applications. You can create the same type of JMS provider at different Scope settings, to offer JMS resources at different levels of visibility to applications.

If you want to manage existing JMS resource definitions, or create a new JMS resource definition, you can select the name of one of the JMS providers in the list.

If you want to define your own JMS provider, other than the default messaging provider or WebSphere MQ, select the Scope setting at which JMS resource definitions are to be visible for that provider, then click **New**.

General properties

Name

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Select JMS resource provider

The variable, *{0}*, indicates the type of JMS resource that you are creating.

You select the scope setting on an earlier page. The choice of JMS providers depends on the scope that you selected. You might see a choice like the following list:

- Default messaging provider.

  Select this option if you want to the type of JMS resource to be provided by a service integration bus, as part of WebSphere Application Server.
- My JMSprovider

  Select this option if you want the type of JMS resource to be provided by your own JMS provider; not the default messaging provider or WebSphere MQ. You assign the name, in this example “My JMSprovider”, when you define the JMS provider to WebSphere Application Server. You must have installed and configured your own JMS provider before applications can use the JMS resources.
- WebSphere MQ messaging provider

  Select this option if you want the type of JMS resource to be provided by WebSphere MQ. You must have installed and configured WebSphere MQ before applications can use the JMS resources.
- V5 default messaging provider

  Select this option if you want the type of JMS resource to be provided by a WebSphere Application Server Version 5 node.

Activation specification collection

In the administrative console page, to view this page click **Resources** → **JMS** → **Activation specification**.

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS activation specifications that are available to WebSphere applications at the scope indicated by the **Scope** field.

Use a JMS activation specification if you want to use a message-driven bean as a J2EE Connector Architecture (JCA) 1.5 resource, to act as a listener on the default messaging provider.

General properties

Name

Provider

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Connection factory collection

In the administrative console page, to view this page click **Resources** → **JMS** → **Connection factory**.

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS connection factories that are available to WebSphere applications at the scope indicated by the **Scope** field.

A JMS connection factory is used to create connections to JMS destinations. When an application needs a JMS connection, an instance can be created by the factory of the JMS provider named in the Provider column of the list.

This type of connection factory is for applications that use the JMS 1.1 domain-independent interfaces (referred to as the “common interfaces” in the JMS specification).

This type of JMS connection factory can also be used by the domain-specific (queue and topic) interfaces, as used in JMS 1.0.2, so applications can still use those interfaces without the need for you to create a domain-specific connection factory, such as a queue connection factory.

General properties

Name

Provider

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Queue connection factory collection

In the administrative console page, to view this page click **Resources** → **JMS** → **Queue connection factory**.

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS queue connection factories that are available to WebSphere applications at the scope indicated by the **Scope** field.

A JMS connection factory is used to create connections to JMS destinations. When an application needs a JMS connection, an instance can be created by the factory for the JMS provider that is named in the Provider column of the list.

This type of connection factory is for applications that use the JMS 1.0.2 queue-specific interfaces.

General properties

Name

Provider

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Queue collection

In the administrative console page, to view this page click **Resources** → **JMS** → **Queue**.

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS queue destinations that are available to WebSphere applications at the scope indicated by the **Scope** field.

Use topic destination administrative objects to manage JMS queues for the JMS provider that is named in the Provider column of the list. Connections to the queue are created by a connection factory (or queue connection factory) for that JMS provider.

General properties

Name

Provider

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Topic connection factory collection

In the administrative console page, to view this page click **Resources** → **JMS** → **Topic connection factory**.

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS topic connection factories that are available to WebSphere applications at the scope indicated by the **Scope** field.

A JMS connection factory is used to create connections to JMS destinations. When an application needs a JMS connection, an instance can be created by the factory for the JMS provider that is named in the Provider column of the list.

This type of connection factory is for applications that use the JMS 1.0.2 topic-specific interfaces.

General properties

Name

Provider

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Topic collection

In the administrative console page, to view this page click **Resources** → **JMS** → **Topic**.

To browse or change the properties of a listed item, select its name in the list.

To act on one or more of the listed items, select the check boxes next to the names of the items that you want to act on, then use the buttons provided.

To change what entries are listed, or to change what information is shown for entries in the list, use the Filter settings.

This page lists the JMS topic destinations that are available to WebSphere applications at the scope indicated by the **Scope** field.

Use topic destination administrative objects to manage JMS topics for the JMS provider that is named in the Provider column of the list. Connections to the topic are created by a connection factory (or topic connection factory) for that JMS provider.

General properties

Name

Provider

Description

Scope The level to which this resource definition is visible; for example, the cell, node, cluster, or server level.

Buttons

New	Click this button to create a new JMS resource of this type.
Delete	Click this button to delete the selected items.

Choosing messaging providers for a mixed environment

If your existing or planned messaging environment involves both WebSphere MQ and WebSphere Application Server systems, choose between the default messaging provider, the WebSphere MQ messaging provider, or a mixture of the two, by considering your messaging requirements, your business environment, and the needs of each messaging application.

About this task

For messaging between application servers, with interaction with a WebSphere MQ system, you can use the default messaging provider or the WebSphere MQ provider. Neither provider is necessarily better than the other. The choice of providers depends primarily on factors relating to your business environment and planned changes to that environment, and also on what each JMS application needs to do. Moreover, these two types of messaging providers are not mutually exclusive:

- You can configure both types of providers within one cell.
- Different applications can use the same, or different, providers.

Factors relating to your business environment include the following:

- Messaging requirements
- Existing skill set
- Existing messaging infrastructure
- Planned changes to that infrastructure

Configuring and managing your messaging infrastructure is simpler if you use just one provider. If your messaging is primarily in WebSphere MQ, you should probably choose the WebSphere MQ messaging provider. Similarly, if your messaging is primarily in WebSphere Application Server, you should probably choose the default messaging provider.

If your business environment does not clearly indicate that you should use just one provider, then you should consider using a mixture of the two, and choosing the most appropriate messaging provider for each application. A useful way of doing this is to identify the types of destinations (service integration bus, or WebSphere MQ queue or topic) that the application is using. If the application uses only bus destinations, the natural choice is to use the default messaging provider (solution ″DMP″). If the application needs to communicate with one or more WebSphere MQ destinations, you can choose any of the following solutions depending upon your business environment, usage scenarios, and system topologies:

- Use the WebSphere MQ messaging provider (solution ″MQP″).
- Use the default messaging provider to integrate a WebSphere MQ server (a WebSphere MQ queue manager or queue-sharing group) as a bus member (solution ″DMP interop bus member″).
- Use the default messaging provider to integrate a WebSphere MQ network as a foreign bus, using WebSphere MQ links (solution ″DMP interop, foreign bus″).

For more information about these solutions, see Overview of interoperation with WebSphere MQ.

To help you choose between these solutions, several of the following steps contain tables in which each row represents a business or system requirement, and asterisks (*) indicate the solutions that are likely to be most effective for meeting the requirement. These tables are designed to provide general guidance, rather than to identify precisely a “right” solution. Most requirements have multiple possible solutions, and the absence of an asterisk does not necessarily mean that you cannot use that solution. To derive best guidance from using each of these tables:

- Focus on the rows that reflect your most important requirements.
- For all the rows that you consider, count the number of asterisks for each solution.

The solutions with the largest number of asterisks are likely to be the most effective.

1. If you have limited experience of WebSphere MQ or WebSphere Application Server, and are trying to decide which product best meets your messaging needs, see “Service integration and WebSphere MQ messaging - a comparison” on page 1156.

   Note: Whichever of these products you choose as the main focus for your messaging, you can still use either the default messaging provider or the WebSphere MQ provider for interoperation between the products.

2. Consider your business environment, to see if you can use just one provider.

   In deciding which provider to use, consider the following constraints:

   - The current and future messaging requirements
   - The existing messaging infrastructure
   - The skill set that you have in your organization

   If the majority of your messaging is today performed in WebSphere MQ, continue with that approach and configure WebSphere MQ as an external JMS provider (that is, use the WebSphere MQ messaging provider) in WebSphere Application Server. If the JMS requirements of your WebSphere Application Server applications are limited, it is debatable whether using a service integration bus for those applications gives sufficient benefit.

   If you have messaging applications in WebSphere Application Server that have no requirement to interoperate with your WebSphere MQ network, use the default messaging provider (the service integration bus). If your WebSphere Application Server messaging requirements demand a tighter integration with WebSphere Application Server, the service integration bus provides the following benefits:

   - Integrated administration
   - WebSphere Application Server high availability capabilities
   - WebSphere Application Server scalability

If you choose to use the default messaging provider to interoperate between service integration and WebSphere MQ, be aware that there is an added cost involved in converting messages between service integration format and WebSphere MQ format.

Also consider the following messaging scenarios:

- A large installed backbone of WebSphere MQ queue managers, perhaps with WebSphere Message Broker.

  If you want to use WebSphere Application Server to run a newly introduced messaging application, you can deploy a WebSphere Application Server (JMS) messaging application that will exchange messages with an existing application that uses a WebSphere MQ queue or topic.
- A WebSphere Application Server installation, perhaps with existing Web and enterprise applications, but no WebSphere Application Server messaging application.

  If you have no existing messaging infrastructure, you can deploy a WebSphere Application Server (JMS) messaging application to exchange messages with an existing WebSphere Application Server messaging application that uses a service integration bus destination.
- An infrastructure that uses WebSphere Application Server to connect WebSphere Application Server messaging applications.

  Introduce WebSphere Application Server (JMS) messaging between a pair of WebSphere Application Server applications.
- An infrastructure that includes both WebSphere MQ and service integration buses. This could be the result of a merger, or because the message traffic tends to be from WebSphere Application Server to WebSphere Application Server, or from WebSphere MQ to WebSphere MQ, but not typically between WebSphere Application Server and MQ.

  Deploy a WebSphere Application Server (JMS) messaging application to exchange messages with an application that uses a WebSphere MQ queue or topic.
- A WebSphere Process Server or WebSphere Enterprise Service Bus infrastructure, which uses Service Component Architecture (SCA).

  You can choose either a WebSphere MQ or a service integration bus binding for your SCA components.

3. If your business environment does not clearly indicate that you should use just one messaging provider, use a mixture of the two and choose the most appropriate provider for each application, based upon the destination types that the application uses.

   The application might need to exchange messages with existing partner applications or services that use one or more known destinations of known type. Alternatively, the partner applications or services might not yet be deployed and the choice of destination type might still be open, in which case the solution architect needs to decide how best to connect the applications or services together.

   If the application uses multiple destinations, there are four possible outcomes:

   - The application uses only bus destinations.
   - The application uses only WebSphere MQ destinations.
   - The application uses a mixture of bus and WebSphere MQ destinations.
   - The destination types are not yet known.

   Note: If there is no clear business or technical reason why the application uses WebSphere MQ destinations rather than bus destinations, and the partner application is also a WebSphere Application Server JMS application, consider migrating the existing destinations to service integration so that the application uses only bus destinations.
4. If the application uses only bus destinations, configure the application and its JMS resources to use the default messaging provider.
5. If the application uses only WebSphere MQ destinations (queues or topics), use the following checklist to determine which provider solution to use.

Table 28. Provider checklist for an application that uses only WebSphere MQ destinations.

Question:	MQP	DMP interop, bus member	DMP interop, foreign bus
Is performance critical? (If so, use WebSphere MQ directly, rather than perform message conversion.)	*		
Does the application need to send or receive large messages (that is, messages > 500k.)?	*		
Is location transparency desirable for simplifying programming and deployment of applications?		*	*
Does the application need to consume from a WebSphere MQ queue, the configuration of which is fixed? (That is, the queue cannot be moved to service integration and you do not want to deploy a push-style WebSphere MQ application to send messages to a bus destination.)	*	*	
Is the partner application a JMS application that will run outside WebSphere Application Server, as a bus or WebSphere MQ client? (Do not mix service integration and WebSphere MQ unless you have to do so; a pure WebSphere MQ or service integration solution is simpler and avoids the cost of converting messages between service integration and WebSphere MQ formats.)	*		
Is the partner application a non-JMS (non-WebSphere Application Server) application? (Wherever possible choose a pure WebSphere MQ or service integration solution. Use the MQI WebSphere MQ client, or the XMS WebSphere MQ client, or the XMS bus client depending on your API preference.)	*		
Do you prefer traffic passing between your WebSphere MQ network and WebSphere Application Server applications to be funneled into a single long-running connection?			*
Do you want to use the high availability features of WebSphere Application Server?			*
Is XA 2pc needed between the application and a WebSphere MQ queue sharing group?		*	*
Is XA 2pc needed between the application and a WebSphere MQ cluster?		*	

6. If the application uses a mixture of bus and WebSphere MQ destinations, for example consuming from service integration and sending to WebSphere MQ, then either of the default messaging provider interoperation models can support this by using a single connection factory or activation specification. Use the following checklist to help you decide between a bus member and a foreign bus solution.

Table 29. Provider checklist for an application that uses a mixture of bus and WebSphere MQ destinations.

Question:	DMP interop, bus member	DMP interop, foreign bus
Does the application need to consume from a WebSphere MQ shared queue?	*	
Is there a need to distribute work to a pool of WebSphere Application Server workers from a WebSphere MQ queue?	*	
Do you prefer traffic passing between your WebSphere MQ network and WebSphere Application Server applications to be funneled into a single long-running connection?		*
Do you need distributed WebSphere MQ in versions earlier than WebSphere Application Server Version 7 and WebSphere MQ Version 7?		*
Do you want store and forward capabilities to allow the application to continue to send messages when the WebSphere MQ queue manager is unavailable?		*

Table 29. Provider checklist for an application that uses a mixture of bus and WebSphere MQ destinations. (continued)

Question:	DMP interop, bus member	DMP interop, foreign bus
Do you prefer not to configure server connection channels? (This is because they open a port, which could be seen as a security risk.)		*
Do you prefer to define a server connection channel, rather than a pair of sender and receiver channels?	*	

7. If the destination types are not yet known, decide the relative priorities of the known concerns then use the following checklist to assess how well each of them is addressed by the possible provider solutions.

   The choice is really over what type of destinations this application should use. The destination types are not yet fixed, so any of the four solutions is possible, but as a general rule you should aim for solution “DMP” or “MQP”, because a pure WebSphere MQ or service integration solution is simpler and avoids the cost of converting messages between service integration and WebSphere MQ formats.

Table 30. Provider checklist for an application for which the destination types are not yet known.

Question:	DMP	MQP	DMP interop, bus member	DMP interop, foreign bus
Do you have an existing base of strong skills in managing WebSphere MQ?		*	*	*
Do you want management of all messaging to be handled by the WebSphere MQ team?		*		
Do you have administrators skilled in WebSphere Application Server but not in WebSphere MQ?	*			
Do you want a messaging product with a large installed base (including references) and a wide choice of ISV tools?		*		
Are you reluctant to buy a separately licensed product in addition to WebSphere Application Server?	*			
Are you reluctant to install and manage a separate product in addition to WebSphere Application Server?	*			
Are you already using WebSphere Message Broker? (If so, you need WebSphere MQ anyway).		*	*	*
Are you using the WebSphere Enterprise Service Bus to mediate messages from, or deliver them to, a WebSphere MQ queue? (For example, using WebSphere Business Integration adapters, or connecting to a service provider such as CICS.)		*		
Does the application need to send or receive large messages (that is, messages > 500k.)?		*		
Is location transparency desirable for simplifying programming and deployment of applications?	*		*	*
Do the throughput requirements need multiple parallel channels or routes?		*	*	*

Table 30. Provider checklist for an application for which the destination types are not yet known. (continued)

Question:	DMP	MQP	DMP interop, bus member	DMP interop, foreign bus
Does the application need to consume from a WebSphere MQ queue, the configuration of which is fixed? (that is, the queue cannot be moved to service integration and you don't want to deploy a push-style WebSphere MQ application to send messages to a bus destination.)	*	*	*	
Is the partner application a JMS application that will also run in WebSphere Application Server? (Service integration runs in the WebSphere Application Server application server. On distributed platforms that means it is in-process. On the z/OS platform it is in another region.)	*			
Is the partner application a JMS application that will run outside WebSphere Application Server, as a bus or WebSphere MQ client? (Do not mix service integration and WebSphere MQ unless you have to do so; a pure WebSphere MQ or service integration solution is simpler and avoids the cost of converting messages between service integration and WebSphere MQ formats.)	*	*		
Is the partner application a non-JMS (non-WebSphere Application Server) application? (Wherever possible choose a pure WebSphere MQ or service integration solution. Use the MQI WebSphere MQ client, or the XMS WebSphere MQ client, or the XMS bus client depending on your API preference.)	*	*		
Is maintenance of strict message order important?	*			
Does the application require the flexibility and convenience of a WebSphere MQ cluster? (WebSphere MQ clustering makes administration simpler and provides selective parallelism of clustered queues. That is, instances of a clustered queue can be created on any (but not necessarily all) queue managers in the WebSphere MQ cluster. Messages sent to the clustered queue can be addressed to a specific instance of the queue, or allowed to select an instance dynamically based on workload management statistics. WebSphere Application Server clustering provides some of this flexibility, but you cannot create partitions of a bus destination on a subset of the messaging engines in a cluster bus member.)		*	*	*
Does the application need the level of high availability provided by WebSphere MQ for z/OS shared queues?		*	*	*
Do you want to use the high availability or scalability features of WebSphere Application Server clustering?	*		*	*

Service integration and WebSphere MQ messaging - a comparison

If you are not already an established user of either WebSphere Application Server or WebSphere MQ, and you are considering whether the service integration platform or WebSphere MQ better meets your messaging needs, use this table to compare the main features of the two platforms.

Table 31. Comparison of service integration and WebSphere MQ main features

service integration (the default messaging provider for WebSphere Application Server)	**WebSphere MQ**
Service integration is closely integrated with WebSphere Application Server, and is a natural fit if you are using the Java Platform, Enterprise Edition (Java EE).	WebSphere MQ can connect to almost anything. It provides a very heterogeneous environment .
Service integration supports multiple languages through XMS clients, and multiple platforms.	WebSphere MQ supports multiple languages and multiple platforms.
Service integration is a single process, pure Java implementation.	WebSphere MQ has many Independent Software Vendor (ISV) tools.
Service integration provides strong performance for both persistent and non-persistent messages for JMS.	WebSphere MQ supports JMS and non-JMS messaging interfaces, and provides strong performance for non-JMS applications.
Service integration is designed for a maximum message size of about 40 megabytes on a 32-bit operating system (subject to heap usage).	WebSphere MQ supports large message sizes up to about 100 megabytes.
Service integration is tightly integrated with some Web services implementations.	WebSphere MQ is a natural fit if you are using WebSphere Message Brokers.
Service integration messaging is included in a single administrative model for WebSphere Application Server, WebSphere Enterprise Service Bus and WebSphere Process Server.	WebSphere MQ can integrate existing infrastructure and applications (for example CICS).
Service integration bus clustering is integrated with WebSphere Application Server clustering for high availability and scalability.	WebSphere MQ clustering provides selective parallelism of clustered queues.

Note: The messaging platform that you choose for a given task does not necessarily determine which JMS messaging provider you should use. For example:

- If you need to handle large messages, you probably need to use the WebSphere MQ messaging provider.
- If you are using WebSphere Message Brokers, you can use either the default messaging provider or the WebSphere MQ messaging provider.

Learning about messaging with WebSphere Application Server

Use this topic to learn about the use of asynchronous messaging for enterprise applications with WebSphere Application Server.

About this task

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) and J2EE Connector Architecture (JCA) programming interfaces. These interfaces provide a common way for Java programs (clients and Java EE applications) to create, send, receive, and read asynchronous requests, as messages. For additional information about messaging resources, see: Messaging resources.

- “Types of messaging providers” on page 1158
- “Styles of messaging in applications” on page 1162
- “JMS interfaces - explicit polling for messages” on page 1163
- “Message-driven beans - automatic message retrieval” on page 1164
- Configuring JMS resources for the WebSphere MQ messaging provider
- “Message-driven beans - JCA components” on page 1165

- “WebSphere Application Server cloning and WebSphere MQ clustering” on page 1169
- “Asynchronous messaging - security considerations” on page 1171

Types of messaging providers

You can configure any of three main types of Java Message Service (JMS) providers in WebSphere Application Server: The WebSphere Application Server default messaging provider (which uses service integration as the provider), the WebSphere MQ messaging provider (which uses your WebSphere MQ system as the provider) and third-party messaging providers (which use another company’s product as the provider).

Overview

WebSphere Application Server supports JMS messaging through the following providers:

- “Default messaging provider”
- “WebSphere MQ messaging provider” on page 1159
- “Third-party messaging provider” on page 1160

Your applications can use messaging resources from any of these JMS providers. The choice of provider is most often dictated by requirements to use or integrate with an existing messaging system. For example, you might already have a messaging infrastructure based on WebSphere MQ. In this case, you can either connect directly using the WebSphere MQ messaging provider, or configure a service integration bus with links to a WebSphere MQ network and then access the bus through the default messaging provider.

Note: For backwards compatibility with earlier releases, WebSphere Application Server Version 7 also includes support for the Version 5 default messaging provider and the Version 6 WebSphere MQ messaging provider. This support enables your applications that still use these resources to communicate with Version 5 and Version 6 nodes in Version 7 mixed cells.

Default messaging provider

If you mainly want to use messaging between applications in WebSphere Application Server, perhaps with some interaction with a WebSphere MQ system, the default messaging provider is the natural choice. This provider is based on service integration technologies and is fully integrated with the WebSphere Application Server runtime environment. To use the default messaging provider, you configure the following resources:

- Configure a connection factory or activation specification to connect your application to a service integration bus.
- Assign a queue or topic to a bus destination on the bus. This topic or queue is then available to any application that can access the bus destination.

A service integration bus comprises messaging engines that run in WebSphere Application Server processes and dynamically connect to one another using dynamic discovery. A messaging application connects to the bus through a messaging engine. Messaging engines use WebSphere Application Server clustering to provide high availability and scalability, and they use the same management framework as the rest of WebSphere Application Server. Bus client applications can run from within WebSphere Application Server (JMS), or run as stand-alone Java clients (using the J2SE Client for JMS) or run as non-Java clients (XMS).

In a pure WebSphere Application Server environment, service integration provides the following benefits:

- An all-Java implementation for JMS within a single server process, yielding good performance for local JMS traffic.

Note: On z/OS systems, WebSphere Application Server employs a multi-region architecture. The service integration runtime artefacts are in the same application server, but not in the same region as the applications. This contrasts with distributed platforms, on which the application server is a single process and the service integration runtime artefacts and the applications all run in that process.

- Direct delivery of messages to a consumer application that is ready to receive them, without first writing the messages to disk.
- Avoidance of copying and serialization of message payloads, if the application is written in accordance with the typical pattern of a messaging application. For more information, see Why and when to pass the JMS message payload by reference.
- Full integration of the administration model with WebSphere Application Server, providing management of single and federated nodes within a cell.
- Full use of the provided WebSphere Application Server infrastructure for tracing and threading, and for a range of communications protocols with a single point of administration.
- Extensive interoperation with WebSphere MQ networks.

There are two ways in which you can connect to a WebSphere MQ system through the default messaging provider:

- Connect a bus to a WebSphere MQ network, using a *WebSphere MQ link*. The WebSphere MQ network appears to the service integration bus as a foreign bus, and the service integration bus appears to WebSphere MQ as another queue manager.
- Connect directly to WebSphere MQ queues located on WebSphere MQ queue managers or (for WebSphere MQ for z/OS) queue-sharing groups, using a *WebSphere MQ server* bus member. Each WebSphere MQ queue is made available at a queue-type destination on the bus.

For more information about these two approaches, see Overview of interoperation with WebSphere MQ.

To configure and manage messaging with the default messaging provider, see Managing messaging with the default messaging provider.

WebSphere MQ messaging provider

If your business also uses WebSphere MQ, and you want to integrate WebSphere Application Server messaging applications into a predominately WebSphere MQ network, choose the WebSphere MQ messaging provider, which allows you to define resources for connecting to any queue manager on the WebSphere MQ network.

WebSphere MQ is characterized as follows:

- Messaging is handled by a network of queue managers, each running in its own set of processes and having its own administration.
- Features such as shared queues (on WebSphere MQ for z/OS) and WebSphere MQ clustering simplify administration and provide dynamic discovery.
- Many IBM and partner products support WebSphere MQ with (for example) monitoring and control, high availability and clustering.
- WebSphere MQ clients can run within WebSphere Application Server (JMS), or almost any other messaging environment using a variety of APIs.

For more information about the WebSphere MQ messaging provider, see “Enhanced features of the WebSphere MQ messaging provider” on page 1160. To configure and manage messaging with this provider, see Managing messaging with the WebSphere MQ messaging provider. For more information about scenarios and considerations for using WebSphere MQ with WebSphere Application Server, see the white papers and IBM Redbooks publications provided by WebSphere MQ; for example, through the WebSphere MQ library Web page at http://www.ibm.com/software/ts/mqseries/library/.

Third-party messaging provider

You can configure any third-party messaging provider that supports the JMS Version 1.1 unified connection factory. You might want to do this, for example, because of existing investments.

To administer a third-party messaging provider, use the resource adaptor or client supplied by the third party. You can still use the WebSphere Application Server administrative console to administer the JMS connection factories and destinations that are within WebSphere Application Server, but you cannot use the administrative console to administer the JMS provider itself, or any of its resources that are outside of WebSphere Application Server.

To use message-driven beans (MDBs), third-party messaging providers must include Application Server Facility (ASF), an optional feature that is part of the JMS Version 1.1 specification, or use an inbound resource adapter that conforms to the Java EE Connector Architecture (JCA) Version 1.5 specification.

To work with a third-party provider, see “Managing messaging with a third-party messaging provider” on page 1173.

Enhanced features of the WebSphere MQ messaging provider

The WebSphere MQ messaging provider enables WebSphere Application Server applications and clients to connect to and use WebSphere MQ resources in a JMS-compliant manner. For WebSphere Application Server Version 7.0, this provider includes the enhanced features described in this topic.

Overview

For WebSphere Application Server Version 7.0, the WebSphere MQ messaging provider has enhanced administrative options supporting the following functions:

- “WebSphere MQ channel compression”
- “WebSphere MQ client channel definition table”
- “Client channel exits” on page 1161
- “Transport-level encryption using SSL” on page 1161
- “Automatic selection of the WebSphere MQ transport type” on page 1162

WebSphere MQ channel compression

Data sent over the network between WebSphere Application Server and WebSphere MQ can be compressed, reducing the amount of data that is transferred. Channel compression can be beneficial in the following situations:

- If a cost is incurred that is proportional to the amount of data transferred over a network. For example, nodes in a network might span a leased line for which a utilization charge is applied.
- If the rate at which messaging data can be transferred across a network is the limiting factor in the performance of an application.
- If compressing the data might reduce the cost of its encryption and decryption.

To use WebSphere MQ channel compression, configure the message compression properties of an existing connection factory or activation specification. For more information, see the appropriate step within Configuring JMS resources for the WebSphere MQ messaging provider.

For more information, see the WebSphere MQ topic Channel compression.

WebSphere MQ client channel definition table

The client channel definition table reduces the effort required to configure a connection to a queue manager. Your WebSphere MQ administrator can create a single table of all the WebSphere MQ channels

supported by queue managers in the enterprise, then in WebSphere Application Server you configure a connection to a queue manager by pointing to a table entry and providing any additional information not already contained within the table.

You can also use the client channel definition table to provide a basic failover capability, by specifying that a connection is attempted against several entries in the table. Each specified entry is tried in turn until a queue manager connection is successfully established.

You can use the client channel definition table, with WebSphere MQ messaging provider activation specifications and connection factories, to select the client channel definition to use when establishing a connection to WebSphere MQ. The table can be configured to select from a number of queue managers, depending on their availability.

When you use a client channel definition table, you must bear in mind the following considerations:

- If your client channel definition table can select from more than one queue manager, you might not be able to recover global transactions. Activation specifications and connection factories that specify a client channel definition table must either do so without ambiguity as to the target queue manager, or must avoid using the resources with applications that enlist in global transactions.
- If your client channel definition table contains entries that reference native WebSphere MQ channel exits, the use of these entries is not supported in the WebSphere Application Server environment.

For more information about client channel definition tables, see the developerWorks article WebSphere MQ V6 Java and JMS clients and the client channel definition table, and the WebSphere MQ topic Client channel definition table.

To use a client channel definition table, point to it when you create a new activation specification or connection factory.

Client channel exits

Client channel exits are pieces of Java code that you develop, and that are executed in the application server at key points during the life cycle of a WebSphere MQ channel. Your code can change the runtime characteristics of the communications link between the WebSphere MQ messaging provider and the WebSphere MQ queue manager.

Note: Only client channel exits written in Java are supported for use within the WebSphere Application Server environment.

For more information about client channel exits, see the WebSphere MQ topic Channel exit programs. For a list of the channel exits that work with the WebSphere MQ messaging provider, see the client connection channel row of the table in the WebSphere MQ topic What are channel exit programs?.

To use client channel exits, configure the client transport properties of an existing connection factory or activation specification.

Transport-level encryption using SSL

Transport-level encryption using SSL is the supported way to configure SSL for JMS resources associated with the WebSphere MQ messaging provider. The SSL configuration is associated with the communication link for the connection factory or activation specification. You either define the SSL information in the connection factory, or your WebSphere MQ administrator defines the SSL information in an associated client channel definition table.

Automatic selection of the WebSphere MQ transport type

The WebSphere MQ messaging provider supports the following ways to connect to a WebSphere MQ queue manager:

- Bindings mode (or *call attach*)

  Bindings mode attachment is only possible if the queue manager is located on the same physical machine as the WebSphere Application Server. Bindings mode attachment, where available, typically offers better performance.
- Client mode (or *socket attach*)

  Client mode attachment can be used wherever the WebSphere MQ queue manager and WebSphere Application Server can establish a network connection to one another.
- Bindings mode, then client mode (automatic selection)

  This method tries a bindings mode connection first and, if that fails, a client mode connection is tried.

Every node in a WebSphere Application Server cluster shares identical configuration information. With automatic selection of the WebSphere MQ transport type, all the servers in a cluster can be configured to automatically select their transport. This has the effect that any clustered server that is co-located with a queue manager establishes a bindings mode connection to the queue manager, whereas other servers in the cluster establish client mode connection to the queue manager.

Styles of messaging in applications

This topic describes the ways that applications can use point-to-point and publish/subscribe messaging.

Applications can use the following styles of asynchronous messaging:

Point-to-Point
: Point-to-point applications use *queues* to pass messages between each other. The applications are called point-to-point, because a client sends a message to a specific queue and the message is picked up and processed by a server listening to that queue. It is common for a client to have all its messages delivered to one queue. Like any generic mailbox, a queue can contain a mixture of messages of different types.

Publish/subscribe
: Publish/subscribe systems provide named collection points for messages, called *topics*. To send messages, applications publish messages to topics. To receive messages, applications subscribe to topics; when a message is published to a topic, it is automatically sent to all the applications that are subscribers of that topic. By using a topic as an intermediary, message publishers are kept independent of subscribers.

Both styles of messaging can be used in the same application.

Applications can use asynchronous messaging in the following ways:

One-way
: An application sends a message, and does not want a response. This pattern of use is often referred to as a *datagram*.

Request and response
: An application sends a request to another application and expects to receive a response in return.

One-way and forward
: An application sends a request to another application, which sends a message to yet another application.

These messaging techniques can be combined to produce a variety of asynchronous messaging scenarios.

For more information about how such messaging scenarios are used by WebSphere enterprise applications, see the following topics:

- “JMS interfaces - explicit polling for messages”
- “Message-driven beans - automatic message retrieval” on page 1164

For more information about these messaging techniques and the Java Message Service (JMS), see Sun’s Java Message Service (JMS) specification documentation (http://developer.java.sun.com/developer/technicalArticles/Networking/messaging/).

For more information about message-driven bean and inbound messaging support, see Sun’s Enterprise JavaBeans specification (http://java.sun.com/products/ejb/docs.html).

For information about JCA inbound messaging processing, see Sun’s J2EE Connector Architecture specification (http://java.sun.com/j2ee/connector/download.html).

JMS interfaces - explicit polling for messages

This topic provides an overview of applications that use JMS interfaces to explicitly poll for messages on a destination then retrieve messages for processing by business logic beans (enterprise beans).

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interfaces. JMS provides a common way for Java programs (clients and Java EE applications) to create, send, receive, and read asynchronous requests, as JMS messages.

The base support for asynchronous messaging using JMS, shown in the following figure, provides the common set of JMS interfaces and associated semantics that define how a JMS client can access the facilities of a JMS provider. This enables WebSphere J2EE applications, as JMS clients, to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics).

Applications can use both point-to-point and publish/subscribe messaging (referred to as “messaging domains” in the JMS specification), while supporting the different semantics of each domain.

WebSphere Application Server supports applications that use JMS 1.1 domain-independent interfaces (referred to as the “common interfaces” in the JMS specification). With JMS 1.1, the preferred approach for implementing applications is to use the common interfaces. The JMS 1.1 common interfaces provide a simpler programming model than domain-specific interfaces. Also, applications can create both queues and topics in the same session and coordinate their use in the same transaction.

The common interfaces are also parents of domain-specific interfaces. These domain-specific interfaces (provided for JMS 1.0.2 in WebSphere Application Server Version 5) are supported only to provide inter-operation and backward compatibility with applications that have already been implemented to use those interfaces.

A WebSphere application can use the JMS interfaces to explicitly poll a JMS destination to retrieve an incoming message, then pass the message to a business logic bean. The business logic bean uses standard JMS calls to process the message; for example, to extract data or to send the message on to another JMS destination.

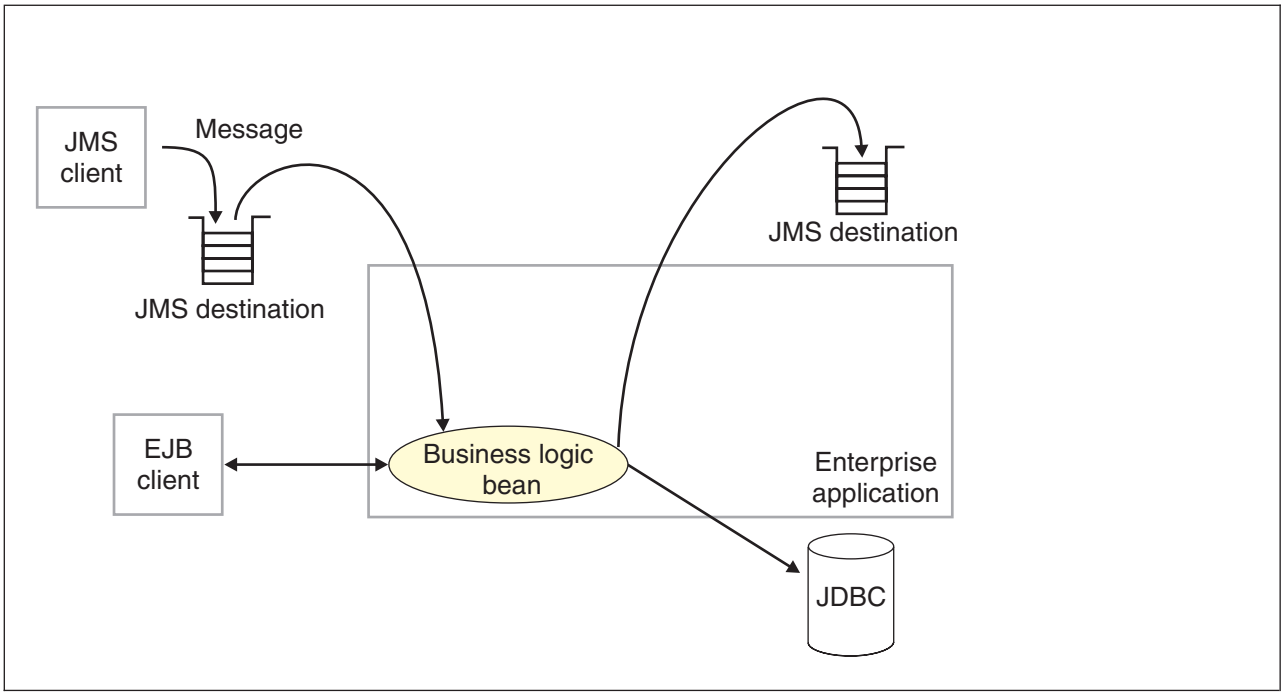


Figure 4. Asynchronous messaging using JMS. This figure shows an enterprise application polling a JMS destination to retrieve an incoming message, which it processes with a business logic bean. The business logic bean uses standard JMS calls to process the message; for example, to extract data or to send the message on to another JMS destination. For more information, see the text that accompanies this figure.

WebSphere applications can use standard JMS calls to process messages, including any responses or outbound messaging. Responses can be handled by an enterprise bean acting as a sender bean, or handled in the enterprise bean that receives the incoming messages. Optionally, this process can use two-phase commit within the scope of a transaction.

WebSphere applications can also use message-driven beans, as described in related topics.

For more details about JMS, see Sun's Java Message Service (JMS) specification documentation.

Message-driven beans - automatic message retrieval

WebSphere Application Server supports the use of message-driven beans as asynchronous message consumers.

Messaging with message-driven beans is shown in the figure Figure 5 on page 1165.

A client sends messages to the destination (or endpoint) for which the message-driven bean is deployed as the message listener. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination. The message-driven bean implements some business logic to process incoming messages on the destination.

Message-driven beans can be configured as listeners on a Java EE Connector Architecture (JCA) 1.5 resource adapter or against a listener port (as in WebSphere Application Server Version 5). With a JCA 1.5 resource adapter, message-driven beans can handle generic message types, not just JMS messages. This makes message-driven beans suitable for handling generic requests inbound to WebSphere Application Server from enterprise information systems through the resource adapter. In the JCA 1.5 specification, such message-driven beans are commonly called *message endpoints* or simply *endpoints*.

All message-driven beans must implement the MessageDrivenBean interface. For JMS messaging, a message-driven bean must also implement the message listener interface, javax.jms.MessageListener.

You are recommended to develop a message-driven bean to delegate the business processing of incoming messages to another enterprise bean, to provide clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client.

Messages arriving at a destination being processed by a message-driven bean have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component. For more information about EJB security, see Securing enterprise bean applications.

For JMS messaging, message-driven beans can use a JMS provider that has a JCA 1.5 resource adapter, for example the default messaging provider that is part of WebSphere Application Server or the WebSphere MQ messaging provider. With a JCA 1.5 resource adapter, you deploy EJB 2.1 message-driven beans as JCA 1.5-compliant resources, to use a J2C activation specification. If the JMS provider does not have a JCA 1.5 resource adapter, for example the V5 default messaging provider, you must configure JMS message-driven beans against a listener port.

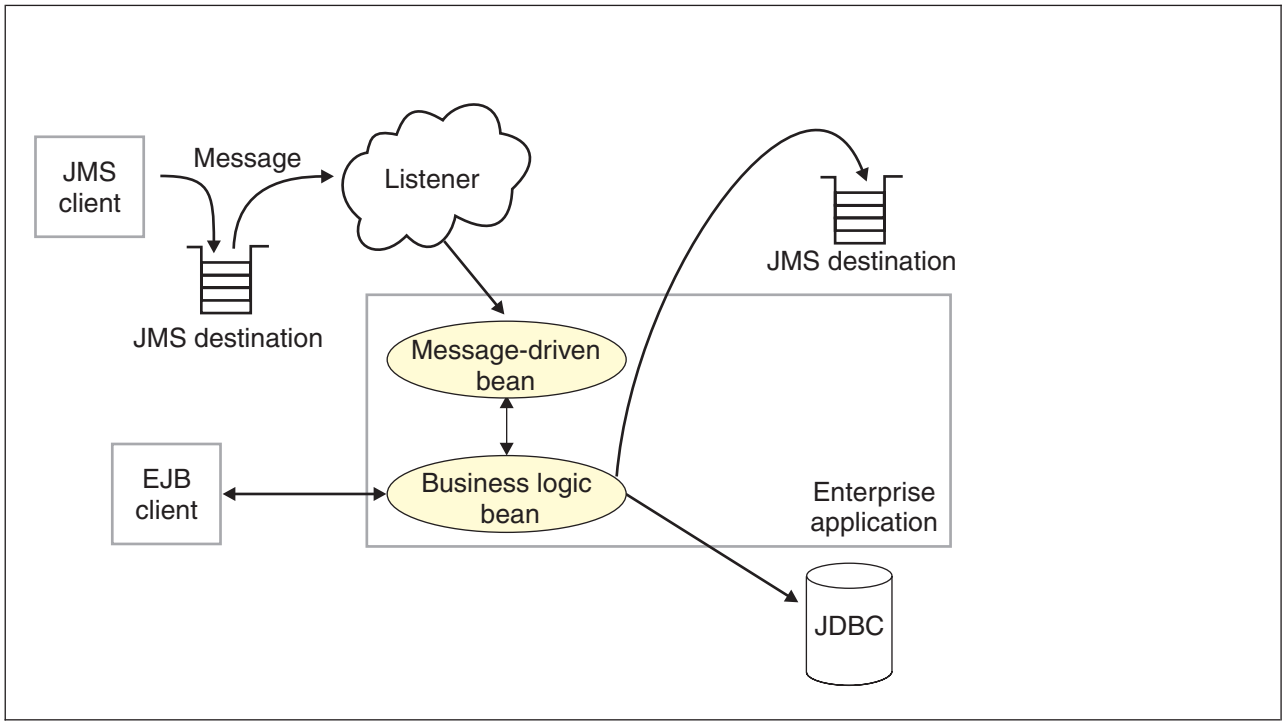


Figure 5. Messaging with message-driven beans. This figure shows an incoming message being passed automatically to the onMessage() method of a message-driven bean that is deployed as a listener for the destination. The message-driven bean processes the message, in this case passing the message on to a business logic bean for business processing. For more information, see the text that accompanies this figure.

Message-driven beans - JCA components

This topic provides an overview of the administrative components that you configure for message-driven beans (MDBs) as listeners on a Java EE Connector Architecture (JCA) 1.5 resource adapter.

Components for a JCA resource adapter

When a resource adapter is installed, it provides definitions and classes for administered objects such as activation specifications. The administrator creates and configures activation specifications with Java™ Naming and Directory Interface (JNDI) names that are then available for applications to use.

The JCA resource adapter uses an activation specification to configure a specific endpoint. Each application that configures one or more endpoints must specify the resource adapter that sends messages to the endpoint

The application uses the activation specification to provide the configuration properties related to the processing of inbound messages.

JMS components used with a JCA messaging provider

MDBs that implement the javax.jms.MessageListener interface can be used for JMS messaging.

An application that uses JMS messaging needs access at runtime to configured objects such as connection factories and destinations.

When the JMS provider is the default JMS provider or the WebSphere MQ messaging provider, the administrator configures these objects for the JMS provider. For example, to configure a JMS activation specification for the WebSphere MQ messaging provider, in the WebSphere Application Server administrative console navigate to **Resources** → **JMS** → **Activation specifications**.

Otherwise the administrator configures these objects for the JMS resource adapter, which connects the application to a JMS provider, by navigating to **Resources** → **Resource Adaptors**.

If the application contains one or more MDBs, the administrator must configure either a JMS activation specification or a message listener port. For JCA-compliant messaging providers, the administrator usually configures an activation specification. But for the WebSphere MQ messaging provider there is a choice; the administrator can configure an activation specification or, for compatibility with previous versions of WebSphere Application Server, the administrator can configure a message listener port.

The JMS activation specification provides the deployer with information about the configuration properties of an MDB related to the processing of the inbound messages. For example, a JMS activation specification specifies the name of the service integration bus to connect to, information about the message acknowledgement modes, message selectors, destination types, and whether or not durable subscriptions are shared across connections with members of a server cluster.

The activation specification identifies a JMS destination by specifying its JNDI name. The MDB acts as a listener on a specific JMS destination.

The JMS destination refers to a service integration bus destination (or WebSphere MQ destination) which the administrator must also configure. For more information about JMS resources and service integration, see Learning about the default messaging provider.

J2C activation specification configuration and use

This topic provides an overview about the configuration and use of J2C activation specifications, used in the deployment of message-driven beans for JCA 1.5 resources.

J2C activation specifications are part of the configuration of inbound messaging support that can be part of a JCA 1.5 resource adapter. Each JCA 1.5 resource adapter that supports inbound messaging defines one or more types of message listener in its deployment descriptor (`messagelistener` in the ra.xml). The message listener is the interface that the resource adapter uses to communicate inbound messages to the message endpoint. A message-driven bean (MDB) is a message endpoint and implements one of the message listener interfaces provided by the resource adapter. By allowing multiple types of message listener, a resource adapter can support a variety of different protocols. For example, the interface javax.jms.MessageListener, is a type of message listener that supports JMS messaging. For each type of message listener that a resource adapter implements, the resource adapter defines an associated activation specification (`activationspec` in the ra.xml). The activation specification is used to set configuration properties for a particular use of the inbound support for the receiving endpoint.

When an application containing a message-driven bean is deployed, the deployer must select a resource adapter that supports the same type of message listener that the message-driven bean implements. As part of the message-driven bean deployment, the deployer needs to specify the properties to set on the J2C activation specification. Later, during application startup, a J2C activation specification instance is created, and these properties are set and used to activate the endpoint (that is, to configure the resource adapter's inbound support for the specific message-driven bean).

Applications with message-driven beans can also specify all, some, or none of the configuration properties needed by the ActivationSpec class, to override those defined by the resource adapter-scoped definition. These properties, specified as activation-config properties in the application's deployment descriptor, are configured when the application is assembled. To change any of these properties requires redeploying the application. These properties are unique to this applications use and are not shared with other message-driven beans. Any properties defined in the application's deployment descriptor take precedence over those defined by the resource adapter-scoped definition. This allows application developers to choose the best defaults for their applications.

WebSphere Application Server activation specification optional binding properties

Binding properties that you can specify for activation specifications to be deployed on WebSphere Application Server.

J2C authentication alias

If you provide values for user name and password as custom properties on an activation specification, you may not want to have those values exposed in clear text for security reasons. WebSphere security allows you to securely define an authentication alias for such cases. Configuration of activation specifications, both as an administrative object and during application deployment, enable you to use the authentication alias instead of providing the user name and password.

If you set the authentication alias field, then you should not set the user name and password custom properties fields. Also, authentication alias properties set as part of application deployment take precedence over properties set on an activation specification administrative object.

Only the authentication alias is ever written to file in an unencrypted form, even for purposes of transaction recovery logging. The security service is used to protect the real user name and password.

During application startup, when the activation specification is being initialized as part of endpoint activation, the server uses the authentication alias to retrieve the real user name and password from security then set it on the activation specification instance.

Destination JNDI name

For resource adapters that support JMS you need to associate javax.jms.Destinations with an activation specification, such that the resource adapter can service messages from the JMS destination. In this case, the administrator configures a J2C Administered Object which implements the javax.jms.Destination interface and binds it into JNDI.

You can configure a J2C Administered Object to use an ActivationSpec class that implements a setDestination(javax.jms.Destination) method. In this case, you can specify the destination JNDI name (that is, the JNDI name for the J2C Administered object that implements the javax.jms.Destination).

A destination JNDI name set as part of application deployment take precedence over properties set on an activation specification administrative object.

During application startup, when the activation specification is being initialized as part of endpoint activation, the server uses the destination JNDI name to look up the destination administered object then set it on the activation specification instance.

Message-driven beans - transaction support

Message-driven beans can handle messages on destinations (or endpoints) within the scope of a transaction.

Transaction handling when using the Message Listener Service with WebSphere MQ JMS

There are three possible cases, based on the MDB deployment descriptor setting you choose: container-managed transaction (required), container-managed transaction (not supported), and bean-managed transaction. For related information see “Message-driven bean deployment descriptor properties” on page 1259.

Container-managed transaction (required)

In this situation, the application server starts a global transaction before it reads any incoming message from the destination, and before the onMessage() method of the MDB is invoked by the application server. This means that other EJBs that are invoked in turn by the message, and interactions with resources such as databases can all be scoped inside this single global transaction, within which the incoming message was obtained.

If this application flow completes successfully, the global transaction is committed. If the flow does not complete successfully, (if the transaction is marked for rollback or if a runtime exception occurs), the transaction is rolled back, and the incoming message is rolled back onto the MDB destination.

Container-managed transaction (not supported)

In this situation there is obviously no global transaction, however the JMS provider can still deliver a message from an MDB destination to the application server in a unit of work. You can consider this as a local transaction, because it does not involve other resources in its transactional scope.

The application server acknowledges message delivery on successful completion of the onMessage() dispatch of the MDB (using the acknowledgement mode specified by the assembler of the MDB).

However, the application server does not perform an acknowledge, if an unchecked runtime exception is thrown from the onMessage() method. So, does the message roll back onto the MDB destination (or is it acknowledged and deleted)?

The answer depends on whether or not a syncpoint is used by the WebSphere MQ JMS provider and can vary depending on the operating platform (in particular the z/OS operating platform can impart different behavior here).

If WebSphere MQ establishes a syncpoint around the MDB message consumption in this container-managed transaction (not supported) case, the message is rolled back onto the destination after an unchecked exception.

If a syncpoint is not used, then the message is deleted from the destination after an unchecked exception.

For related information, see the technote ’MDB behavior is different on z/OS than on distributed when getting nonpersistent messages within syncpoint’ at http://www.ibm.com/support/docview.wss?uid=swg21231549.

Bean-managed transaction

This situation is similar to the container-managed transaction (not supported) case. Even though there might be a user transaction in this case, any user transaction started within the onMessage dispatch of the MDB does not include consumption of the message from the message-driven bean destination within the transaction scope. To do this, use the container-managed transaction (required) scenario.

Message redelivery

In each of the previous three cases, a message that is rolled back onto the MDB destination is eventually re-dispatched. If the original rollback was due to a temporary system problem, you would expect the re-dispatch of the MDB with this message to succeed on re-dispatch. If, however, the rollback was due to a specific message-related problem, the message would repeatedly be rolled back and re-dispatched. This would be an inefficient use of processing resources.

The application server handles this scenario which is known as a poison message scenario, by tracking the frequency with which a message is dispatched, and by stopping the associated listener port after a specified number of redeliveries has occurred. This is a configurable value on the Maximum Retries property on a listener port. For more information, see “Listener port settings” on page 1227.

Note: A Maximum Retries value of zero stops the listener port after a single failure to successfully complete an onMessage().

Because stopping the listener port stops the processing for all MDBs mapped to that listener port, this solution is rather unspecific. Instead of relying on the WebSphere Application Server message listener service to stop the listener port if a poison message scenario occurs, the other solution is to set up a backout queue (BOQUEUE), and a backout threshold value (BOTHRESH). If you do this, WebSphere MQ handles the poison message. For more information on handling poison messages, see the *WebSphere MQ Using Java* book, which is available from the WebSphere MQ library page at http://www.ibm.com/software/integration/wmq/library/.

Inbound resource adapter transaction handling

An MDB can be configured for bean or container transaction handling. The owner of the resource adapter owner must tell the MDB developer how to set up the MDB for transaction handling.

WebSphere Application Server cloning and WebSphere MQ clustering

Here is a summary of information about using WebSphere Application Server horizontal cloning with WebSphere MQ server clustering support. It describes a scenario that shows how the message listener service can be configured to take advantage of WebSphere MQ server clustering and provides some information about how to resolve potential runtime failures in the clustering scenario.

The information in this topic is based on the scenario shown in the figure Figure 6 on page 1170.

Note: WebSphere MQ server clustering is only available with the full WebSphere MQ product installed as a JMS provider.

Each JMS listener is used to retrieve messages from a destination defined to the server. In the following information the listener configurations are the same for each WebSphere application server. Each application server host contains a WebSphere application server and a WebSphere MQ server. If a host is only used to distribute messages, it only contains a WebSphere MQ server. There can be many servers defined in the configuration, although for simplicity the information in this topic is based on a scenario containing only three servers as shown in the following figure:

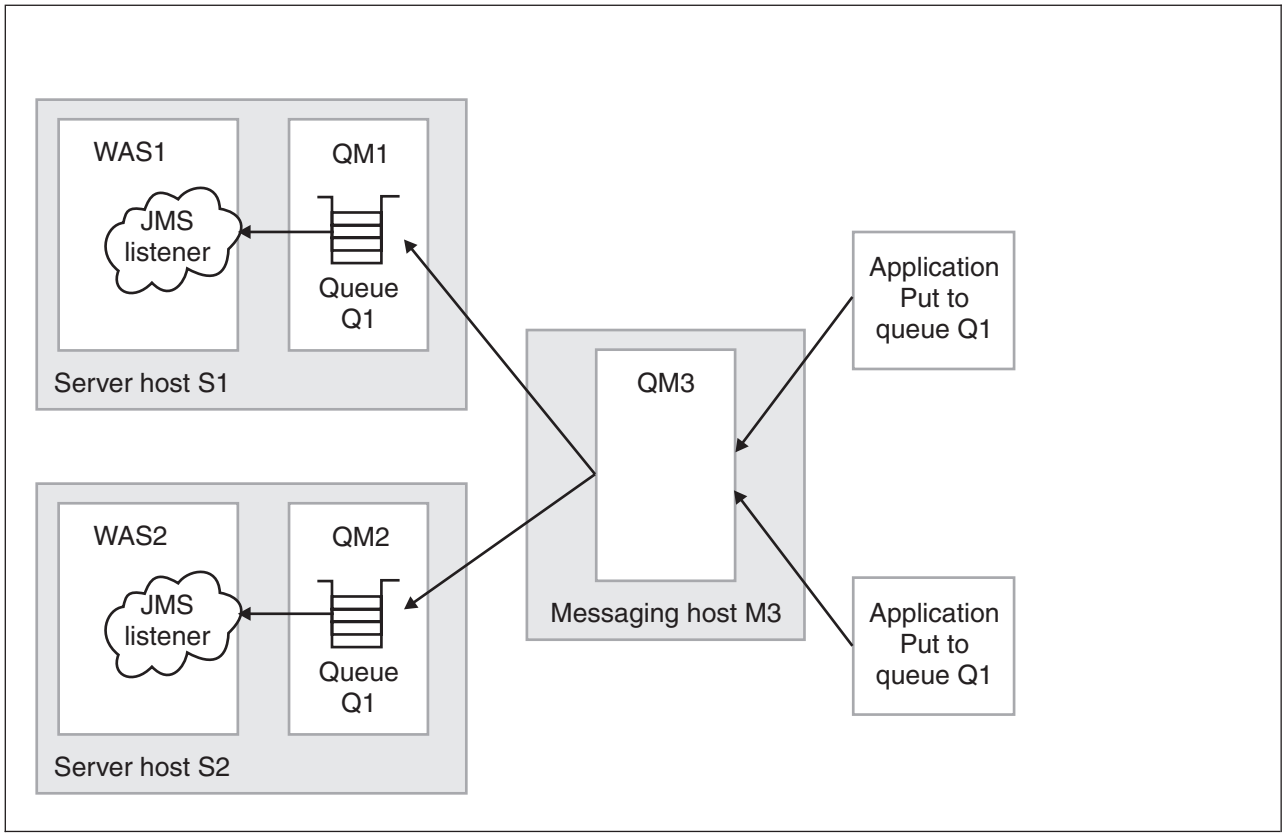


Figure 6. WebSphere Application Server horizontal cloning with WebSphere MQ clustered queues. This figure shows two WebSphere Application Server hosts, with horizontal clustering, and a messaging host used to distribute messages for WebSphere MQ server clustering. For more information, see the text that accompanies this figure.

The scenario shown in the previous figure comprises the following three hosts:

- Server host S1 contains the following servers:

  WebSphere MQ server.

  The server is defined to have a queue manager, QM1, and a local queue, Q1. The queue manager belongs to a cluster. The queue is populated by the WebSphere MQ server located on host M3. Applications can add messages directly to the queue, Q1, but would not be subjected to the control of the WebSphere MQ cluster.

  WebSphere Application Server

  This contains a cloned application server, WAS1, which is configured with a JMS listener. The listener is configured to retrieve messages from JMS destination Q1.
- Server host S2 contains the following servers:

  WebSphere MQ server.

  The server is defined to have a queue manager, QM2, and a local queue, Q1. The queue manager belongs to the same cluster as QM1 on host S1. As with QM1, the queue is populated by the WebSphere MQ server located on host M3. Applications can add messages directly to the queue, Q1, but would not be subjected to the control of the MQ cluster.

  WebSphere Application Server

  This contains a cloned application server, WAS2 (identical to WAS1 on host S1), which is configured with a JMS listener. The listener is configured to retrieve messages from JMS destination Q1.
- Messaging host M3 contains the following servers:

  WebSphere MQ server.

  The server is defined to have a queue manager, QM3, which also belongs to the same cluster as QM1 and QM2. Applications add messages to the queue manager and queue Q1. The cluster to which this queue manager belongs causes messages to be distributed to all other queue managers in the cluster which have queue Q1 defined.

Note: Queue Q1 is not defined as a local queue on this host. If the queue was defined locally, then messages would remain on the server for local processing; messages would not be distributed by the queue manager cluster control to the other queue managers in the cluster that do have the queue defined.

This host does not have a WebSphere application server defined. All message retrieval processing is performed by the other two application servers on hosts S1 and S2.

Recovery scenarios

There are several failure scenarios that could occur with the clustering configuration; for example:

- WAS server failures.

  In this scenario the failure of any single WebSphere application server results in the messages for the specified destination remaining on the queue, until the server is restarted.
- WebSphere MQ Queue Manager failures.

  There are two different failures to consider:
  1. Failure of a queue manager on the same host as a WebSphere application server (for example, failure of QM2 on host S2). In this case messages are delivered to the other available application servers, until the WebSphere MQ server is back online, when messages are processed as expected.
  2. Failure of the messaging host M3 and its queue manager, QM3. In this case, the result of an outage is more significant because no messages are delivered to the other queue managers in the cluster. In a fully-deployed and scaled production system, host M3 would not be designed to be a single point of failure, and additional messaging servers would be added to the cluster configuration.

Asynchronous messaging - security considerations

This topic describes considerations that you should be aware of if you want to use security for asynchronous messaging with WebSphere Application Server.

Security for messaging is enabled only when WebSphere Application Server administrative security is enabled. In this case:

- JMS connections made to the JMS provider are authenticated.
- Access to JMS resources owned by the JMS provider is controlled by access authorizations.
- Requests to create new connections to the JMS provider must provide a user ID and password for authentication.
- The user ID and password do not need to be provided by the application.

Note: Users exploiting the connector thread identity support do not have to provide a user ID and password for authentication (see the subsequent links).

If authentication is successful, then the JMS connection is created; if the authentication fails then the connection request is ended.

Standard J2C authentication is used for a request to create a new connection to the JMS provider. If your resource authentication (res-auth) is set to Application, set the alias in the Component-managed Authentication Alias. If the application that tries to create a connection to the JMS provider specifies a user ID and password, those values are used to authenticate the creation request. If the application does not specify a user ID and password, the values defined by the Component-managed Authentication Alias are used. If the connection factory is not configured with a Component-managed Authentication Alias, then you receive a runtime JMS exception when an attempt is made to connect to the JMS provider.

If your res-auth property is set to Container, you can set the Container-managed Authentication Alias on the Connection Factory, and specify the user ID and password within this alias. If you are running in Bindings transport mode (that is, the TransportType property on the Connection Factory is set to

"BINDINGS"), then you can also exploit the connector thread identity function instead of specifying a container-managed alias. For more information, see "Connection thread identity" on page 1035 and "Using thread identity support" on page 1036.

For more information about configuring a transport type of *bindings then client* or *bindings*, refer to Configuring the WebSphere MQ messaging provider with native libraries information.

If you are working with a message-driven bean and are configuring a message-driven bean listener under the Message Listener Service, see "Configuring security for message-driven beans that use listener ports" on page 1234 for more information.

Note: In addition to the authorization needed for creating a connection to a JMS provider that you set up when creating a JMS Connection, you also typically need authorization to access specific JMS resources associated with that JMS Provider for example, permission to write to a given queue. For more information about using WebSphere MQ as your JMS provider, see the WebSphere MQ documentation library page at http://www.ibm.com/software/integration/wmq/library/.

Note:

1. User IDs longer than 12 characters cannot be used for authentication with the Version 5 default messaging provider or WebSphere MQ. For example, the default Windows NT user ID, **Administrator**, is not valid for use because it contains 13 characters. Therefore, an authentication alias for a WebSphere JMS provider or WebSphere MQ connection factory must specify a user ID no longer than 12 characters.

Authorization to access messages stored by the default messaging provider is controlled by authorization to access the service integration bus destinations on which the messages are stored. For information about authorizing permissions for individual bus destinations, see Administering destination roles.

Other ways of managing messaging

For messaging between application servers, most requirements are best met by either the default messaging provider or the WebSphere MQ messaging provider. However, you can instead use a third-party messaging provider (that is, use another company's product as the provider). For backwards compatibility with earlier releases, there is also support for the V5 default messaging provider.

Before you begin

If you are not sure which provider combination is best suited to your needs, see "Types of messaging providers" on page 1158.

About this task

Enterprise applications in WebSphere Application Server can use asynchronous messaging through services based on Java Message Service (JMS) messaging providers and their related messaging systems. These messaging providers conform to the JMS Version 1.1 specification.

The choice of provider depends on what your JMS application needs to do, and on other factors relating to your business environment and planned changes to that environment.

- Choose a third-party messaging provider.

  You can configure any third-party messaging provider that supports the JMS Version 1.1 unified connection factory. You might want to do this, for example, because of existing investments.

  To administer a third-party messaging provider, you use the resource adaptor or client supplied by the third party. You can still use the WebSphere Application Server administrative console to administer the

JMS connection factories and destinations that are within WebSphere Application Server, but you cannot use the administrative console to administer the JMS provider itself, or any of its resources that are outside of WebSphere Application Server.

To use message-driven beans (MDBs), third-party messaging providers must include Application Server Facility (ASF), an optional feature that is part of the JMS Version 1.1 specification, or use an inbound resource adapter that conforms to the Java EE Connector Architecture (JCA) Version 1.5 specification.

To work with a third-party provider, see "Managing messaging with a third-party messaging provider."

- Choose the (deprecated) V5 default messaging provider.

  This deprecated provider is identical to the WebSphere Application Server Version 5 default provider. Only the name has changed. It provides backwards compatibility that enables WebSphere Application Server Version 6 or later applications to connect to WebSphere Application Server Version 5 resources in a mixed cell. It also allows WebSphere Application Server Version 5 applications to connect to WebSphere Application Server Version 6 or later resources in a mixed cell. To configure and manage messaging to interoperate with WebSphere Application Server Version 5, see "Maintaining Version 5 default messaging resources" on page 1182.

Managing messaging with a third-party messaging provider

Enabling your messaging applications to use JMS resources provided by a third-party messaging provider other than WebSphere MQ.

Before you begin

For messaging between application servers, perhaps with some interaction with a WebSphere MQ system, you can use the default messaging provider. To integrate WebSphere Application Server messaging into a predominately WebSphere MQ network, you can use the WebSphere MQ messaging provider. You can also use a third-party messaging provider as described in this topic. To choose the provider that is best suited to your needs, see "Choosing a messaging provider" on page 1145.

About this task

You can install a messaging provider other than the default messaging provider or the WebSphere MQ messaging provider.

WebSphere Application Server applications can use the JMS 1.1 interfaces or JMS 1.0.2 interfaces to access JMS resources provided by a third-party messaging provider, and you can use the administrative console to administer the JMS connection factories and destinations for the provider.

In a mixed-version WebSphere Application Server deployment manager cell, you can administer third-party messaging resources on Version 7, Version 6 and Version 5 nodes. For Version 5 nodes, the administrative console presents the subset of resources and properties that are applicable to WebSphere Application Server Version 5.

To use a third-party messaging provider with WebSphere Application Server, complete one or more of the following steps:

- Define a third-party messaging provider.
- List third-party JMS messaging resources.
- Configure JMS resources for a third-party messaging provider.

Defining a third-party messaging provider

Use this task to define a third-party messaging provider to WebSphere Application Server, for use instead of the default messaging provider or WebSphere MQ messaging provider.

Before you begin

Before you configure a third-party messaging provider, you might want to check whether your requirement can be met by the default messaging provider or the WebSphere MQ messaging provider that are supplied with WebSphere Application Server. To choose the provider that is best suited to your needs, see “Choosing a messaging provider” on page 1145.

About this task

You can configure any third-party messaging provider that supports the JMS Version 1.1 unified connection factory. You might want to do this, for example, because of existing investments.

To administer a third-party messaging provider, you use the resource adaptor or client supplied by the third-party. You can still use the WebSphere Application Server administrative console to administer the JMS connection factories and destinations that are within WebSphere Application Server, but you cannot use the administrative console to administer the JMS provider itself, or any of its resources that are outside of WebSphere Application Server.

To use message-driven beans (MDBs), third-party messaging providers must include Application Server Facility (ASF), an optional feature that is part of the JMS Version 1.1 specification, or use an inbound resource adapter that conforms to the J2EE Connector Architecture (JCA) Version 1.5 specification.

To define a new third-party messaging provider to WebSphere Application Server, use the administrative console to complete the following steps:

1. In the navigation pane, click **Resources** → **JMS** → **JMS Providers** . The existing messaging providers are displayed, including the default messaging provider and the WebSphere MQ messaging provider.
2. To define a new third-party messaging provider, click **New** in the content pane. Otherwise, to change the definition of an existing messaging provider, click the name of the provider.
3. Specify the following required properties. You can specify other properties, as described in a later step.

   Name The name by which this messaging provider is known for administrative purposes within IBM WebSphere Application Server.

   External initial context factory
   The Java classname of the initial context factory for the JMS provider.

   External provider URL
   The JMS provider URL for external JNDI lookups.
4. Optional: Click **Apply**. This enables you to specify additional properties.
5. Optional: Specify other properties for the messaging provider.

   Under Additional Properties, you can use the **Custom Properties** link to specify custom properties for your initial context factory, in the form of standard javax.naming properties.
6. Click **OK**.
7. Save the changes to the master configuration.
8. To have the changed configuration take effect, stop then restart the application server.

What to do next

You can now configure JMS resources for your messaging provider, as described in “Configuring JMS resources for a third-party messaging provider” on page 1181.

Listing third-party JMS messaging resources

Use this task with the WebSphere Application Server administrative console to display administrative lists of JMS resources provided by a messaging provider other than the default messaging provider or the WebSphere MQ messaging provider.

About this task

You can use the WebSphere Application Server administrative console to display lists of the following types of JMS resources provided by a 3rd party messaging provider. You can use the panels displayed to select JMS resources to administer, or to create or delete JMS resources (where appropriate).

To display administrative lists of JMS resources for a third-party messaging provider, complete the following steps:

1. Start the administrative console.
2. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
3. Click the name of the third-party messaging provider.
4. In the content pane, under Additional Resources, click the link for the type of JMS resource. For more information about the detailed settings displayed for each resource type, see the related reference topics.

JMS provider collection:

Use this panel to list JMS providers, or to select a JMS provider to view or change its configuration properties.

To view this administrative console page, click **Resources** → **JMS** → **JMS providers**

To view or change the properties of a JMS provider or its resources, select its name in the list displayed.

To define a new third-party messaging provider, click **New**.

To act on one or more of the JMS providers listed, click the check boxes next to the names of the objects that you want to act on, then use the buttons provided.

Name The name by which this JMS provider is known for administrative purposes.

Description

A description of this JMS provider for administrative purposes.

For related information about JMS messaging providers and asynchronous messaging, see

- "Types of messaging providers" on page 1158
- "Asynchronous messaging in WebSphere Application Server using JMS" on page 309

Third-party JMS connection factory collection:

The JMS connection factories configured for a third-party messaging provider for both point-to-point and publish/subscribe messaging. Use this panel to create or delete JMS connection factories, or to select a connection factory to browse or change its configuration properties.

This panel shows a list of the JMS connection factories configured for a third-party messaging provider, with a summary of their configuration properties.

To view this administrative console page, use the administrative console to complete the following steps:

1. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
2. In the content pane, click the name of the third-party messaging provider that you want to support the JMS connection factory.
3. Under Additional Properties, click **Connection factories**.

To define a new JMS connection factory, click **New**.

To view or change the properties of a JMS connection factory, select its name in the list displayed.

To act on one or more of the JMS connection factories listed, click the check boxes next to the names of the objects that you want to act on, then use the buttons provided.

Third-party JMS connection factory settings:

Use this panel to browse or change the configuration properties of a JMS connection factory configured for use with a third-party messaging provider. These configuration properties control how connections are created to the JMS destinations on the provider.

A JMS connection factory is used to create connections to JMS destinations. The JMS connection factory is created by the associated JMS provider.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
2. If appropriate, in the content pane, change the scope of the third-party messaging provider.
3. In the content pane, click the name of the messaging provider that supports the JMS connection factory.
4. Under Additional Properties, click **Connection factories**.
5. Click the name of the JMS connection factory that you want to work with.

A JMS connection factory for a third-party messaging provider (that is, a provider other than the default, the V5 default or the WebSphere MQ messaging provider) has the following properties:

Scope:

Specifies the level to which this resource definition is visible to applications.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

The scope displayed is for information only, and cannot be changed on this panel. If you want to browse or change this resource (or other resources) at a different scope, change the scope on the messaging provider settings panel, then click **Apply**, before clicking the link for the type of resource.

Data type String

Name:

The name by which this JMS connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the associated messaging provider.

Data type String

Type:

Whether this connection factory is for creating JMS queue destinations or JMS topic destinations.

Select one of the following options:

QUEUE
A JMS queue connection factory for point-to-point messaging.

TOPIC
A JMS topic connection factory for publish/subscribe messaging.

JNDI name:

The JNDI name that is used to bind the connection factory into the WebSphere Application Server name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

Category:

A category used to classify or group this connection factory, for your IBM WebSphere Application Server administrative records.

Data type String

External JNDI name:

The JNDI name that is used to bind the connection factory into the name space of the third-party messaging provider.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Component-managed Authentication Alias:

This alias specifies a user ID and password to be used to authenticate connection to a JMS provider for application-managed authentication.

This property provides a list of the J2C authentication data entry aliases that have been defined to WebSphere Application Server. You can select a data entry alias to be used to authenticate the creation of a new connection to the JMS provider.

If you have enabled administrative security for WebSphere Application Server, select the alias that specifies the user ID and password used to authenticate the creation of a new connection to the JMS provider. The use of this alias depends on the resource authentication (res-auth) setting declared in the connection factory resource reference of an application component's deployment descriptors.

Container-managed Authentication Alias:

This alias specifies a user ID and password to be used to authenticate connection to a JMS provider for container-managed authentication.

This property provides a list of the J2C authentication data entry aliases that have been defined to WebSphere Application Server. You can select a data entry alias to be used to authenticate the creation of a new connection to the JMS provider.

If you have enabled administrative security for WebSphere Application Server, select the alias that specifies the user ID and password used to authenticate the creation of a new connection to the JMS provider. The use of this alias depends on the resource authentication (res-auth) setting declared in the connection factory resource reference of an application component's deployment descriptors.

Mapping-Configuration Alias:

The module used to map authentication aliases.

This field provides a list of the modules that have been configured on the **Global Security** → **JAAS Configuration** → **Application Logins Configuration** property. For more information about the mapping configurations, see Java Authentication and Authorization service configuration entry settings.

Data type Enum

Default Null

Range

ClientContainer
The client container maps authentication aliases.

WSLogin
The WSLogin module maps authentication aliases.

DefaultPrincipalMapping
The JAAS configuration maps an authentication alias to its userid and password.

Connection pool:

Specifies an optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the messaging provider to improve performance. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Session pool:

An optional set of session pool settings.

This link provides a panel of optional connection pool properties, common to all J2C connectors.

The application server pools connections and sessions with the messaging provider to improve performance. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Custom properties:

An optional set of name and value pairs for custom properties passed to the messaging provider.

Third-party JMS destination collection:

The JMS destinations configured in an associated third-party messaging provider for point-to-point and publish/subscribe messaging. Use this panel to create or delete JMS destinations, or to select a JMS destination to browse or change its configuration properties.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. In the content pane, click the name of the third-party messaging provider that you want to support the JMS destination.
3. Under Additional Properties, click **Queues** for point-to-point messaging or **Topics** for publish/subscribe messaging.

To define a new JMS destination, click **New**.

To view or change the properties of a JMS destination, select its name in the list displayed.

To act on one or more of the JMS destinations listed, click the check boxes next to the names of the objects that you want to act on, then use the buttons provided.

Third-party JMS destination settings:

Use this panel to browse or change the configuration properties of the selected JMS destination for use with an associated third-party messaging provider.

A JMS destination is used to configure the properties of a JMS destination for the associated third-party messaging provider (that is, not the default messaging provider or the WebSphere MQ messaging provider). Connections to the JMS destination are created by the associated JMS connection factory.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. In the content pane, click the name of the third-party messaging provider that you want to support the JMS destination.
3. Under Additional Properties, click **Queues** for point-to-point messaging or **Topics** for publish/subscribe messaging.
4. Click the name of the JMS destination that you want to work with.

A JMS destination for use with a third-party messaging provider has the following properties.

Scope:

Specifies the level to which this resource definition is visible to applications.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

The scope displayed is for information only, and cannot be changed on this panel. If you want to browse or change this resource (or other resources) at a different scope, change the scope on the messaging provider settings panel, then click **Apply**, before clicking the link for the type of resource.

Data type String

Name:

The name by which the queue is known for administrative purposes within WebSphere Application Server.

Data type String

Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for publish/subscribe).

Select one of the following options:
Queue
A JMS queue destination for point-to-point messaging.
Topic A JMS topic destination for publish/subscribe messaging.

JNDI name:

The JNDI name that is used to bind the queue into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Description:

A description of the queue, for administrative purposes

Data type String

Category:

A category used to classify or group this queue, for your WebSphere Application Server administrative records.

Data type String

External JNDI name:

The JNDI name that is used to bind the queue into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Configuring JMS resources for a third-party messaging provider

Use the following tasks to configure the JMS connection factories and destinations for a third-party messaging provider (that is, a messaging provider other than the default messaging provider or the WebSphere MQ messaging provider).

Before you begin

You only need to complete these tasks if your WebSphere Application Server environment uses a third-party messaging provider to support enterprise applications that use JMS. To enable use of a third-party messaging provider, you must have installed and configured the messaging provider, as described in Defining a third-party messaging provider.

About this task

To configure JMS resources for a third-party messaging provider, complete the following tasks:

- Configure a JMS connection factory for a third-party messaging provider.
- Configure a JMS destination for a third-party messaging provider.

Configuring a JMS connection factory for a third-party messaging provider:

Use this task to browse or change the properties of a JMS connection factory for use with a JMS messaging provider other than the default, V5 default or WebSphere MQ messaging providers.

About this task

To configure a JMS connection factory for use with a third-party messaging provider, use the administrative console to complete the following steps:

1. Display the third-party messaging provider. In the navigation pane, click **Resources** → **JMS** → **JMS Providers** .
2. Select the third-party provider for which you want to configure a connection factory.
3. Optional: Change the **Scope** setting to the level at which the connection factory is visible to applications.
4. In the content pane, under Additional Properties, click **Connection factories** This displays a table listing any existing JMS connection factories, with a summary of their properties.
5. To browse or change an existing JMS connection factory, click its name in the list. Otherwise, to create a new connection factory, complete the following steps:
   a. Click **New** in the content pane.
   b. Specify the following required properties. You can specify other properties, as described in a later step.

      Name The name by which this JMS connection factory is known for administrative purposes within IBM WebSphere Application Server.

      Type Select whether the connection factory is for JMS queues (QUEUE) or JMS topics (TOPIC).

      JNDI Name
      The JNDI name that is used to bind the JMS connection factory into the WebSphere Application Server name space.

      External JNDI Name
      The JNDI name that is used to bind the JMS connection factory into the name space of the messaging provider.

   c. Click **Apply**. This defines the JMS connection factory to WebSphere Application Server, and enables you to browse or change additional properties.
6. Optional: Change properties for the JMS connection factory, according to your needs.

7. Click **OK**.
8. Save any changes to the master configuration.
9. To have the changed configuration take effect, stop then restart the application server.

Configuring a JMS destination for a third-party messaging provider:

Use this task to browse or change the properties of a JMS destination for use with a third-party messaging provider (that is, a provider other than the default messaging provider or the WebSphere MQ messaging provider).

About this task

To configure a JMS destination for use with a third-party messaging provider, use the administrative console to complete the following steps:

1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. Click the name of the third-party messaging provider.
3. In the content pane, under Additional Properties, click **Queues** for point-to-point messaging or **Topics** for publish/subscribe messaging. This displays a table listing any existing JMS destinations, with a summary of their properties.
4. To browse or change an existing JMS destination, click its name in the list. Otherwise, to create a new destination, complete the following steps:
   a. Click **New** in the content pane.
   b. Specify the following required properties. You can specify other properties, as described in a later step.

      Name The name by which this JMS destination is known for administrative purposes within WebSphere Application Server.

      Type Select whether the destination is for JMS queues (QUEUE) or JMS topics (TOPIC).

      JNDI Name
      The JNDI name that is used to bind the JMS destination into the WebSphere Application Server name space.

      External JNDI Name
      The JNDI name that is used to bind the JMS destination into the name space of the messaging provider.

   c. Click **Apply**. This defines the JMS destination to WebSphere Application Server, and enables you to browse or change additional properties.
5. Optional: Change properties for the JMS destination, according to your needs.
6. Click **OK**.
7. Save any changes to the master configuration.
8. To have the changed configuration take effect, stop then restart the application server.

Maintaining Version 5 default messaging resources

This topic is the entry-point into a set of topics about maintaining messaging resources provided for WebSphere Application Server Version 5.1 applications by the default messaging provider.

About this task

JMS applications running on WebSphere Application Server Version 5.1 can use JMS resources provided by the default messaging provider in WebSphere Application Server Version 7. You can use the

WebSphere Application Server administrative console to manage the JMS connection factories and destinations for WebSphere Application Server Version 5.1 applications. Such JMS resources are maintained as *V5 Default Messaging* resources.

In a deployment manager cell, you can use V5 Default Messaging resources to maintain Version 5 default messaging resources for Version 7, Version 6 and Version 5 nodes in the cell.

- V5 Default Messaging configured against a Version 7 node provides a JMS transport to a messaging engine of a service integration bus that supports the default messaging provider in WebSphere Application Server Version 7. The messaging engine emulates the service of a JMS server running on WebSphere Application Server Version 5.1.
- V5 Default Messaging configured against a Version 5.1 node provides a JMS transport to a JMS server running on WebSphere Application Server Version 5.1.

You can also use the administrative console to manage a JMS server on a Version 5.1 node.

- List Version 5 default messaging resources.
- Configure Version 5 default JMS resources.
- Manage Version 5 JMS servers in a deployment manager cell.
- Configure authorization security for a Version 5 default messaging provider.
- Tune JMS destinations.

Listing Version 5 default messaging resources

Use the WebSphere Application Server administrative console to list JMS resources for the Version 5 default messaging provider, for administrative purposes.

About this task

You use the WebSphere Application Server administrative console to list JMS resources, if you want to view, modify or delete any of the following resources:

- Connection factory (unified)
- Queue connection factory
- Topic connection factory
- Queue
- Topic
- Activation specification

When you use the Administrative Console to locate these resources, two different navigation pathways are available:

- Provider-centric navigation. This lets you view all providers (or just those for a specified scope if required), then navigate to a specific resource for a specific provider. This is the traditional way of navigating to a resource when you know which provider owns it. Any navigation that starts with **Resources** → **JMS** → **JMS providers** is provider-centric.
- Resource-centric navigation lets you view all resources of a specified type, then navigate to a resource. This is useful if you want to find a resource, but you do not know which provider owns it (you can list all resources of a given type across all scopes, for all providers, in a single panel). Any navigation that follows the pattern **Resources** → **JMS** → **[resource]**, where [resource] is one of the resource types listed above is resource-centric.

You can use either of these navigation pathways to locate resources of any type.

- Using provider-centric navigation, for example to navigate to a specified connection factory
  1. Start the WebSphere Application Server administrative console.

2. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**. This opens the providers collection which lists all currently configured providers across all scopes (you can modify the scope if required).
3. From the providers collection, select the required provider. This opens the configuration tab for that provider. The configuration tab contains a set of links to all the resources owned by that provider.
4. From the configuration tab, click the link for a resource type, for example the connection factories link. This opens the connection factories collection which lists all the connection factories for that provider.
5. From the connection factories collection, select the required connection factory. You can now view and work with the connection factory's properties

- Using resource-centric navigation, for example to navigate to a specified connection factory
  1. Start the WebSphere Application Server administrative console.
  2. In the navigation pane, expand **Resources** → **JMS** → **Connection factories**. This opens the connection factories collection which lists all the connection factories across all providers.
  3. From the connection factories collection, select the required connection factory. You can now view and work with the connection factory's properties.

JMS provider settings:

Use this panel to view the configuration properties of a selected JMS provider. You cannot change the properties of a default messaging provider or a WebSphere MQ messaging provider.

To view this page, use the administrative console to complete one of the following steps:

- In the navigation pane, click **Resources** → **JMS** → **JMS providers**. This displays a list of JMS providers in the content pane. For each JMS provider in the list, the entry indicates the *scope* level at which JMS resource definitions are visible to applications. You can create the same type of JMS provider at different Scope settings, to offer JMS resources at different levels of visibility to applications.
- If you want to manage JMS resources that are defined at a different scope setting, change the **Scope** setting to the required level.
- In the Providers column of the list displayed, click the name of a JMS provider.

If you want to browse or change JMS resources of the JMS provider, click the link for the type of resource under Additional Properties. For more information about the administrative console panels for the types of JMS resources, see the related topics.

For default messaging providers and WebSphere MQ messaging providers, only the scope, name, and description properties are displayed for information only. You cannot change these properties.

For another type of JMS provider that you have defined yourself, extra properties are displayed that you can change.

The default messaging provider is installed and runs as part of WebSphere Application Server, and is based on service integration technologies.

Scope:

The level to which this resource definition is visible; the cell, node, or server level.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes. For more information about the scope setting, see Scope settings.

Name:

The name by which the JMS provider is known for administrative purposes.

Data type String

Default

- Default messaging provider.

  For JMS resources to be provided by a service integration bus, as part of WebSphere Application Server.
- *My JMSprovider*

  For JMS resources to be provided by your own JMS provider; not the default messaging provider or WebSphere MQ. You assign the name, in this example “My JMSprovider”, when you define the JMS provider to WebSphere Application Server. You must have installed and configured your own JMS provider before applications can use the JMS resources.
- WebSphere MQ messaging provider

  For JMS resources to be provided by WebSphere MQ. You must have installed and configured WebSphere MQ before applications can use the JMS resources.
- V5 default messaging provider.

  For JMS resources to be provided by a WebSphere Application Server Version 5 node in a deployment manager cell.

Description:

A description of the JMS provider, for administrative purposes within WebSphere Application Server.

Data type String

Classpath:

A list of paths or JAR file names which together form the location for the JMS provider classes. Each class path entry is on a separate line (separated by using the Enter key) and must not contain path separator characters (such as ’;’ or ’: ’). Class paths can contain variable (symbolic) names to be substituted using a variable map. Check your driver installation notes for specific JAR file names that are required.

This property does not apply to default messaging providers or WebSphere MQ providers.

Data type String

Native library path:

An optional path to any native libraries (*.dll, *.so). Each native path entry is on a separate line (separated by using the Enter key) and must not contain path separator characters (such as ’;’ or ’: ’). Native paths can contain variable (symbolic) names to be substituted using a variable map.

This property does not apply to default messaging providers or WebSphere MQ providers.

Data type String

External initial context factory:

The Java classname of the initial context factory for the JMS provider.

This property does not apply to default messaging providers or WebSphere MQ providers.

For example, for an LDAP service provider the value has the form: `com.sun.jndi.ldap.LdapCtxFactory`.

Data type	String
Default	Null

External provider URL:

The JMS provider URL for external JNDI lookups.

This property does not apply to default messaging providers or WebSphere MQ providers.

For example, an LDAP URL for a messaging provider has the form: `ldap://hostname.company.com/contextName`.

Data type	String
Default	Null

Version 5 JMS server collection:

On a WebSphere Application Server Version 5 node, a JMS server provides the functions of the JMS provider. Use this panel to list JMS servers on WebSphere Application Server Version 5 nodes within the administration domain, or to select a JMS server to view or change its configuration properties.

There can be at most one JMS server on each Version 5 node in the administration domain, and any application server within the domain can access JMS resources served by any JMS server on any node in the domain.

To view this page, use the administrative console to complete the following step:
1. In the navigation pane, select **Servers** → **Version 5 JMS Servers**.

To browse or change the properties of a JMS server, select its name in the list displayed.

To act on one or more of the JMS servers listed, click the check box next to the server name, then use the buttons provided.

Version 5 JMS server settings:

The JMS functions on a Version 5 node in a deployment manager cell are served by a JMS server. Use this panel to view or change the configuration properties of the selected JMS server.

JMS servers are supported only to aid migration of WebSphere Application Server Version 5 nodes to WebSphere Application Server Version 6.

You can use this panel to configure a general set of JMS server properties, which add to the default values of properties configured automatically for the Version 5 default messaging provider.

Before you set any of the values on this panel, configure the JMS Server for each node by using the WebSphere Application Server Profile Management Tool described in the topic “Using the Profile Management Tool″ or by using the zpmt command described in the topic “Configuring z/OS application-serving environments with the zpmt command″.

Name:

The name by which the JMS server is known for administrative purposes within IBM WebSphere Application Server.

This name should not be changed.

Data type	String
Units	Not applicable
Default	WebSphere Internal JMS Server
Range	Not applicable

Description:

A description of the JMS server, for administrative purposes within IBM WebSphere Application Server.

This string should not be changed.

Data type	String
Default	WebSphere Internal JMS Server

Short Name: Specifies the short name of the server. For WebSphere Application Server for z/OS, the server short name must be unique within a cell. The short name is also the default z/OS job name, which identifies the server to the native facilities of the operating system, such as Workload Manager (WLM), Automatic Restart Manager, SAF (for example, RACF), started task control, and others.

The system assigns a default short name that is automatically unique within the cell.

Unique Id: Specifies a unique identifier for this server.

The unique ID property is read only. The system automatically generates the value.

Queue Names:

The names of the queues hosted by this JMS server. Each queue name must be added on a separate line.

Each queue listed in this field must have a separate queue administrative object with the same administrative name. To make a queue available to applications, define a WebSphere queue and add its name to this field on the JMS Server panel for the host on which you want the queue to be hosted.

Data type	String
Units	Queue name
Range	Each entry in this field is a queue name of up to 45 characters, which must match exactly (including use of upper- and lowercase characters) the WebSphere queue administrative object defined for the queue.

Initial State:

The state that you want the JMS server to have when it is next restarted.

Data type	Enum
Default	Started

Range

Started
The JMS server is started automatically.

Stopped
The JMS server is not started automatically. If any deployed enterprise applications are to use JMS server functions provided by the JMS server, the system administrator must start the JMS server manually or select the Started value of this property then restart the JMS server.

To restart a JMS server on a Version 5 node, stop then restart that JMS server.

Version 5 WebSphere queue connection factory settings:

Use this panel to browse or change the configuration properties of the selected JMS queue connection factory for point-to-point messaging for use by WebSphere Application Server Version 5 applications.

A WebSphere queue connection factory is used to create JMS connections to the default messaging provider for use by WebSphere Application Server Version 5 applications.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. **(Optional)** In the content pane, change the **Scope** setting to the level at which JMS resources are visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource.
3. In the content pane, click the name of the V5 default messaging provider that you want to support the JMS destination.
4. Under Additional Resources, click **Queue connection factories**. This displays a list of any existing JMS queue connection factories.
5. Click the name of the JMS queue connection factory that you want to work with.

A queue connection factory for the embedded WebSphere JMS provider has the following properties:

Scope:

Specifies the level to which this resource definition is visible to applications.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

The scope displayed is for information only, and cannot be changed on this panel. If you want to browse or change this resource (or other resources) at a different scope, change the scope on the messaging provider settings panel, then click **Apply**, before clicking the link for the type of resource.

Data type String

Name:

The name by which this JMS queue connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type String
Default Null

JNDI name:

The JNDI name that is used to bind the JMS connection factory into the name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

Category:

A category used to classify or group this connection factory, for your IBM WebSphere Application Server administrative records.

Data type String

Node:

The WebSphere node name of the administrative node where the JMS server runs for this connection factory. Connections created by this factory connect to that JMS server.

Data type Enum
Default Null
Range Pull-down list of Version 5 nodes in the WebSphere administrative domain.

Component-managed Authentication Alias:

This alias specifies a user ID and password to be used to authenticate connection to the messaging provider for application-managed authentication.

This property provides a list of the J2C authentication data entry aliases that have been defined to WebSphere Application Server. You can select a data entry alias to be used to authenticate the creation of a new connection to the JMS provider.

If you have enabled administrative security for WebSphere Application Server, select the alias that specifies the user ID and password used to authenticate the creation of a new connection to the messaging provider. The use of this alias depends on the resource authentication (res-auth) setting declared in the connection factory resource reference of an application component's deployment descriptors.

Note: User IDs longer than 12 characters cannot be used for authentication with the Version 5 default messaging provider. For example, the default Windows NT user ID, **Administrator**, is not valid because it contains 13 characters. Therefore, an authentication alias for a WebSphere queue connection factory must specify a user ID no longer than 12 characters.

Container-managed Authentication Alias:

This alias specifies a user ID and password to be used to authenticate connection to the messaging provider for container-managed authentication.

The specification of a login configuration and associated properties on the component resource reference determines the container-managed authentication strategy when the res-auth value is Container. If the 'DefaultPrincipalMapping' login configuration is used, the associated property is a container-managed authentication alias. This field is used only in the absence of a loginConfiguration on the component resource reference. To define a new alias, see the related item J2EE Connector Architecture (J2C) authentication data entries.

This property provides a list of the J2C authentication data entry aliases that have been defined to WebSphere Application Server. You can select a data entry alias to be used to authenticate the creation of a new connection to the JMS provider.

If you have enabled administrative security for WebSphere Application Server, select the alias that specifies the user ID and password used to authenticate the creation of a new connection to the messaging provider. The use of this alias depends on the resource authentication (res-auth) setting declared in the connection factory resource reference of an application component's deployment descriptors.

Note: User IDs longer than 12 characters cannot be used for authentication with the Version 5 default messaging provider. For example, the default Windows NT user ID, **Administrator**, is not valid because it contains 13 characters. Therefore, an authentication alias for a WebSphere topic connection factory must specify a user ID no longer than 12 characters.

Mapping-Configuration Alias:

The module used to map authentication aliases.

This field is deprecated in 6.0. The specification of a login configuration and associated properties on the component resource reference determines the container-managed authentication strategy when the res-auth value is Container. This field is used only in the absence of a loginConfiguration on the component resource reference.

This field provides a list of the modules that have been configured on the **Security** → **JAAS Configuration** → **Application Logins Configuration** property. For more information about the mapping configurations, see Java Authentication and Authorization service configuration entry settings.

Data type Enum

Default Null

Range

ClientContainer
The client container maps authentication aliases.

WSLogin
The WSLogin module maps authentication aliases.

DefaultPrincipalMapping
The JAAS configuration maps an authentication alias to its userid and password.

XA Enabled:

Specifies whether the connection factory is for XA or non-XA coordination of messages and controls if the application server uses XA QCF/TCF. Enable XA if multiple resources are used in the same transaction.

If you clear this checkbox property (for non-XA coordination), the JMS session is still enlisted in a transaction, but uses the resource manager local transaction calls (session.commit and session.rollback) instead of XA calls. This can lead to an improvement in performance. However, this means that only a single resource can be enlisted in a transaction in WebSphere Application Server.

Last participant support enables you to enlist one non-XA resource with other XA-capable resources.

For a WebSphere Topic Connection Factory with the **Port** property set to `DIRECT` this property does not apply, and always adopts non-XA coordination.

Data type	Checkbox
Default	Selected (enabled for XA coordination)
Range	**Selected** The connection factory is enabled for XA-coordination of messages **Cleared** The connection factory is not enabled for XA coordination of messages
Recommended	Do not enable XA coordination when the message queue or topic received is the only resource in the transaction. Enable XA coordination when other resources, including other queues or topics, are involved.

Connection pool:

Specifies an optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the messaging provider to improve performance. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Session pool:

An optional set of session pool settings.

This link provides a panel of optional connection pool properties, common to all J2C connectors.

The application server pools connections and sessions with the messaging provider to improve performance. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

WebSphere topic connection factory settings:

Use this panel to browse or change the configuration properties of the selected JMS topic connection factory for publish/subscribe messaging by WebSphere Application Server Version 5 applications.

A WebSphere topic connection factory is used to create JMS connections to the default messaging provider for use by WebSphere Application Server Version 5 applications.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. **(Optional)** In the content pane, change the **Scope** setting to the level at which JMS resources are visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource.
3. In the content pane, click the name of the V5 default messaging provider that you want to support the JMS destination.
4. Under Additional Resources, click **Topic connection factories**. This displays a list of any existing JMS topic connection factories.
5. Click the name of the JMS topic connection factory that you want to work with.

A JMS topic connection factory for use with the Version 5 default messaging provider has the following properties.

Scope:

Specifies the level to which this resource definition is visible to applications.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

The scope displayed is for information only, and cannot be changed on this panel. If you want to browse or change this resource (or other resources) at a different scope, change the scope on the messaging provider settings panel, then click **Apply**, before clicking the link for the type of resource.

Data type String

Name:

The name by which this JMS topic connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type String
Default Null

JNDI name:

The JNDI name that is used to bind the topic connection factory into the name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Description:

A description of this topic connection factory for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

Category:

A category used to classify or group this topic connection factory, for your IBM WebSphere Application Server administrative records.

Data type String

Node:

The WebSphere node name of the administrative node where the JMS server runs for this connection factory. Connections created by this factory connect to that JMS server.

Data type Enum
Default Null
Range Pull-down list of nodes in the WebSphere administrative domain.

Port:

Which of the two ports that connections use to connect to the JMS server. The QUEUED port is for full-function JMS publish/subscribe support, the DIRECT port is for non-persistent, non-transactional, non-durable subscriptions only.

Note: Message-driven beans cannot use the direct listener port for publish/subscribe support. Therefore, any topic connection factory configured with **Port** set to `Direct` cannot be used with message-driven beans.

Data type Enum
Units Not applicable
Default QUEUED
Range
QUEUED
The listener port used for full-function JMS-compliant, publish/subscribe support.
DIRECT
The listener port used for direct TCP/IP connection (non-transactional, non-persistent, and non-durable subscriptions only) for publish/subscribe support.

Component-managed Authentication Alias:

This alias specifies a user ID and password to be used to authenticate connection to the messaging provider for application-managed authentication.

This property provides a list of the J2C authentication data entry aliases that have been defined to WebSphere Application Server. You can select a data entry alias to be used to authenticate the creation of a new connection to the JMS provider.

If you have enabled administrative security for WebSphere Application Server, select the alias that specifies the user ID and password used to authenticate the creation of a new connection to the messaging provider. The use of this alias depends on the resource authentication (res-auth) setting declared in the connection factory resource reference of an application component's deployment descriptors.

Note: User IDs longer than 12 characters cannot be used for authentication with the Version 5 default messaging provider. For example, the default Windows NT user ID, **Administrator**, is not valid because it contains 13 characters. Therefore, an authentication alias for a WebSphere topic connection factory must specify a user ID no longer than 12 characters.

Container-managed Authentication Alias:

This alias specifies a user ID and password to be used to authenticate connection to the messaging provider for container-managed authentication.

The specification of a login configuration and associated properties on the component resource reference determines the container-managed authentication strategy when the res-auth value is Container. If the 'DefaultPrincipalMapping' login configuration is used, the associated property is a container-managed authentication alias. This field is used only in the absence of a loginConfiguration on the component resource reference. To define a new alias, see the related item J2EE Connector Architecture (J2C) authentication data entries.

This property provides a list of the J2C authentication data entry aliases that have been defined to WebSphere Application Server. You can select a data entry alias to be used to authenticate the creation of a new connection to the JMS provider.

If you have enabled administrative security for WebSphere Application Server, select the alias that specifies the user ID and password used to authenticate the creation of a new connection to the JMS provider. The use of this alias depends on the resource authentication (res-auth) setting declared in the connection factory resource reference of an application component's deployment descriptors.

Note: User IDs longer than 12 characters cannot be used for authentication with the embedded WebSphere JMS provider. For example, the default Windows NT user ID, **Administrator**, is not valid for use with embedded WebSphere messaging, because it contains 13 characters. Therefore, an authentication alias for a WebSphere JMS provider connection factory must specify a user ID no longer than 12 characters.

Mapping-Configuration Alias:

The module used to map authentication aliases.

The specification of a login configuration and associated properties on the component resource reference determines the container-managed authentication strategy when the res-auth value is Container. This field is used only in the absence of a loginConfiguration on the component resource reference.

This field provides a list of the modules that have been configured on the **Security** → **JAAS Configuration** → **Application Logins Configuration** property. For more information about the mapping configurations, see Java Authentication and Authorization service configuration entry settings.

Data type	Enum
Default	Null

Range

ClientContainer
The client container maps authentication aliases.
WSLogin
The WSLogin module maps authentication aliases.
DefaultPrincipalMapping
The JAAS configuration maps an authentication alias to its userid and password.

Clone Support:

Select this checkbox to enable clone support to allow the same durable subscription across topic clones.

Data type Enum
Default Cleared
Range

Selected
Clone support is enabled.
Cleared
Clone support is disabled.

If you select this property, you must also specify a value for the **Client ID** property.

Client ID:

The JMS client identifier used for connections to the queue manager.

Data type String
Range A valid JMS client ID

XA Enabled:

Specifies whether the connection factory is for XA or non-XA coordination of messages and controls if the application server uses XA QCF/TCF. Enable XA if multiple resources are used in the same transaction.

If you clear this checkbox property (for non-XA coordination), the JMS session is still enlisted in a transaction, but uses the resource manager local transaction calls (session.commit and session.rollback) instead of XA calls. This can lead to an improvement in performance. However, this means that only a single resource can be enlisted in a transaction in WebSphere Application Server.

Last participant support enables you to enlist one non-XA resource with other XA-capable resources.

For a WebSphere Topic Connection Factory with the **Port** property set to `DIRECT` this property does not apply, and always adopts non-XA coordination.

Data type Checkbox
Default Selected (enabled for XA coordination)
Range

Selected
The connection factory is enabled for XA-coordination of messages
Cleared
The connection factory is not enabled for XA coordination of messages

Recommended	Do not enable XA coordination when the message queue or topic received is the only resource in the transaction. Enable XA coordination when other resources, including other queues or topics, are involved.

Connection pool:

Specifies an optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the messaging provider to improve performance. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Session pool:

An optional set of session pool settings.

This link provides a panel of optional connection pool properties, common to all J2C connectors.

The application server pools connections and sessions with the JMS provider to improve performance. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.

Version 5 WebSphere queue destination settings:

Use this panel to view or change the configuration properties of the selected JMS queue destination for point-to-point messaging by WebSphere Application Server Version 5 applications.

A queue destination is used to configure a JMS queue of the default messaging provider for use by WebSphere Application Server Version 5 applications. Connections to the queue are created by the associated V5 Default Messaging WebSphere queue connection factory.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. **(Optional)** In the content pane, change the **Scope** setting to the level at which JMS resources are visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource. However, the JMS resource has only the subset of properties that apply to WebSphere Application Server Version 5. If you want to define a JMS resource at Cell level for use on non-Version 5 nodes, you should define the JMS resource for the Version 6 default messaging provider.
3. In the content pane, click the name of the V5 default messaging provider that you want to support the JMS destination.
4. Under Additional Resources, click **Queues**. This displays a list of any existing JMS queue destinations.
5. Click the name of the JMS queue destination that you want to work with.

A JMS queue for use with the internal WebSphere JMS provider has the following properties.

Scope:

Specifies the level to which this resource definition is visible to applications.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

The scope displayed is for information only, and cannot be changed on this panel. If you want to browse or change this resource (or other resources) at a different scope, change the scope on the messaging provider settings panel, then click **Apply**, before clicking the link for the type of resource.

Data type String

Name:

The name by which the queue is known for administrative purposes within IBM WebSphere Application Server.

To enable applications to use this queue, you must add the queue name to the Queue Names field on the panel for the JMS server that hosts the queue.

Data type String

JNDI name:

The JNDI name that is used to bind the queue into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Description:

A description of the queue, for administrative purposes

Data type String
Default Null

Category:

A category used to classify or group this queue, for your IBM WebSphere Application Server administrative records.

Data type String

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type Enum

Default APPLICATION DEFINED

Range

APPLICATION DEFINED
Messages on the destination have their persistence defined by the application that put them onto the queue.

NON PERSISTENT
Messages on the destination are not persistent.

PERSISTENT
Messages on the destination are persistent. When a persistent message is put to a queue, all of the message data is written to the messaging log (under the *embedded_messaging_install*\log directory) to make recovery of the message possible.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type Enum

Default APPLICATION DEFINED

Range

APPLICATION DEFINED
The priority of messages on this destination is defined by the application that put them onto the destination.

QUEUE DEFINED
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.

SPECIFIED
The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type Integer

Units Message priority level

Default 0

Range 0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type Enum

Default APPLICATION DEFINED

Range

APPLICATION DEFINED
The expiry timeout for messages on this queue is defined by the application that put them onto the queue.

UNLIMITED
Messages on this queue have no expiry timeout, so those messages never expire.

SPECIFIED
The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the **Specified expiry** property.*

Specified expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type Integer
Units Milliseconds
Default 0
Range Greater than or equal to 0
- 0 indicates that messages never timeout
- Other values are an integer number of milliseconds

Version 5 WebSphere topic destination settings:

Use this panel to browse or change the configuration properties of the selected JMS topic destination for publish/subscribe messaging by WebSphere application server Version 5 applications.

A WebSphere topic destination is used to configure the properties of a JMS topic for the default messaging provider on a Version 5 node in a deployment manager cell. Connections to the topic are created by the associated topic connection factory.

To view this page, use the administrative console to complete the following steps:
1. In the navigation pane, click **Resources** → **JMS** → **JMS providers**.
2. **(Optional)** In the content pane, change the **Scope** setting to the level at which JMS resources are visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource. However, the JMS resource has only the subset of properties that apply to WebSphere Application Server Version 5. If you want to define a JMS resource at Cell level for use on non-Version 5 nodes, you should define the JMS resource for the Version 6 default messaging provider.
3. In the content pane, click the name of the V5 default messaging provider that you want to support the JMS destination.
4. Under Additional Resources, click **Topics**. This displays a list of any existing JMS topic destinations.
5. Click the name of the JMS topic destination that you want to work with.

A JMS topic destination for use with the Version 5 default messaging provider has the following properties.

Scope:

Specifies the level to which this resource definition is visible to applications.

Resources such as messaging providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

The scope displayed is for information only, and cannot be changed on this panel. If you want to browse or change this resource (or other resources) at a different scope, change the scope on the messaging provider settings panel, then click **Apply**, before clicking the link for the type of resource.

Data type String

Name:

The name by which the topic is known for administrative purposes within IBM WebSphere Application Server.

Data type String

JNDI name:

The JNDI name that is used to bind the topic into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type String

Description:

A description of the topic, for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

Category:

A category used to classify or group this topic, for your IBM WebSphere Application Server administrative records.

Data type String

Topic:

The name of the topic as defined to the JMS provider.

Data type String
Default Null
Range The topic value can be dot notation and include wildcard characters.

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type Enum
Default APPLICATION DEFINED
Range
APPLICATION DEFINED
Messages on the destination have their persistence defined by the application that put them onto the queue.
NON-PERSISTENT
Messages on the destination are not persistent.
PERSISTENT
Messages on the destination are persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type Enum
Units Not applicable
Default APPLICATION DEFINED
Range
APPLICATION DEFINED
The priority of messages on this destination is defined by the application that put them onto the destination.
QUEUE DEFINED
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.
SPECIFIED
The priority of messages on this destination is defined by the **Specified priority** property. *If you select this option, you must define a priority on the **Specified priority** property.*

Specified priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type Integer
Units Message priority level
Default 0
Range 0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type Enum
Units Not applicable
Default APPLICATION DEFINED

Range

APPLICATION DEFINED
The expiry timeout for messages on this queue is defined by the application that put them onto the queue.

UNLIMITED
Messages on this queue have no expiry timeout, so those messages never expire.

SPECIFIED
The expiry timeout for messages on this queue is defined by the **Specified expiry** property. *If you select this option, you must define a timeout on the **Specified expiry** property.*

Specified expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds
Default	0
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Configuring Version 5 default JMS resources

Use the following tasks to configure the JMS connection factories and destinations WebSphere Application Server Version 5 applications.

About this task

You only need to complete these tasks if you have WebSphere application server Version 5 applications that need to use JMS resources provided by the default messaging provider. Such JMS resources are maintained as *V5 Default Messaging* resources.

In a deployment manager cell, you can use *V5 Default Messaging* resources to maintain Version 5 default messaging resources for both Version 6 and Version 5 nodes in the cell.

- Configuring a connection for Version 5 default messaging
- Configuring a Version 5 default JMS queue connection factory
- Configuring a Version 5 default JMS topic connection factory
- Configuring a Version 5 default JMS queue destination
- Configuring a Version 5 default JMS topic destination

Configuring a connection for Version 5 default messaging:

Use this task to configure a connection to Version 5 default messaging. This task is provided to help you manually migrate nodes from v5 to v6. If you use the supplied tools to migrate existing v5 nodes to v6, the jmsserver, appserver and port that you create with this task are defined automatically.

About this task

To configure a connection for Version 5 default messaging, use the administrative console to complete the following steps:

1. Create an application server with the name jmsserver (only one of these can exist on each node). In the navigation pane, expand **Servers** → **Application servers**.
2. On the new server, define a new JMSSERVER_QUEUED_ADDRESS port with the host set to the v6 server host, and the port set to the same as the SIB_MQ_ENDPOINT_ADDRESS of the server which is a member of your bus. The appserver called jmsserver does not actually have to be started; it is only used for looking up the port address.
3. Define a queue connection factory and queue at the Node or Cell scope. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
4. In the content pane, click the name of the V5 default messaging provider.
5. Define a queue destination on your bus with the same name as your queue defined under v5 default messaging.
6. Define an alias destination on your bus with the same name as your queue defined under v5 default messaging but with WQ_ appended to the front the name. For example, if your queue has the name MyV5Queue, your alias should have the name WQ_MyV5Queue.
7. Point the alias at your queue destination with the correct name. The migration process targets the queue with the WQ_ prefix; defining the alias to point to the real queue helps migration.
8. Define the WebSphere MQ Client Link on your messaging engine on the bus. Keep the WebSphere MQ channel name WAS.JMS.SVRCONN and set the queue manager name so that it contains your node name. For example, if your node name is MyNode, you would set the queue manager name to WAS_<MyNode>_jmsserver. Now set the queue manager name to the same; in this example it would be WAS_MyNode_jmsserver. The WebSphere MQ Client Link will show a status of inactive, this is normal.
9. Restart your server so that the new JNDI definitions bind correctly. Your v5 client should now be able to connect to v6 using the host and bootstrap address of the v6 system in the provider url component of the initial context. Your client should now also be able to send messages to the destination on the bus.
10. If you open the Client connections view and click the refresh icon, the hostname of the connecting system is visible whilst the client is connected. In the navigation pane, expand**Buses** → **[bus name]** → **Messaging engines** → **[messaging engine name]** → **WebSphere MQ client links** → **[client link name]** → **Runtime** → **Client connections view**.
11. Click **OK**.
12. Save any changes to the master configuration.
13. To have the changed configuration take effect, stop then restart the application server.

Configuring a Version 5 queue connection factory:

Use this task to browse or change the properties of a JMS queue connection factory for point-to-point messaging with the default messaging provider on a Version 5 node in a deployment manager cell. This task contains an optional step for you to create a new JMS queue connection factory.

About this task

To configure a JMS queue connection factory for use by WebSphere application server Version 5 applications, use the administrative console to complete the following steps:

1. Display the Version 5 default messaging provider. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
2. Select the Version 5 provider for which you want to configure a queue connection factory.
3. Optional: Change the **Scope** setting to the level at which the JMS queue connection factory is visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource.
4. In the content pane, under Additional Properties, click **Queue connection factories** This displays any existing JMS queue connection factories for the Version 5 messaging provider in the content pane.

5. To browse or change an existing JMS queue connection factory, click its name in the list. Otherwise, to create a new connection factory, complete the following steps:
   a. Click **New** in the content pane.
   b. Specify the following required properties. You can specify other properties, as described in a later step.

      Name The name by which this JMS queue connection factory is known for administrative purposes within IBM WebSphere Application Server.

      JNDI Name
      The JNDI name that is used to bind the JMS queue connection factory into the name space.
   c. Click **Apply**. This defines the JMS queue connection factory to WebSphere Application Server, and enables you to browse or change additional properties.
6. Optional: Change properties for the queue connection factory, according to your needs.
7. Click **OK**.
8. Save any changes to the master configuration.
9. To have the changed configuration take effect, stop then restart the application server.

Configuring a Version 5 JMS topic connection factory:

Use this task to browse or change a JMS topic connection factory for publish/subscribe messaging by WebSphere Application Server Version 5 applications.

About this task

To configure a JMS topic connection factory for use by WebSphere application server Version 5 applications, use the administrative console to complete the following steps:

1. Display the Version 5 default messaging provider. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
2. Select the Version 5 provider for which you want to configure a topic connection factory.
3. Optional: Change the **Scope** setting to the level at which the JMS topic connection factory is visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource.
4. In the content pane, under Additional Properties, click **Topic connection factories** This displays any existing JMS topic connection factories for the Version 5 messaging provider in the content pane.
5. To browse or change an existing JMS topic connection factory, click its name in the list. Otherwise, to create a new connection factory, complete the following steps:
   a. Click **New** in the content pane.
   b. Specify the following required properties. You can specify other properties, as described in a later step.

      Name The name by which this JMS topic connection factory is known for administrative purposes within IBM WebSphere Application Server.

      JNDI Name
      The JNDI name that is used to bind the JMS topic connection factory into the name space.
   c. Click **Apply**. This defines the JMS topic connection factory to WebSphere Application Server, and enables you to browse or change additional properties.
6. Optional: Change properties for the topic connection factory, according to your needs.
7. Click **OK**.
8. Save any changes to the master configuration.
9. To have the changed configuration take effect, stop then restart the application server.

Configuring a Version 5 WebSphere queue destination:

Use this task to browse or change the properties of a JMS queue destination for point-to-point messaging by WebSphere Application Server Version 5 applications. This task contains an optional step for you to create a new topic destination.

About this task

To optimize performance, configure the topic destination properties to best fit your applications. For more information, see “Performance considerations for WebSphere Application Server Version 5 queue destinations” on page 1212.

To configure a JMS queue destination for use by WebSphere Application Server Version 5 applications, use the administrative console to complete the following steps:

1. Display the Version 5 default messaging provider. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
2. Optional: Change the **Scope** setting to the level at which the JMS destination is visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource.
3. Select the Version 5 provider for which you want to configure a queue destination.
4. In the content pane, under Additional Properties, click **Queues** This displays any existing queue destinations for the Version 5 default messaging provider in the content pane.
5. To browse or change an existing JMS queue destination, click its name in the list. Otherwise, to create a new queue destination, complete the following steps:
   a. Click **New** in the content pane.
   b. Specify the following required properties. You can specify other properties, as described in a later step.

      Name The name by which this queue destination is known for administrative purposes within IBM WebSphere Application Server.

      JNDI Name
      The JNDI name that is used to bind the queue destination into the name space.
   c. Click **Apply**. This defines the queue destination to WebSphere Application Server, and enables you to browse or change additional properties.
6. Optional: Change properties for the queue destination, according to your needs.
7. Click **OK**.
8. Save any changes to the master configuration.
9. To make a queue destination available to applications, host the queue on a JMS server. To add a new queue to a JMS server or to change an existing queue on a JMS server, you define the administrative name of the queue to the JMS server.

   To define the administrative name of the queue to the JMS server, see Managing Version 5 JMS servers in a deployment manager cell.
10. To have the changed configuration take effect, stop then restart the application server.

Configuring a Version 5 WebSphere topic destination:

Use this task to browse or change the properties of a JMS topic destination for publish/subscribe messaging by WebSphere application server Version 5 applications.. This task contains an optional step for you to create a new topic destination.

About this task

To optimize performance, configure the topic destination properties to best fit your applications. For more information, see “Performance considerations for WebSphere Application Server Version 5 topic destinations” on page 1213.

To configure a JMS topic destination for use WebSphere Application Server Version 5 applications, use the administrative console to complete the following steps:

1. Display the Version 5 default messaging provider. In the navigation pane, expand **Resources** → **JMS** → **JMS providers**.
2. Select the Version 5 provider for which you want to configure a topic destination.
3. Optional: Change the **Scope** setting to the level at which the JMS destination is visible to applications. If you define a Version 5 JMS resource at the Cell scope level, all users in the cell can look up and use that JMS resource.
4. In the content pane, under Additional Properties, click **Topics** This displays any existing JMS topic destinations for the Version 5 default messaging provider in the content pane.
5. To browse or change an existing JMS topic destination, click its name in the list. Otherwise, to create a new topic destination, complete the following steps:
   a. Click **New** in the content pane.
   b. Specify the following required properties. You can specify other properties, as described in a later step.

      Name The name by which this topic destination is known for administrative purposes within IBM WebSphere Application Server.

      JNDI Name
      The JNDI name that is used to bind the topic destination into the name space.

      Topic The name of the topic in the default messaging provider, to which messages are sent.
   c. Click **Apply**. This defines the topic destination to WebSphere Application Server, and enables you to browse or change additional properties.
6. Optional: Change properties for the topic destination, according to your needs.
7. Click **OK**.
8. Save any changes to the master configuration.
9. To have the changed configuration take effect, stop then restart the application server.

Managing Version 5 JMS servers in a deployment manager cell

Use this task to manage JMS servers on WebSphere Application Server Version 5 nodes in a deployment manager cell.

About this task

The use of JMS servers is only intended to support migration from WebSphere Application Server Version 5 nodes to WebSphere Application Server Version 6.

In a WebSphere Application Server deployment manager cell, each Version 5 node can have at most one JMS server, and any Version 5 application server within the cell can use JMS resources served by any of those JMS servers. Applications on WebSphere Application Server Version 6 can also use JMS resources served by any of those JMS servers.

You can use the WebSphere administrative console to display a list of all Version 5 JMS servers, to show and control their runtime status. You can also configure a general set of JMS server properties, which add to the default values of properties configured automatically for the Version 5 default messaging provider.

Note: In general, the default values of properties for the Version 5 default messaging provider are adequate for JMS servers. However, if you are running high messaging loads, you may need to change some WebSphere MQ properties; for example, WebSphere MQ properties for log file locations, file pages, and buffer pages. For more information about configuring WebSphere MQ properties, see the *WebSphere MQ System Administration* book, SC33-1873, which is available from the IBM Publications Center or from the WebSphere MQ collection kit, SK2T-0730.

To manage a Version 5 JMS server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers** → **JMS Servers** This displays a table of the JMS servers, showing their runtime status.
2. Optional: If you want to change the runtime status of a JMS server, complete the following steps:
   a. In the table of JMS servers, select the JMS servers that you want to act on.
      - To act on one or more specific JMS servers, select the check box next to the JMS server name.
      - To act on all JMS servers, select the check box next to the JMS servers title of the table.
   b. Click one of the actions displayed to change the status of the JMS servers; for example, click **Stop** to stop a JMS server.

   The status of the JMS servers that you have acted on is updated to show the result of your actions.
3. Optional: If you want to change the properties of a JMS server, complete the following steps:
   a. Click the name of the server
   b. Set one or more of the following configuration properties:

      Initial state
      If you want the JMS server to be started automatically when the application server is next started, set this property to `Started`.

      Number of threads
      Set the number of concurrent threads to be used by the publish/subscribe matching engine. The number of concurrent threads should only be set to a small number.
4. Optional: If you want the JMS server to host a new JMS queue, add the queue name to the Queue Names field.

   The name must match the name of a JMS Queue administrative object, including the use of upper- and lowercase.
5. Optional: If you want to stop the JMS server hosting a JMS queue, remove the queue name from the Queue Names field.
6. Click **OK**.
7. Save your changes to the master configuration.
8. To have the changed configuration take effect, stop then restart the JMS server.

Configuring authorization security for a Version 5 default messaging provider

Use this task to configure authorization security for the default messaging provider on a WebSphere Application Server Version 5 node in a deployment manager cell.

About this task

To configure authorization security for the Version 5 default messaging provider complete the following steps.

Note: Security for the Version 5 default messaging provider is enabled when you enable WebSphere Application Server security on the Version 5 node. For more information about enabling security, see Enabling security.

1. Configure authorization settings to access JMS resources owned by the embedded messaging subsystem.

Authorization to access JMS resources owned by the embedded messaging subsystem is controlled by settings in the *wempspath*/*wempsname*/config/integral-jms-authorizations.xml file. The settings grant or deny authenticated userids access to internal JMS provider resources (queues or topics). As supplied, the integral-jms-authorisations.xml file grants the following permissions:

- Read and write permissions to all queues.
- Pub, sub, and persist to all topics.

To configure authorization settings, edit the integral-jms-authorisations.xml file according to the information in this topic and in that file.

2. Edit the queue-admin-userids element to create a list of userids with administrative access to all queues. Administrative access is needed to create queues and perform other administrative activities on queues. For example, consider the following queue-admin-userids section:

```
<queue-admin-userids>
    <userid>adminid1</userid>
    <userid>adminid2</userid>
</queue-admin-userids>
```

In this example the userids adminid1 and adminid2 are defined to have administrative access to all queues.

3. Edit the queue-default-permissions element to define the default queue access permissions. These permissions are used for queues for which you do not define specific permissions (in queue sections). If this section is not specified, then access permissions exist only for those queues for which you have specifically created queue elements.

   For example, consider the following queue-default-permissions element:

```
<queue-default-permissions>
  <permission>write</permission>
</queue-default-permissions>
```

In this example the default access permission for all queues is **write**. This can be overridden for a specific queue by creating a queue element that sets its access permission to **read**.

4. If you want to define specific access permissions for a queue, create a queue element, then define the following elements:

   For example, consider the following queue element:

```
<queue>
  <name>q1</name>
  <public>
  </public>
  <authorize>
    <userid>useridr</userid>
    <permission>read</permission>
  </authorize>
  <authorize>
    <userid>useridw</userid>
    <permission>write</permission>
  </authorize>
  <authorize>
    <userid>useridrw</userid>
    <permission>read</permission>
    <permission>write</permission>
  </authorize>
</queue>
```

In this example for the queue q1, the userid useridr has read permission, the userid useridw has write permission, the userid useridrw has both read and write permissions, and all other userids have no access permissions (<public></public>).

5. Edit topic elements to define the access permissions for publish/subscribe topic destinations.

   For topics, you can grant and deny access permissions. Full permission inheritance is supported on topics. If you do not define specific access permissions for a userid on a specific topic then

permissions are inherited first from the public permissions on that topic then from the parent topic. The inheritance of access permissions continues until the root topic from which the root permissions are assumed.

a. If you want to define default access permissions for the root topic, edit a topic element with an empty name element. If you omit such a topic section, topics have no default topic permissions other than those defined by specific topic elements. For example, consider the following topic element for the root topic:

```
<topic>
   <name></name>
   <public>
     <permission>+pub</permission>
   </public>
 </topic>
```

In this example, the default access permission for all topics is set to publish. This can be overridden by other topic elements for specific topic names.

b. If you want to define access permissions for a specific topic, create a topic element with the name for the topic then define the access permissions in the public and authorize elements of the topic element. For example, consider the following topic section:

```
<topic>
  <name>a/b/c</name>
  <public>
    <permission>+sub</permission>
  </public>
  <authorize>
    <userid>useridpub</userid>
    <permission>+pub</permission>
  </authorize>
</topic>
```

In this example, the subscribe permission is granted to anyone accessing any topic whose name starts with `a/b/c`. Also, the userid useridpub is granted publish permission for any topic whose name starts with `a/b/c`.

6. Save the integral-jms-authorizations.xml file.

Results

If the dynamic update setting is selected, changes to the integral-jms-authorizations.xml file become active when the changed file is saved, so there is no need to stop and restarted the JMS server. If the dynamic update setting is not selected, you need to stop and restart the JMS server to make changes active.

Dynamic updating is available, by ensuring proper tagging in the integral-jms-authorizations.xml file `<dyanmic-update>true</dynamic-update>`.

Authorization settings for Version 5 default JMS resources:

Use the integral-jms-authorisations.xml file to view or change the authorization settings for Java Message Service (JMS) resources owned by the default messaging provider on WebSphere Application Server Version 5 nodes.

Authorization to access default JMS resources owned by the default messaging provider on WebSphere Application Server nodes is controlled by the following settings in the *wempspath*/*wempsname*/`config/integral-jms-authorizations.xml` file.

This structure of the settings in integral-jms-authorisations.xml is shown in the following example. Descriptions of these settings are provided after the example. To configure authorization settings, follow the instructions provided in Configuring authorization security for the Version 5 JMS providers

```
<integral-jms-authorizations>

  <dynamic-update>true</dynamic-update>

  <queue-admin-userids>
    <userid>adminid1</userid>
    <userid>adminid2</userid>
  </queue-admin-userids>

  <queue-default-permissions>
    <permission>write</permission>
  </queue-default-permissions>

  <queue>
    <name>q1</name>
    <public>
    </public>
    <authorize>
      <userid>useridr</userid>
      <permission>read</permission>
    </authorize>
    <authorize>
      <userid>useridw</userid>
      <permission>write</permission>
    </authorize>
  </queue>

  <queue>
    <name>q2</name>
    <public>
      <permission>write</permission>
    </public>
    <authorize>
      <userid>useridr</userid>
      <permission>read</permission>
    </authorize>
  </queue>

  <topic>
    <name></name>
    <public>
      <permission>+pub</permission>
    </public>
  </topic>

  <topic>
    <name>a/b/c</name>
    <public>
      <permission>+sub</permission>
    </public>
    <authorize>
      <userid>useridpub</userid>
      <permission>+pub</permission>
    </authorize>
  </topic>

</integral-jms-authorizations>
```

dynamic-update: Controls whether or not the JMS Server checks dynamically for updates to this file.
true (Default) Enables dynamic update support.
false Disables dynamic update checking and improves authorization performance.

queue-admin-userids: This element lists those userids with administrative access to all Version 5 default queue destinations. Administrative access is needed to create queues and perform other administrative activities on queues. You define each userid within a separate userid sub element:

<userid>*adminid***</userid>**
Where *adminid* is a user ID that can be authenticated by IBM WebSphere Application Server.

queue-default-permissions: This element defines the default queue access permissions that are assumed if no permissions are specified for a specific queue name. These permissions are used for queues for which you do not define specific permissions (in queue elements). If this element is not specified, then no access permissions exist unless explicitly authorized for individual queues.

You define the default permission within a separate permission sub element:
<permission>*read-write***</permission>**
Where *read-write* is one of the following keywords:
read By default, userids have read access to Version 5 default queue destinations.
write By default, userids have write access to Version 5 default queue destinations.

queue: This element contains the following authorization settings for a single queue destination:
name The name of the queue.
public The default public access permissions for the queue. This is used only for those userids that have no specific authorize element. If you leave this element empty, or do not define it at all, only those userids with authorize elements can access the queue.

You define each default permission within a separate permission element.
authorize
The access permissions for a specific userid. Within each authorize element, you define the following elements:
userid The userid that you want to assign a specific access permission.
permission
An access permission for the associated userid.

You define each permission within a separate permission element. Each permission element can contain the keyword read or write to define the access permission.

For example, consider the following queue element:

```
<queue>
  <name>q1</name>
  <public>
  </public>
  <authorize>
    <userid>useridr</userid>
    <permission>read</permission>
  </authorize>
  <authorize>
    <userid>useridw</userid>
    <permission>write</permission>
  </authorize>
  <authorize>
    <userid>useridrw</userid>
    <permission>read</permission>
    <permission>write</permission>
  </authorize>
</queue>
```

topic: This element contains the following authorization settings for a single topic destination:

Each topic element has the following sub elements:
name The name of the topic, without wildcards or other substitution characters.
public The default public access permissions for the topic. This is used only for those userids that have no specific authorize element. If you leave this element empty, or do not define it at all, only those userids with authorize elements can access the topic.

You define each default permission within a separate permission element.

authorize
The access permissions for a specific userid. Within each authorize element, you define the following elements:
userid The userid that you want to assign a specific access permission.
permission
An access permission for the associated userid.

You define each permission within a separate permission element. Each permission element can contain one of the following keywords to define the access permission:
+pub Grant publish permission
+sub Grant subscribe permission
+persist
Grant persist permission
-pub Deny publish permission
-sub Deny subscribe permission
-persist
Deny persist permission

Tuning JMS destinations

Use this task to configure the properties of a JMS destination to optimize performance of applications that use the WebSphere Application Version 5 default messaging provider or WebSphere MQ as a JMS provider.

About this task

To optimize performance, configure destination properties to best fit your applications. You should also consider queue attributes of the JMS server that are associated with the queue name. For more information, see the following topics:

- "Performance considerations for WebSphere Application Server Version 5 queue destinations"
- "Performance considerations for WebSphere Application Server Version 5 topic destinations" on page 1213
- "Performance considerations for WebSphere MQ queue destinations" on page 1213
- "Performance considerations for WebSphere MQ topic destinations" on page 1214

Performance considerations for WebSphere Application Server Version 5 queue destinations:

To optimize performance, configure the queue destination properties to best fit your applications. For example, setting the Expiry property to SPECIFIED and the Specified Expiry property to 30000 milliseconds for the expiry timeout, reduces the number of messages that can be queued.

To ensure that there are enough underlying WebSphere MQ resources available for the queue, you must ensure that you configure the queue destination properties adequately for your application usage.

For queue destinations configured on a WebSphere Application Server Version 5 node, you should also consider queue attributes of the internal JMS server that are associated with the queue name. Inappropriate queue attributes can reduce the performance of WebSphere operations.
BOQNAME
The excessive backout requeue name. This can be set to a local queue name that can hold the messages which were rolled back by the WebSphere applications. This queue name can be a system dead letter queue.
BOTHRESH
The backout threshold and can be set to a number once the threshold is reached, the message will be moved to the queue name specified in BOQNAME.

For more information about using these properties, see:

- “Handling poison messages” in the *WebSphere MQ Using Java* book
- The *WebSphere MQ Script (MQSC) Command Reference* book

Performance considerations for WebSphere Application Server Version 5 topic destinations:

To optimize performance, configure the JMS destination properties to best fit your applications.

For example, setting the Expiry property to SPECIFIED and the Specified Expiry property to 30000 milliseconds for the expiry timeout, reduces the number of messages that can be queued.

For JMS destinations configured on a WebSphere Application Server Version 5 node, ensure that there are enough underlying WebSphere MQ resources available for the queue, you must ensure that you configure the queue destination properties adequately for your application usage.

- Ensure the queue attribute, INDXTYPE is set to MSGID for the following system queues:
  - SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE
  - SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE
- Ensure the queue attribute, INDXTYPE is set to CORRELID for the following system queues:
  - SYSTEM.JMS.ND.SUBSCRIBER.QUEUE
  - SYSTEM.JMS.D.SUBSCRIBER.QUEUE

For more information about using these properties, see:

- The *WebSphere MQ Using Java* book
- The *WebSphere MQ Script (MQSC) Command Reference* book

Performance considerations for WebSphere MQ queue destinations:

To optimize performance, configure the queue destination properties to best fit your message-driven beans or other applications that use the queue destinations.

For example:

- When MDB applications are configured to WebSphere MQ queues on z/OS, the INDEX by MSGID is very important.
- Setting the Expiry property to SPECIFIED and the Specified Expiry property to 30000 milliseconds for the expiry timeout, reduces the number of messages that can be queued.

To ensure that there are enough underlying WebSphere MQ resources available for the queue, you must ensure that you configure the queue destination properties adequately for use by your message-driven beans or other applications that use the queue.

You should also consider queue attributes of the internal JMS server that are associated with the queue name. Inappropriate queue attributes can reduce the performance of WebSphere operations.

You should also consider the queue attributes associated with the queue name you created with WebSphere MQ. Inappropriate queue attributes can reduce the performance of WebSphere operations. You can use WebSphere MQ commands to change queue attributes for the queue name.

BOQNAME
: The excessive backout requeue name. This can be set to a local queue name that can hold the messages which were rolled back by the WebSphere applications. This queue name can be a system dead letter queue.

BOTHRESH
: The backout threshold and can be set to a number once the threshold is reached, the message will be moved to the queue name specified in BOQNAME.

INDXTYPE
Set this to MSGID. This causes an index of message identifiers to be maintained, which can improve WebSphere MQ retrieval of messages.

DEFSOPT
Set this to SHARED (for shared input from the queue).

SHARE
This must be specified (so that multiple applications can get messages from this queue).

For more information about using these properties, see:

- For BOQNAME and BOTHRESH, see “Handling poison messages” in the *WebSphere MQ Using Java* book
- The *WebSphere MQ Script (MQSC) Command Reference* book

Performance considerations for WebSphere MQ topic destinations:

To optimize performance, configure the topic destination properties to best fit your applications. For example, setting the Expiry property to SPECIFIED and the Specified Expiry property to 30000 milliseconds for the expiry timeout, reduces the number of messages that can be queued.

To ensure that there are enough underlying WebSphere MQ resources available for the queue, you must ensure that you configure the queue destination properties adequately for your application usage.

- Ensure the queue attribute, INDXTYPE is set to MSGID for the following system queues:
  - SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE
  - SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE
- Ensure the queue attribute, INDXTYPE is set to CORRELID for the following system queues:
  - SYSTEM.JMS.ND.SUBSCRIBER.QUEUE
  - SYSTEM.JMS.D.SUBSCRIBER.QUEUE

For more information about using these properties, see:

- The *WebSphere MQ Using Java* book
- The *WebSphere MQ Script (MQSC) Command Reference* book

JMS components on Version 5 nodes

To provide messaging support on a WebSphere Application Server Version 5 node, there is at most one JMS server and some number of JMS resources configured for the default messaging JMS provider on that node.

A *JMS server* on a Version 5 node serves the JMS resources (connection factories and destinations) for that node. The JMS server is managed as a separate process to application servers on the same node. Any application server within the domain can access JMS resources served by any JMS server on any node in the domain.

A *connection factory* encapsulates the configuration properties used to create connections with the JMS provider, to enable applications to access JMS destinations.

The main components of JMS support on a Version 5 node are shown in the figure The main components of WebSphere JMS support.

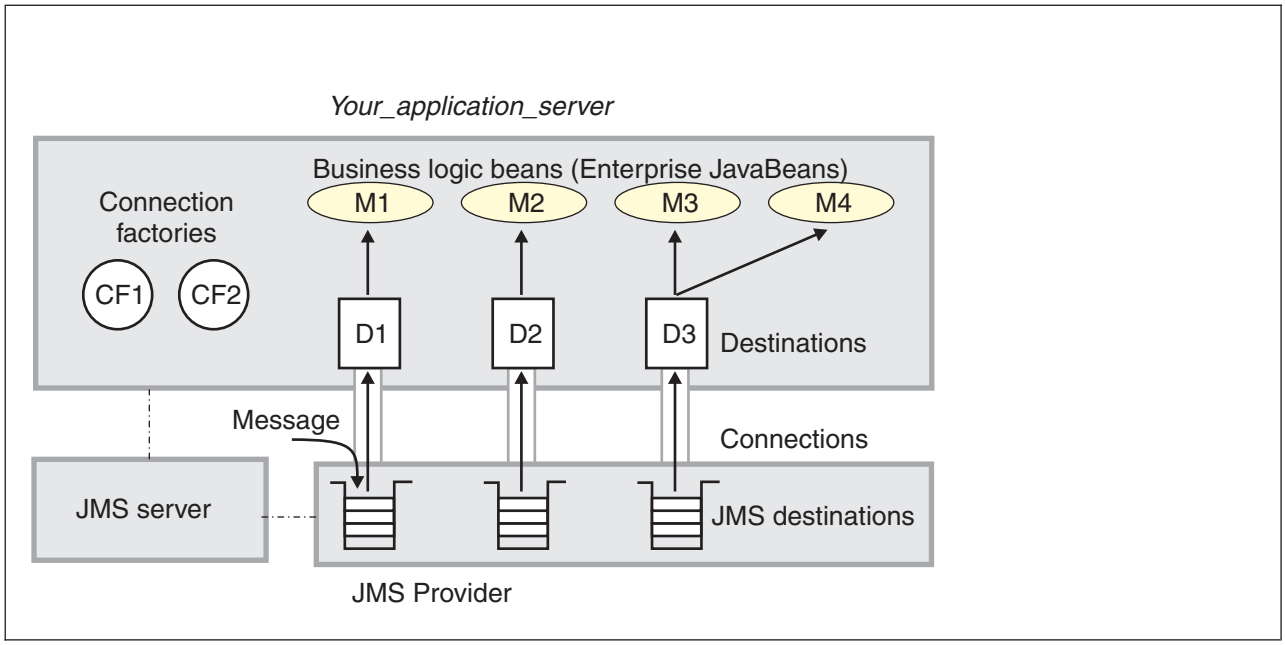


Figure 7. The main JMS components on a version 5 node. This figure shows the main JMS components on a version 5 node, from JMS provider through a connection to a destination, then to a WebSphere enterprise application (acting as a JMS client) that processes the message retrieved from the destination. For more information, see the text that accompanies this figure.

Administering listener ports and activation specifications for message-driven beans

Use these tasks to manage the listener ports and activation specifications used by message-driven beans (MDBs). If the JMS provider is implemented as a J2EE Connector Architecture (JCA) resource adapter use an activation specification, otherwise use a listener port. These tasks are supplementary to the tasks of administering resource adapters, and JMS provider resources.

About this task

Note: From WebSphere Application Server Version 7 listener ports are deprecated. For information about the facilities available to aid migration of configuration information from a listener port to an activation specification for use with the Websphere MQ messaging provider, refer to related tasks.

Use the WebSphere Application Server administrative console to configure the following resources for message-driven beans:

- J2C activation specifications for JCA 1.5-compliant message-driven beans. Activation specifications must be provided when the application's resources are configured using the default messaging provider or any generic J2C Resource Adapter that supports inbound messaging.
- The message listener service, listener ports, and listeners for EJB 2.0 message-driven beans deployed against listener ports. Listener ports must be provided when using the following JMS providers: V5 Default Messaging, or Generic.

Listener port or activation specification?

WebSphere Application Server Version 6 or later supports the enhancements made to the EJB 2.1 and Java EE specifications that allow any resource that has a JCA 1.5 adapter to trigger an MDB (formerly only JMS resources could trigger MDBs). WebSphere Application Server Version 5 and EJB 2.0 used the listener port which specified a JMS connection factory and a destination. This allowed a pool of MDB instances to connect to the messaging system and listen to the designated destination. EJB 2.1 specifies an additional requirement of an MDB working with a JCA adapter using an activation specification.

If you are developing WebSphere Application Server Version 6 or later applications, you should consider the following questions:

- Should I use a listener port or an activation specification?
- Should I convert my existing listener ports to activation specifications?
- Will listener ports eventually not be supported anymore?

Here are some guidelines to help you decide:

- If you are using J2EE 1.2 and EJB 1.1 with WebSphere Application Server v4, MDBs are not used, so you do not have to make a decision. WebSphere Application Server Version 4 uses message beans, but these are not MDBs or EJBs.
- If you are using J2EE 1.3 and EJB 2.0 with WebSphere Application Server Version 5, you must use listener ports. The MDBs are JMS MDBs that implement MessageListener, and there is no JCA support. WebSphere Application Server Version 5 uses listener ports to associate MDB classes with their JMS destinations.
- If you are using Java EE and EJB 2.1 with WebSphere Application Server Version 6 or later and not using JMS, you must use activation specifications. A connector MDB uses JCA to access its resources, so the connector must therefore be configured with an activation specification. This is for new bean development, and does not affect the conversion of MDBs from EJB 2.0 to EJB 2.1.
- If you are using Java EE and EJB 2.1 with WebSphere Application Server Version 6 or later, the decision depends on whether your JMS provider API is implemented with JCA. In Java EE, the JMS 1.1 API can now be implemented with the JCA 1.5 API. If so, your MDB is a JMS MDB that is implemented as a connector MDB, and must therefore be configured with an activation specification. If not, this is the same JMS situation as for J2EE 1.3, and you must configure this EJB 2.1 MDB in the same way as you would configure an EJB 2.0 MDB, which in WebSphere Application Server is to use a listener port.

To summarize: in WebSphere Application Server Version 6 or later, the decision on whether or not to use a listener port or an activation specification depends on the following scenarios:

- whether you upgrade your EJB 2.0 MDBs to EJB 2.1
- whether you want your EJB 2.1 MDB to use a JCA adapter
- whether you access your JMS provider via a JCA adapter.

If you have JMS API implementations that do not use JCA, WebSphere Application Server requires listener ports; if all JMS API implementations use JCA, listener ports are no longer required.

You can update the configuration data at any time, but some updates only take effect when the appropriate server is next started.

For additional information about administering support for message-driven beans, see the following topics:

- Configuring a JMS activation specification for MDBs used by the default messaging provider
- “Configuring a J2C activation specification”
- “Configuring a J2C administered object” on page 1221
- Configuring message listener resources for EJB 2.0 message-driven beans

Configuring a J2C activation specification

Use this task to configure a J2EE Connector (J2C) activation specification used to deploy message-driven beans with an external resource adapter.

About this task

Use this task if you want to use a message-driven bean as a listener on a Java Connector Architecture (JCA) 1.5 resource adapter other than the default messaging JMS provider.

You can create or modify a J2C activation specification under an installed resource adapter at the cell, node, or server scope. You can select the message listener type from those provided by the given resource adapter.

Configuring a J2C activation specification offers two distinct advantages:

- The activation specification configuration information can be shared among multiple message-driven beans across multiple applications.
- Updates to the configuration properties can be made without the need to redeploy the application.

The following guidelines show which scenarios use activation specifications or listener ports:

- If you are using J2EE 1.2 and Enterprise JavaBeans (EJB) 1.1 with WebSphere Application Server Version 4, message-driven beans are not used so you do not need listener ports or activation specifications. WebSphere Application Server Version 4 uses message beans, but these are not message-driven beans or enterprise beans.
- If you are using J2EE 1.3 and EJB 2.0 with WebSphere Application Server Version 5, you must use listener ports. The message-driven beans are JMS message-driven beans that implement MessageListener, and there is no JCA support. WebSphere Application Server Version 5 uses listener ports to associate message-driven bean classes with their JMS destinations.
- If you are using J2EE 1.4 and EJB 2.1 with WebSphere Application Server Version 6, you must use activation specifications. A connector message-driven bean uses JCA to access its resources, so the connector must therefore be configured with an activation specification. This is for new bean development, and does not affect the conversion of message-driven beans from EJB 2.0 to EJB 2.1.
- If you are using J2EE 1.4 and EJB 2.1 with WebSphere Application Server Version 6, the decision depends on whether your JMS provider API is implemented with JCA. In J2EE 1.4, the JMS 1.1 API can now be implemented with the JCA 1.5 API. If so, your message-driven bean is a JMS message-driven bean that is implemented as a connector message-driven bean, and must therefore be configured with an activation specification. If not, this is the same JMS situation as for J2EE 1.3, and you must configure this EJB 2.1 message-driven bean in the same way as you would configure an EJB 2.0 message-driven bean, which in WebSphere Application Server is to use a listener port.

To configure a J2C activation specification for an external resource adapter, use the administrative console to complete the following steps. This task contains an optional step for you to create a new activation specification.

1. Display the external resource adapter. In the navigation pane, click **Resources** → **Resource Adapters** → *adapter_name*. This displays in the content pane a table of properties for the external resource adapter, including links to the types of J2C resources that it provides.
2. Optional: Change the **Scope** setting to the scope level at which the activation specification is to be visible to applications, according to your needs.
3. In the content pane, under the Activation specifications heading, click **J2C Activation Specifications**. This lists any existing J2C activation specifications for the external resource adapter in the content pane.
4. Display the properties of the J2C activation specification. If you want to display an existing J2C activation specification, click one of the names listed.

   Alternatively, if you want to create a new J2C activation specification, click **New**, then specify the following required properties:

   Name Type the name by which the activation specification is known for administrative purposes. The JNDI name is automatically generated based on the value for the Name property.

   Message listener type
   Select the message listener type that this activation specification instance should support. This list is based on the deployment descriptor of the external resource adapter.

Depending on the external resource adapter, there can be additional required properties that need to be supplied. To provide values for these properties, click **Custom properties**. When creating a new activation specification, you may need to click **Apply** before this custom property selection is available.

5. Specify properties for the activation specification, according to your needs .
6. Click **OK**.
7. Save your changes to the master configuration.

J2C Activation Specifications collection:

This page contains a list of J2C activation specifications for a resource adapter configuration and is used to create new J2C activation specifications, to select J2C activation specifications for configuration changes, or to delete J2C activation specifications.

Activation specification definitions and classes are provided by a resource adapter when it is installed. Using this information, the administrator can create and configure J2C activation specifications with JNDI names that are then available for applications to use. The resource adapter uses a J2C activation specification to configure a specific endpoint instance. Each application configuring one or more endpoints must specify the resource adapter that sends messages to the endpoint. The application must use the activation specification to provide the configuration properties related to the processing of the inbound messages.

The following guidelines show which scenarios use activation specifications or listener ports:

- If you are using Java 2 Platform, Enterprise Edition (J2EE) 1.2 and EJB 1.1 with WebSphere Application Server v4, MDBs are not used so you do not need listener ports or activation specifications. WebSphere Application Server v4 uses message beans, but these are not MDbs or EJBs.
- If you are using J2EE 1.3 and EJB 2.0 with WebSphere Application Server v5, you must use listener ports. The MDBs are JMS MDBs that implement MessageListener, and there is no JCA support. WebSphere Application Server v5 uses listener ;ports to associate MDB classes with their JMS destinations.
- If you are using J2EE 1.4 and EJB 2.1 with WebSphere Application Server v6, you must use activation specifications. A connector MDB uses JCA to access its resources, so the connector must therefore be configured with an activation specification. This is for new bean development, and does not affect the conversion of MDBs from EJB 2.0 to EJB 2.1.
- If you are using J2EE 1.4 and EJB 2.1 with WebSphere Application Server v6, the decision depends on whether your JMS provider API is implemented with JCA. In J2EE 1.4, the JMS 1.1 API can now be implemented with the JCA 1.5 API. If so, your MDB is a JMS MDB that is implemented as a connector MDB, and must therefore be configured with an activation specification. If not, this is the same JMS situation as for J2EE 1.3, and you must configure this EJB 2.1 MDB in the same way as you would configure an EJB 2.0 MDB, which in WebSphere Application Server is to use a listener port.

You can access this administrative console page in one of two ways:

- **Resources** → **Resource Adapters** → **Resource adapters** → ***resource_adapter*** → **J2C activation specifications**.
- **Resources** → **Resource Adapters** → **J2C activation specifications**.

Name:

Specifies the display name of the J2C activation specification instance.

A string with no spaces meant to be a meaningful text identifier for the J2C activation specification.

Data type String

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name for the J2C activation specification instance.

Data type String

Scope:

Specifies the scope of the resource adapter that supports this activation specification. Only applications that are installed within this scope can use this activation specification.

Provider:

Specifies the resource adapter that encapsulates the appropriate classes for this activation specification.

Description:

A free-form text string to describe the J2C activation specification instance.

Data type String

Message Listener Type:

The Message Listener Type that is used by this activation specification.

The list of available classes is provided by the resource adapter.

Data type String

J2C Activation Specifications settings:

Use this page to specify the settings for a J2C activation specification.

The resource adapter uses a J2C activation specification to configure a specific endpoint instance. Each application configuring one or more endpoints must specify the resource adapter that sends messages to the endpoint. The application must use the activation specification to provide the configuration properties related to the processing of the inbound messages.

You can access this administrative console page in one of two ways:

- **Resources** → **Resource Adapters** → **Resource adapters** → ***resource_adapter*** → **J2C activation specifications** → ***activation_specification***.
- **Resources** → **Resource Adapters** → **J2C activation specifications** → ***activation_specification***.

Scope:

Specifies the scope of the resource adapter that supports this activation specification. Only applications that are installed within this scope can use this activation specification.

Provider:

Specifies the resource adapter that encapsulates the appropriate classes for this activation specification.

Name:

Specifies the display name of the J2C activation specification instance.

A string with no spaces meant to be a meaningful text identifier for the J2C activation specification. Name is required

Data type String

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name for the J2C activation specification instance.

The JNDI name is required. If you do not specify one, it is created from the Name field. If not specified, the JNDI name defaults to *eis/[name]*

Data type String

Description:

A free-form text string to describe the J2C activation specification instance.

Data type String

Authentication alias:

This optional field is used to bind the J2C activation specification to an authentication alias (configured through the security JAAS screens).

This alias is used to access a user name and password that are set on the configured J2C activation specification. This field is only meaningful if the J2C activation specification you are configuring has a UserName and Password field.

If you have defined security domains in the application server, you can click **Browse...** to select a J2C authentication alias for the resource that you are configuring. Security domains allow you to isolate J2C authentication aliases between servers. The tree view is useful in determining the security domain to which an alias belongs, and the tree view can help you determine the servers that will be able to access each authentication alias. The tree view is tailored for each resource, so domains and aliases are hidden when you cannot use them.

Data type Text

Message Listener Type:

The Message Listener Type used by this activation specification.

For new objects, the list of available classes is provided by the resource adapter in a drop-down list. After you create the activation specification, the field is a read only text field.

Data type Drop-down list or text

Destination JNDIName:

The destination JNDIName field only appears when a message of type javax.jms.Destination with name *Destination* is received.

Configuring a J2C administered object

Use this task to configure a J2C administered object used to configure objects with an external resource adapter.

About this task

To configure a J2C administered object for an external resource adapter, use the administrative console to complete the following steps. This task contains an optional step for you to create a new administered object.

1. Display the external resource adapter. In the navigation pane, click **Resources** → **Resource Adapters** → *adapter_name*. This displays in the content pane a table of properties for the external resource adapter, including links to the types of J2C resources that it provides.
2. Optional: Change the **Scope** setting to the scope level at which the activation specification is to be visible to applications, according to your needs.
3. In the content pane, under the Additional Properties heading, click **J2C Administered Objects**. This lists any existing J2C administered objects for the external resource adapter in the content pane.
4. Display the properties of the J2C administered object. If you want to display an existing J2C administered object, click one of the names listed.

   Alternatively, if you want to create a new J2C administered object, click **New**, then specify the following required properties:

   Name Type the name by which the J2C administered object is known for administrative purposes. The JNDI name is automatically generated based on the value for the Name property.

   Administered object class
   Select the administered object class that this instance should support. This list is based on the deployment descriptor of the external resource adapter.

   Depending on the external resource adapter, there can be additional required properties that need to be supplied. To provide values for these properties, click **Custom properties**. When creating a new administered object, you may need to click **Apply** before this custom property selection is available.
5. Specify properties for the administered object, according to your needs .
6. Click **OK**.
7. Save your changes to the master configuration.

J2C Administered Objects collection:

Use this page to specify administered object settings for a Resource Adapter.

Administered object definitions and classes are provided by a resource adapter when you install it. Using this information, the administrator can create and configure J2C administered objects with JNDI names that are then available for applications to use. Some messaging styles may need applications to use special administered objects for sending and synchronously receiving messages (through connection objects using messaging style specific APIs). It is also possible that administered objects may be used to perform transformations on an asynchronously received message in a message provider-specific way. Administered objects can be accessed by a component by using either a resource environment reference or a message destination reference (preferred).

You can access this administrative console page in one of two ways:

- **Resources** → **Resource Adapters** → **Resource adapters** → ***resource_adapter*** → **J2C administered objects**
- **Resources** → **Resource Adapters** → **Resource adapters** → **J2C administered objects**

Name:

Specifies display name assigned to this administered object.

Data type String

JNDI Name:

Specifies the JNDI name of the administered object.

Data type String

Scope:

Specifies the scope of the resource adapter that supports this administered object. Only applications that are installed within this scope can use this object.

Provider:

Specifies the resource adapter that encapsulates the appropriate classes for this administrative object.

Description:

Specifies a description for the administered object.

Data type String

Administered object class:

Specifies the Administered Object class that is associated with this J2C administered object. This class must be one that is provided by the resource adapter.

Data type String

J2C Administered Object settings:

Use this page to specify the settings for an administered object.

Administered object definitions and classes are provided by a resource adapter when you install it. Using this information, the administrator can create and configure J2C administered objects with JNDI names that are then available for applications to use. Some messaging styles may need applications to use special administered objects for sending and synchronously receiving messages (through connection objects using messaging style specific APIs). It is also possible that administered objects may be used to perform transformations on an asynchronously received message in a message provider-specific way. Administered objects can be accessed by a component by using either a resource environment reference or a message destination reference (preferred).

You can access this administrative console page in one of two ways:

- **Resources** → **Resource Adapters** → **Resource adapters** → ***resource_adapter*** → **J2C administered objects** → ***J2C_administered_object***
- **Resources** → **Resource Adapters** → **Resource adapters** → **J2C administered objects** → ***J2C_administered_object***

Scope:

Specifies the scope of the resource adapter that supports this administered object. Only applications that are installed within this scope can use this object.

Data type String

Provider:

Specifies the resource adapter that encapsulates the appropriate classes for this administrative object.

Data type String

Name:

Specifies the name of the J2C administered object instance.

A string with no spaces meant to be a meaningful text identifier for the administered object. This name is required.

Data type String

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name that this administered object is bound under.

The JNDI name is required. If you do not specify one, it is created from the Name field. If not specified, the JNDI name defaults to *eis/[name]*

Data type String

Description:

Specifies a text description of the J2C administered object instance.

Data type String

Administered object class:

For new objects, the list of available classes is provided by the resource adapter in a drop-down list. You can only select classes from this list.

After you create the administered object, you cannot modify the administered object class; it is read only.

Data type Class name

Configuring message listener resources for message-driven beans

Use the following tasks to configure resources needed by the message listener service to support message-driven beans for use with a JMS provider that does not have a J2EE Connector Architecture (JCA) 1.5 resource adapter.

Before you begin

Before configuring message listener resources for a message-driven bean, consider the Message Listener Service implementation on the z/OS platform, which affects how you should configure a listener port. For more information about the considerations, see “Message Listener Service on z/OS.”

About this task

For JMS messaging, message-driven beans can use a JMS provider that has a JCA 1.5 resource adapter, such as the default messaging provider that is part of WebSphere Application Server Version 6. With a JCA 1.5 resource adapter, you deploy EJB 2.1 message-driven beans as JCA resources to use a J2C activation specification. If the JMS provider does not have a JCA 1.5 resource adapter, such as the V5 Default Messaging and WebSphere MQ, you must configure JMS message-driven beans against a listener port (as in WebSphere Application Server Version 5).

If the message-driven bean uses a queue hosted by WebSphere MQ as a JMS provider, you should optimize performance by configuring the queue destination properties to best fit your message-driven bean. For more information about performance considerations, see “Performance considerations for WebSphere MQ queue destinations” on page 1213.

Here are some guidelines on which scenarios use listener ports or activation specifications:

- If you are using J2EE 1.2 and EJB 1.1 with WebSphere Application Server v4, MDBs are not used, so you do not used listener ports or activation specifications because WebSphere Application Server v4 uses message beans, but these are not MDbs or EJBs.
- If you are using J2EE 1.3 and EJB 2.0 with WebSphere Application Server v5, you must use listener ports. The MDBs are JMS MDBs that implement MessageListener, and there is no JCA support. WebSphere Application Server v5 uses listener ports to associate MDB classes with their JMS destinations.
- If you are using J2EE 1.4 and EJB 2.1 with WebSphere Application Server v6, the decision depends on whether your JMS provider API is implemented with JCA. In J2EE 1.4, the JMS 1.1 API can be implemented with the JCA 1.5 API.
  - If your JMS provider API is implemented with JCA, your MDB is a JMS MDB that is implemented as a connector MDB. A connector MDB uses JCA to access its resources, and so the connector must be configured with an activation specification. This is for new bean development, and does not affect the conversion of MDBs from EJB 2.0 to EJB 2.1.
  - If your JMS provider API is not implemented with JCA, you have the same JMS situation as for Java EE 1.3, and you must configure this EJB 2.1 MDB in the same way as you would configure an EJB 2.0 MDB, which in WebSphere Application Server is to use a listener port.

If you want to deploy an enterprise application to use JMS message-driven beans with a JMS provider that does not have a JCA 1.5 resource adapter, refer to the following subtopics:

- “Message Listener Service on z/OS”
- “Configuring the message listener service” on page 1225
- “Creating a new listener port” on page 1232
- “Configuring a listener port” on page 1233
- “Deleting a listener port” on page 1234
- “Configuring security for message-driven beans that use listener ports” on page 1234
- “Administering listener ports” on page 1235

Message Listener Service on z/OS:

On the z/OS platform, the Message Listener Service implementation exploits the scalable server architecture provided by WebSphere Application Server for z/OS, by dividing the listener port functionality across the controller and servant processes.

A single message listener is registered in the controller portion of the ListenerPort, while application dispatch (that is, dispatch of the user application's onMessage(Message) EJB method) is then spread across the various servants comprising the server. This implementation is referred to as "listening in the controller" for the message-driven bean messages.

Note: An important point to note is that the WebSphere Application Server for z/OS Message Listener Service implementation does not use the just-described "listening in the controller" internal configuration in all cases, but rather only in the cases in which the message-driven bean is mapped to a queue (that is, listening on a Queue) or mapped (listening) to a topic through a durable subscription.

In the case in which the message-driven bean is listening to a topic through a nondurable subscription, the listener port registers a listener in *each servant*, rather than registering a single listener in the controller.

For a topic on which a message-driven bean is listening through a non-durable subscription, a single message published to that topic is dispatched once for each servant comprising the server. In contrast, if a message-driven bean is listening on a queue, or a topic through a durable subscription, a single message sent is dispatched once only, to a single servant, for the entire server.

The configuration of the listener port, in the "listening in the controller" case, is administered externally through the controls described in Tuning MDB processing on z/OS and Concepts and considerations for MDB settings on z/OS.

For a non-durable subscription message-driven bean, which is listening in each servant as described above, the listener ports are configured in the same manner as they are on distributed platforms. The single set of settings configured for the listener port are applied identically to each servant in this case.

Configuring the message listener service:

Use this task to configure the properties of the message listener service for an application server, to support message-driven beans deployed against listener ports.

About this task

If the JMS provider does not have a J2EE Connector Architecture (JCA) 1.5 resource adapter, such as the V5 Default Messaging and WebSphere MQ, you must configure JMS message-driven beans against a listener port (as in WebSphere Application Server Version 5).

If you want to deploy an enterprise application to use message-driven beans with listener ports, you can use this task to browse or change the configuration of the message listener service for an application server.

To configure the message listener service for an application server, use the administrative console to complete the following steps:

1. Display the listener service settings page:
   a. In the navigation pane, select **Servers** → **Application Servers**.
   b. In the content pane, click the name of the application server.
   c. Under Communications, click **Messaging** → **Message Listener Service**.
2. Optional: Browse or change the value of properties for the message-driven bean thread pool.
   a. Click **Thread Pool**

b. Change the following properties, to suit your needs:

Minimum size
The minimum number of threads to allow in the pool.

Maximum size
The maximum number of threads to allow in the pool.

Thread inactivity timeout
The number of milliseconds of inactivity that should elapse before a thread is reclaimed. A value of 0 indicates not to wait and a negative value (less than 0) means to wait forever.

Note: The administrative console does not allow you to set the inactivity timeout to a negative number. To do this you must modify the value directly in the config.xml file.

Allow thread allocation beyond maximum thread size
Select this check box to enable the number of threads to increase beyond the maximum size configured for the thread pool.

c. Click **OK**.

3. Optional: Specify any of the following optional properties that you need, as **Custom properties** of the message listener service:

   NON.ASF.RECEIVE.TIMEOUT, MQJMS.POOLING.TIMEOUT, MQJMS.POOLING.THRESHOLD, MAX.RECOVERY.RETRIES, and RECOVERY.RETRY.INTERVAL.

   For more information about these custom properties, see Custom Properties.

   To browse or change the properties, complete the following steps:

   a. Click **Custom properties**
   b. For each custom property, specify a value to suit your needs.

      If you have not specified a property before:

      1) Click **New**.
      2) Type the name of the property.
      3) Type the value of the property.
      4) Click **OK**.

4. Save your changes to the master configuration.
5. To have the changed configuration take effect, stop then restart the Application Server.

Message listener service:

The message listener service is an extension to the JMS functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

This panel displays links to the Additional Properties pages for Listener Ports and Custom Properties for the message listener service.

To view this administrative console page, click **Servers** → **Application Servers** → *application_server* → **[Communications] Messaging** → **Message Listener Service**

Custom Properties:

An optional set of name and value pairs for custom properties of the message listener service.

You can use the Custom properties page to define the following properties for use by the message listener service.

- NON.ASF.RECEIVE.TIMEOUT

- MQJMS.POOLING.TIMEOUT
- MQJMS.POOLING.THRESHOLD
- MAX.RECOVERY.RETRIES
- RECOVERY.RETRY.INTERVAL
- “DYNAMIC.CONFIGURATION.ENABLED” on page 1232

Message listener port collection:

The message listener ports configured in the administrative domain

This panel displays a list of the message listener ports configured in the administrative domain. Each listener port is used with a message-driven bean to automatically receive messages from an associated JMS destination. You can use this panel to add new listener ports or to change the properties of existing listener ports.

Note: From WebSphere Application Server Version 7 listener ports are deprecated. For information about the facilities available to aid migration of configuration information from a listener port to an activation specification for use with the Websphere MQ messaging provider, refer to related tasks.

To view this administrative console panel, click **Servers** → **Application Servers** → *application_server* → **[Messaging] Message Listener Service** → **Listener Ports**

To manage a listener port, enable the **Select** check box beside the listener port name in the list and click a button:

Button	Resulting action
Convert to activation specification	Opens a wizard that helps you convert the selected listener port to an activation specification.
New	Accesses the panel to configure a new listener port.
Delete	Deletes the selected listener port or ports.
Start	Starts the selected listener port or ports.
Stop	Stops the selected listener port or ports.

For more information about asynchronous messaging, see “Asynchronous messaging in WebSphere Application Server using JMS” on page 309.

Listener port settings:

A listener port is used to simplify administration of the association between a connection factory, destination, and deployed message-driven bean.

Use this panel to view or change the configuration properties of the selected listener port.

To view this administrative console page, click **Servers** → **Application Servers** → *application_server* → **[Communications] Messaging** → **Message Listener Service** → **Listener Ports** → *listener_port*

Name:

The name by which the listener port is known for administrative purposes.

Data type String
Default Null

Initial state:

The state that you want the listener port to have when the application server is next restarted

Data type Enum
Units Not applicable
Default Started
Range
Started
When the application server is next started, the listener port is started automatically.
Stopped
When the application server is next started, the listener port is not started automatically. If message-driven beans are to use this listener port on the application server, the system administrator must start the port manually or select the Started value of this property then restart the application server.

Description:

A description of the listener port, for administrative purposes within IBM WebSphere Application Server.

Data type String
Default Null

Connection factory JNDI name:

The JNDI name for the JMS connection factory to be used by the listener port; for example, `jms/connFactory1`.

Data type String
Default Null

Destination JNDI name:

The JNDI name for the destination to be used by the listener port; for example, `jms/destn1`.

You cannot use a temporary destination for late responses.

Data type String
Default Null

Maximum sessions:

Specifies the maximum number of concurrent sessions that a listener can have with the JMS server to process messages.

Each session corresponds to a separate listener thread and therefore controls the number of concurrently processed messages. Adjust this parameter when the server does not fully use the available capacity of the machine and if you do not need to process messages in a specific message order.

Data type Integer
Units Sessions
Default 1

Range	1 through 2147483647
Recommended	• If you want to process messages in a strict message order, set the value to 1, so only one thread is ever processing messages. • If you want to process multiple messages simultaneously (known as “message concurrency”), set this property to a value greater than 1. Keep this value as low as possible to prevent overloading client applications. A good starting point for a 100% JMS workload with short transaction times is 2 to 4 sessions per processor. If longer running transactions exist, you may need more sessions, which should be determined by experimentation.

Maximum retries:

The maximum number of times that the listener tries to deliver a message to a message-driven bean instance before the listener is stopped, in the range 0 through 2147483647.

Note: A WebSphere MQ queue has a similar property called the **BackoutThreshold** property. If your listener port is reading from a WebSphere MQ queue, then the retry limit and the behavior when the limit is reached is determined by whichever of these two properties is set to the lower limit:

- If you exceed the WebSphere MQ queue **BackoutThreshold** limit, the message that cannot be delivered is moved to somewhere else by WebSphere MQ (for example, to the WebSphere MQ backout requeue queue or the WebSphere MQ dead letter queue) and the listener port services the next message on the queue. In this case, WebSphere Application Server might not know that the message has not been delivered successfully.
- If you exceed the listener port **maximum retries** limit, the listener port stops. You then manually intervene to investigate the problem, possibly to remove the message from the WebSphere MQ queue then restart the listener port.

Data type	Integer
Units	Retry attempts
Default	0 (no retries)
Range	0 (no retries) through 2147483647

Maximum messages:

The maximum number of messages that the listener can process in one transaction.

If the queue is empty, the listener processes each message when it arrives. Each message is processed within a separate transaction.

For the WebSphere V5 default messaging provider or WebSphere MQ as the JMS provider, if messages start accumulating on the queue then the listener can start processing messages in batches. For third-party messaging providers, this property value is passed to the JMS provider but the effect depends on the JMS provider.

Data type	Integer
Units	Number of messages
Default	1
Range	1 through 2147483647

Recommended	For the WebSphere default messaging providers or WebSphere MQ as the JMS provider, if you want to process multiple messages in a single transaction, then set this value to more than 1. If messages start accumulating on the queue, then a value greater than 1 enables multiple messages to be batch-processed into a single transaction, and eliminates much of the overhead of transactions on JMS messages. **Note:** • If one message in the batch fails processing with an exception, the entire batch of messages is put back on the queue for processing. • Any resource lock held by any of the interactions for the individual messages are held for the duration of the entire batch. • Depending on the amount of processing that messages need, and if XA transactions are being used, setting a value greater than 1 can cause the transaction to time out. If an XA transaction does time out routinely because processing multiple messages exceeds the transaction timeout, reduce this property to 1 (to limit processing to one message per transaction) or increase your transaction timeout.

Message listener service custom properties:

Use this panel to view or change an optional set of name and value pairs for custom properties of the message listener service.

To view this administrative console page, click **Servers** → **Application Servers** → *application_server* → **[Communications] Messaging** → **Message Listener Service** → **Custom Properties**

You can use the Custom properties page to define the following properties for use by the message listener service.

- NON.ASF.RECEIVE.TIMEOUT
- MQJMS.POOLING.TIMEOUT
- MQJMS.POOLING.THRESHOLD
- MAX.RECOVERY.RETRIES
- RECOVERY.RETRY.INTERVAL
- DYNAMIC.CONFIGURATION.ENABLED

NON.ASF.RECEIVE.TIMEOUT:

The timeout in milliseconds for synchronous message receives performed by message-driven bean listener sessions in the non-ASF mode of operation.

You should set this property to a non-zero value only if you want to enable the non-ASF mode of operation for all message-driven bean listeners on the application server.

The message listener service has two modes of operation, Application Server Facilities (ASF) and non-Application Server Facilities (non-ASF).

- The ASF mode is meant to provide concurrency and transactional support for applications. For publish/subscribe message-drive beans, the ASF mode provides better throughput and concurrency, because in the non-ASF mode the listener is single-threaded.

- The non-ASF mode is mainly for use with third-party messaging providers that do not support JMS ASF, which is an optional extension to the JMS specification. The non-ASF mode is also transactional but, because the path length is shorter than the ASF mode, usually provides improved performance.

  Use non-ASF if:
  - Your third-party messaging provider does not provide JMS ASF support
  - You are using message-driven beans with WebSphere topic connections with the DIRECT port, because the embedded publish/subscribe broker using that port does not support XA transactions or JMS ASF.
  - Message order is a strict requirement

Data type Integer

Units Milliseconds

Default ASF mode (custom property not created)

Range 0 or greater milliseconds

0 non-ASF mode is disabled

1 or more
The timeout in milliseconds for non-ASF message-driven bean listener synchronous session receives

Recommended If a transaction timeout occurs, the message must recycle causing extra work. If you want to use the non-ASF mode, set this property to lower than the transaction timeout, but leave spare at least the maximum duration of your message-driven bean's onMessage() method. For example, if your message-driven bean's onMessage() method typically takes a maximum of 10 seconds, and the transaction timeout is set to 120 seconds, you might set the NON.ASF.RECEIVE.TIMEOUT property to no more than 110000 (110000 milliseconds, that is 110 seconds).

MQJMS.POOLING.TIMEOUT:

The number of milliseconds after which a connection in the pool is destroyed if it has not been used.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes.

Data type Integer

Units Milliseconds

Default 5 minutes

Range

MQJMS.POOLING.THRESHOLD:

The maximum number of unused connections in the pool.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if there are more than ten unused connections in the pool.

Data type Integer

Units Number of connections

Default 10

Range

MAX.RECOVERY.RETRIES:

The maximum number of times that a listener port managed by this service tries to recover from a failure before giving up and stopping. When stopped the associated listener port is changed to the stop state. The interval between retry attempts is defined by the RECOVERY.RETRY.INTERVAL custom property.

A failure can be one of two things:

- An unexpected error has occurred when a listener port tries to get a message from the JMS provider.
- The connection between the application server and the JMS provider has been lost, usually due to a network error.

Data type	Integer
Units	Retry attempts
Default	5
Range	0 (no retries) through 2147483647

RECOVERY.RETRY.INTERVAL:

The time in seconds between retry attempts by a listener port to recover from a failure. The maximum number of retry attempts is defined by the MAX.RECOVERY.RETRIES custom property.

A failure can be one of two things:

- An unexpected error has occurred when a listener port tries to get a message from the JMS provider.
- The connection between the application server and the JMS provider has been lost, usually due to a network error.

Data type	Integer
Units	Seconds
Default	60
Range	1 through 2147483647

DYNAMIC.CONFIGURATION.ENABLED:

This property controls whether the application server on which a listener port is created requires to be restarted. Set this property to true to enable dynamic configuration.

Data type	Boolean
Default	False (not selected)

Creating a new listener port:

Use this task to create a new listener port for the message listener service, so that message-driven beans can be associated with the port to retrieve messages.

About this task

Although you can continue to deploy an EJB 2.0 message-driven bean against a listener port (as in WebSphere Application Server Version 5), you are recommended to deploy such beans as J2EE Connector Architecture (JCA) 1.5-compliant resources and to upgrade them to be EJB 2.1 message-driven beans.

If you want to deploy an enterprise application to use EJB 2.0 message-driven beans with listener ports, use this task to create a new listener port for a message-driven bean to retrieve messages from.

To create a new listener port, use the administrative console to complete the following steps:

1. Display the collection list of listener ports:
   a. In the navigation pane, select **Servers** → **Application Servers**.
   b. In the content pane, click the name of the application server.
   c. Under Communications, click **Messaging** → **Message Listener Service**.
   d. Click **Listener Ports**.
2. Click **New**.
3. Specify the following required properties:

   Name The name by which the listener port is known for administrative purposes.

   Connection factory JNDI name
   The JNDI name for the JMS connection factory to be used by the listener port; for example, `jms/connFactory1`

   Destination JNDI name
   The JNDI name for the destination to be used by the listener port; for example, `jms/destn1`.
4. Optional: Change other properties for the listener port, according to your needs.
5. Click **OK**.
6. Save your changes to the master configuration.
7. To have the changed configuration take effect, stop then restart the application server.

Results

If enabled, the listener port is started automatically when a message-driven bean associated with that port is installed.

Configuring a listener port:

Use this task to browse or change the properties of an existing listener port, used by message-driven beans associated with the port to retrieve messages.

About this task

Although you can continue to deploy an EJB 2.0 message-driven bean against a listener port (as in WebSphere Application Server Version 5), you are recommended to deploy such beans as J2EE Connector Architecture (JCA) 1.5-compliant resources and to upgrade them to be EJB 2.1 message-driven beans.

If you have deployed an enterprise application to use EJB 2.0 message-driven beans with listener ports, use this task to browse or change the configuration of a listener port that a message-driven bean retrieves messages from.

Note: From WebSphere Application Server Version 7 listener ports are deprecated. For information about the facilities available to aid migration of configuration information from a listener port to an activation specification for use with the Websphere MQ messaging provider, refer to related tasks.

To configure the properties of a listener port, use the administrative console to complete the following steps:

1. Display the collection list of listener ports:
   a. In the navigation pane, select **Servers** → **Application Servers**.

b. In the content pane, click the name of the application server.
c. Under Communications, click **Messaging** → **Message Listener Service**.
d. Click **Listener Ports**.

2. Click the name of the listener port that you want to work with. This displays the properties of the listener port in the content pane.
3. Optional: Change properties for the listener port, according to your needs.
4. Click **OK**.
5. Save any changes to the master configuration.
6. To have a changed configuration take effect, stop then restart the application server.

Deleting a listener port:

Use this task to delete a listener port from the message listener service, to prevent message-driven beans associated with the port from retrieving messages.

About this task

To delete a listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. Under Communications, click **Messaging** → **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, click **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, select the check box for the listener port that you want to delete.
6. Click **Delete**. This action stops the port (needed to allow the port to be deleted) then deletes the port.
7. To save your configuration, click **Save** on the task bar of the Administrative console window.
8. To have the changed configuration take effect, stop then restart the application server.

Configuring security for message-driven beans that use listener ports:

Use this task to configure resource security and security permissions for message-driven beans.

About this task

There are two special security considerations when using message-driven beans (MDBs). In other respects, however, the security considerations for an MDB are identical to those of any other enterprise bean. For instance, access to JDBC resources and Java EE Connector Architecture (JCA) resources (for example CICS, IMS) is handled in the same way as for an entity or session bean.Access to other JMS resources is also handled in the same way as for other enterprise beans.

However to understand this last point about JMS access correctly, it is important to understand that the security considerations when configuring the MDB listener, which can be thought of as part of the application server infrastructure, are unique to MDBs. These considerations which are specific to MDBs are relevant when configuring authentication and authorization for the server to connect to a JMS provider and a Destination so that a message can be selected and so that the MDB can pass this message to the its onMessage() method.

The user's MDB onMessage() application code might not make additional JMS calls, however if the MDB application code accesses additional JMS resources, it is this access which is handled identically to JMS calls made by an entity or session EJB.

MBD security considerations:

The MDB listener's security information is established when the MDB listener's JMS Connection is created. This is the typical JMS programming pattern. The properties used to configure the MDB listener's JMS Connection Factory are also used for specifying these security parameters. By configuring the Connection Factory mapped to in the Listener Port definition, you can control the security parameters used by the MDB listener. The JMS Connection used by a given MDB listener is obtained in the order of precedence based on the configuration of the JMS Connection Factory used by the Message Listener Service Listener Port onto which a given MDB is mapped. For example, if an MDB, mdb1 is mapped onto Listener Port mylp1 and mylp1 uses ConnectionFactory qcf1, you would configure qcf1 to control the configuration of mdb1's MDB listener. The order of precedence is:

1. If a container-managed alias has been defined for this Connection Factory, the userid associated with the container-managed alias is used in the Connection creation call, for example createQueueConnection(userid,password)).
2. If a component-managed alias has been defined for this Connection Factory, the userid associated with the component-managed alias is used.
3. If neither alias is specified and the Connection Factory is defined in Bindings mode (that is, TransportType = "BINDINGS"), the server identity is used. The server identity translates more specifically into the servant identity in the servants, and the controller identity in the controller. In the case of listening-in controller, the controller identity is relevant as well as the servant identity. For related information about listening-in controllers, see "Message Listener Service on z/OS" on page 1224.

Note: The authentication aliases referred to here are those associated with the Connection Factory defined by the Administrator. No application resource reference is associated with the MDB listener and so no authentication alias has to be set at that level.

To set the container-managed alias, (if you elect that option), use the administrative console to complete the following steps:

1. To display the listener port settings, click **Servers** → **Server types** → **WebSphere application servers** → ***application_server*** → **[Communications] Messaging** → **Message listener service** → **[Additional properties] Listener ports** → ***listener_port***
2. To get the name of the JMS connection factory, look at the Connection factory JNDI name property.
3. Display the JMS connection factory properties. For example, to display the properties of a queue connection factory, click **Resources** → **JMS** → **Queue Connection Factories*connection_factory***.
4. Set the "Container-managed authentication alias" property.
5. Click **OK**

Results

Considerations for invoking other EJBs:

Messages arriving at a listener port have no client credentials associated with them. The messages are anonymous. To call secure enterprise beans from a message-driven bean, the message-driven bean must be configured with a RunAs Identity deployment descriptor. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component.

For more information about EJB security, see Securing enterprise bean applications. For more information about configuring security for your application, see Securing applications during assembly and deployment.

Administering listener ports:

Use the following tasks to administer listener ports, which each define the association between a connection factory, a destination, and a message-driven bean.

About this task

Note: From WebSphere Application Server Version 7 listener ports are deprecated. For information about the facilities available to aid migration of configuration information from a listener port to an activation specification for use with the Websphere MQ messaging provider, refer to related tasks.

You can use the WebSphere Application Server administrative console to administer listener ports, as described in the following tasks.

Note: If configured as enabled, a listener port is started automatically when a message-driven bean associated with that port is installed. You do not normally need to start or stop a listener port manually.

- Adding a new listener port Use this task to create a new listener port, to specify a new association between a connection factory, a destination, and a message-driven bean. This enables deployed message-driven beans associated with the port to retrieve messages from the destination.
- Configuring a listener port Use this task to browse or change the configuration properties of a listener port.
- Starting a listener port Use this task to start a listener port manually.
- Stopping a listener port Use this task to stop a listener port manually.

Starting a listener port:

Use this task to start a listener port on an application server, to enable the listeners for message-driven beans associated with the port to retrieve messages.

About this task

A listener is active, that is able to receive messages from a destination, if the deployed message-driven bean, listener port, and message listener service are all started. Although you can start these components in any order, they must all be in a started state before the listener can retrieve messages.

If configured as enabled, a listener port is started automatically when a message-driven bean associated with that port is installed. However, you can start a listener port manually, as described in this topic.

When a listener port is started, the listener manager tries to start the listeners for each message-driven bean associated with the port. If a message-driven bean is stopped, the port is started but the listener is not started, and remains stopped. If you start a message-driven bean, the related listener is started.

To start a listener port on an application server, use the administrative console to complete the following steps:

1. If you want the listener for a deployed message-driven bean to be able to receive messages at the port, check that the message-driven bean has been started.
2. Display the collection list of listener ports:
   a. In the navigation pane, select **Servers** → **Application Servers**.
   b. In the content pane, click the name of the application server.
   c. Under Communications, click **Messaging** → **Message Listener Service**.
   d. Click **Listener Ports**.
3. Select the check box for the listener port that you want to start.
4. Click **Start**.
5. Save your changes to the master configuration.

Stopping a listener port:

Use this task to stop a listener port on an application server, to prevent the listeners for message-driven beans associated with the port from retrieving messages.

About this task

When you stop a listener port as described in this topic, the listener manager stops the listeners for all message-driven beans associated with the port.

To stop a listener port on an application server, use the administrative console to complete the following steps:

1. Display the collection list of listener ports:
   a. In the navigation pane, select **Servers** → **Application Servers**.
   b. In the content pane, click the name of the application server.
   c. Under Communications, click **Messaging** → **Message Listener Service**.
   d. Click **Listener Ports**.
2. Select the check box for the listener port that you want to stop.
3. Click **Stop**.
4. Save your changes to the master configuration.
5. To have the changed configuration take effect, stop then restart the application server.

Message-driven beans - listener port components:

The WebSphere Application Server support for message-driven beans deployed against listener ports is based on JMS message listeners and the message listener service, and builds on the base support for JMS.

Note: From WebSphere Application Server Version 7 listener ports are deprecated. For information about the facilities available to aid migration of configuration information from a listener port to an activation specification for use with the Websphere MQ messaging provided, refer to related tasks.

The main components of WebSphere Application Server support for message-driven beans are shown in the following figure and described after the figure:

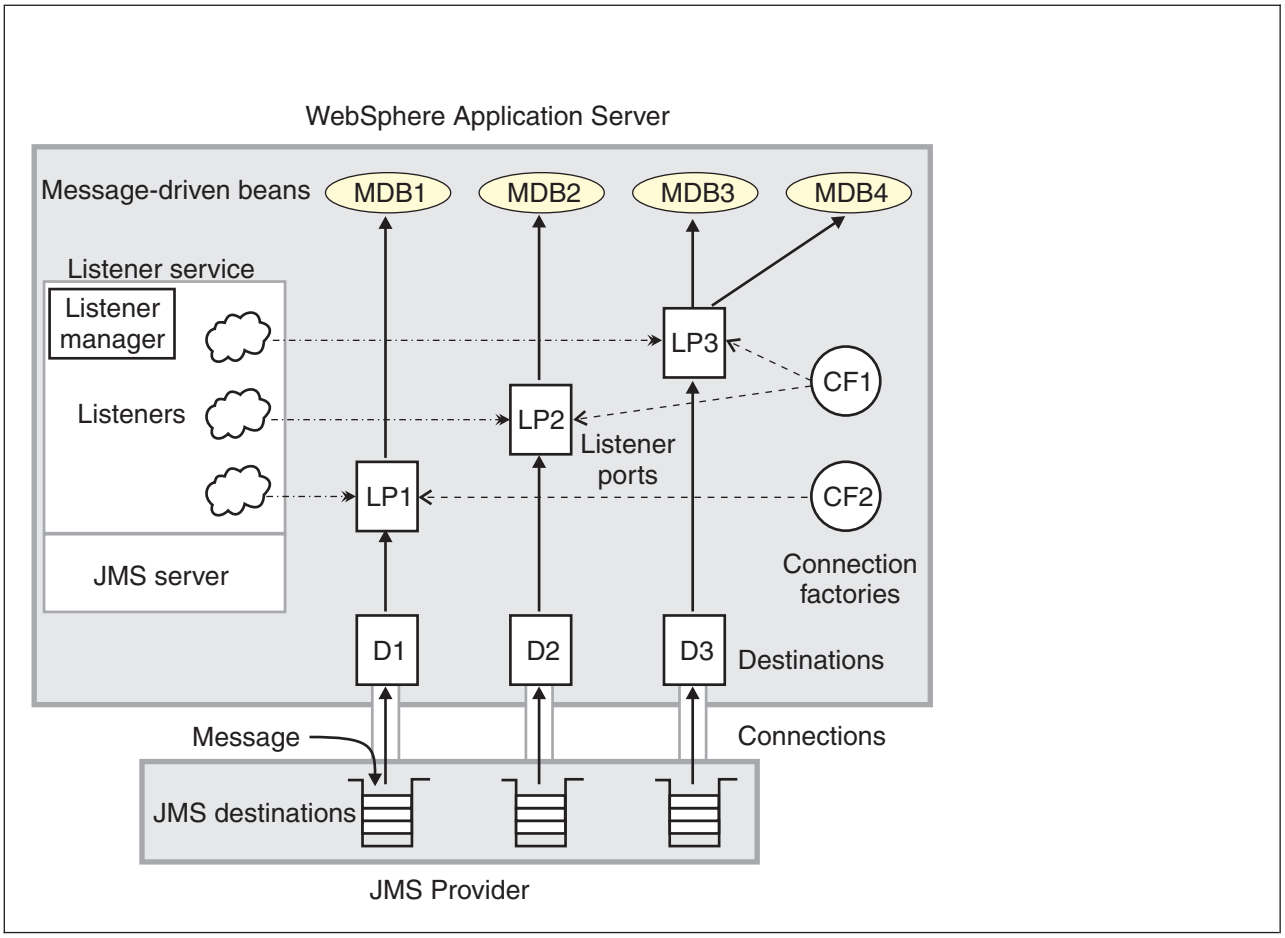


Figure 8. The main components for message-driven beans. This figure shows the main components of WebSphere support for message-driven beans, from JMS provider through a connection to a destination, listener port, then deployed message-driven bean that processes the message retrieved from the destination. Each listener port defines the association between a connection factory, destination, and a deployed message-driven bean. The other main components are the message listener service, which comprises a listener for each listener port, all controlled by the same listener manager. For more information, see the text that accompanies this figure.

The *message listener service* is an extension to the JMS functions of the JMS provider and provides a *listener manager*, which controls and monitors one or more JMS *listeners*.

Each listener monitors either a JMS queue destination (for point-to-point messaging) or a JMS topic destination (for publish/subscribe messaging).

A *connection factory* is used to create connections with the JMS provider for a specific JMS queue or topic destination. Each connection factory encapsulates the configuration parameters needed to create a connection to a JMS destination.

A *listener port* defines the association between a connection factory, a destination, and a deployed *message-driven bean*. Listener ports are used to simplify the administration of the associations between these resources.

When a deployed message-driven bean is installed, it is associated with a listener port and the listener for a destination. When a message arrives on the destination, the listener passes the message to a new instance of a message-driven bean for processing.

When an application server is started, it initializes the listener manager based on the configuration data. The listener manager creates a dynamic session thread pool for use by listeners, creates and starts

listeners, and during server termination controls the cleanup of listener message service resources. Each listener completes several steps for the JMS destination that it is to monitor, including:

- Creating a JMS server session pool, and allocating JMS server sessions and session threads for incoming messages.
- Interfacing with JMS ASF to create JMS connection consumers to listen for incoming messages.
- If specified, starting a transaction and requesting that it is committed (or rolled back) when the EJB method has completed.
- Processing incoming messages by invoking the onMessage() method of the specified enterprise bean.

Important file for message-driven beans

The *server_name*-durableSubscriptions.ser file in the WAS_HOME/temp directory is important for the operation of the WebSphere Application Server messaging service, so should not be deleted.

If you do need to delete the WAS_HOME/temp directory or other files in it, ensure that you preserve the following file:

server_name-**durableSubscriptions.ser**
: You should not delete this file, because the messaging service uses it to keep track of durable subscriptions for message-driven beans. If you uninstall an application that contains a message-driven bean, this file is used to unsubscribe the durable subscription.

Programming to use asynchronous messaging

This topic describes things to consider when designing an enterprise application to use the JMS API directly for asynchronous messaging.

About this task

You can build enterprise beans that use the JMS API directly to provide messaging services along with methods that implement business logic. An enterprise application can explicitly poll for messages on a JMS destination then retrieve messages for processing by business logic beans (enterprise beans).

You can also use message-driven beans (a type of enterprise bean defined in the EJB specification) as asynchronous message consumers. A message-driven bean is invoked by the EJB container when a message arrives at the destination that it is configured to use, without an application having to explicitly poll the destination.

- “Programming to use JMS and messaging directly” This topic provides information about using the Java Message Service (JMS) programming interfaces directly to exchange messages asynchronously.
- “Programming to use message-driven beans” on page 1254 This topic provides information about using message-driven beans as asynchronous message consumers.

Programming to use JMS and messaging directly

Use these tasks to implement WebSphere J2EE applications that use JMS programming interfaces directly.

About this task

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interface.

The base JMS support enables WebSphere enterprise applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). An enterprise application can explicitly poll for messages on a destination.

Using the base support for JMS, you can build enterprise beans that use the JMS API directly to provide messaging services along with methods that implement business logic.

You can use the WebSphere administrative console to administer the JMS support of WebSphere Application Server. For example, you can configure JMS providers and their resources, and can control the activity of the JMS server.

For more information about JMS, see the JMS documentation at http://java.sun.com/products/jms/docs.html.

Designing an enterprise application to use JMS

This topic describes things to consider when designing an enterprise application to use the JMS API directly for asynchronous messaging.

About this task

This topic describes the things that you need to consider when designing an enterprise application to use the JMS API directly for asynchronous messaging.

- For messaging operations, you should write application programs that use only references to the interfaces defined in Sun's javax.jms package. JMS defines a generic view of a messaging that maps onto the underlying transport. An enterprise application that uses JMS, makes use of the following interfaces that are defined in Sun's javax.jms package:

  Connection
  Provides access to the underlying transport, and is used to create Sessions.

  Session
  Provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers.

  MessageProducer
  Used to send messages.

  MessageConsumer
  Used to receive messages.

  The generic JMS interfaces are subclassed into the following more specific versions for Point-to-Point and Publish/Subscribe behavior:

JMS Common Interfaces	Point-to-Point	Publish/Subscribe
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession,	TopicSession,
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

  For more information about using these JMS interfaces, see the Java Message Service Documentation and the WebSphere MQ *Using Java* book, SC34-5456.

  The section "Java Message Service (JMS) Requirements" of the J2EE specification gives a list of methods that must not be called in Web and EJB containers:

```
javax.jms.Session method setMessageListener
javax.jms.Session method getMessageListener
javax.jms.Session method run
javax.jms.QueueConnection method createConnectionConsumer
javax.jms.TopicConnection method createConnectionConsumer
javax.jms.TopicConnection method createDurableConnectionConsumer
javax.jms.MessageConsumer method getMessageListener
javax.jms.MessageConsumer method setMessageListener
javax.jms.Connection setExceptionListener
javax.jms.Connection stop
javax.jms.Connection setClientID
```

This method restriction is enforced in IBM WebSphere Application Server by throwing a javax.jms.IllegalStateException.

- Applications refer to JMS resources that are predefined, as administered objects, to WebSphere Application Server.

  Details of JMS resources that are used by enterprise applications are defined to WebSphere Application Server and bound into the JNDI namespace by the WebSphere administrative support. An enterprise application can retrieve these objects from the JNDI namespace and use them without needing to know anything about their implementation. This enables the underlying messaging architecture defined by the JMS resources to be changed without requiring changes to the enterprise application. When designing an enterprise application, you need to identify the details of the following types of JMS resources:

Point-to-Point	Publish/Subscribe
ConnectionFactory (or QueueConnectionFactory) Queue	ConnectionFactory (or TopicConnectionFactory) Topic

  A connection factory is used to create connections from the JMS provider to the messaging system, and encapsulates the configuration parameters needed to create connections.

  For details of the properties of these JMS resources for any JMS provider, see “Configuring JMS resources for a third-party messaging provider” on page 1181. For details of the extended range of properties provided by the default messaging provider, see Configuring resources for the default messaging provider.

- To improve performance, the application server pools connections and sessions with the JMS provider. You need to configure the connection and session pool properties appropriately for your applications, otherwise you may not get the connection and session behavior that you want.
- Applications must not cache JMS connections, sessions, producers or consumers. WebSphere Application Server closes these objects when a bean or servlet completes, and so any attempt to use a cached object will fail with a javax.jms.IllegalStateException.

  To improve performance, applications can cache JMS objects that have been looked up from JNDI. For example, an EJB or servlet needs to look up a JMS ConnectionFactory only once, but it must call the createConnection method on each instantiation. Because of the effect of pooling on connections and sessions with the JMS provider, there should be no performance impact.
- A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.
- Using durable subscriptions with the default messaging provider. A durable subscription on a JMS topic enables a subscriber to receive a copy of all messages published to that topic, even after periods of time when the subscriber is not connected to the server. Therefore, subscriber applications can operate disconnected from the server for long periods of time, and then reconnect to the server and process messages that were published during their absence. If an application creates a durable subscription, it is added to the runtime list that administrators can display and act on through the administrative console.

  Each durable subscription is given a unique identifier, *clientID##subName* where:

  clientID
  : The client identifier used to associate a connection and its objects with the messages maintained for applications (as clients of the JMS provider). You should use a naming convention that helps you identify the applications, in case you need to relate durable subscriptions to the associated applications for runtime administration.

  subName
  : The subscription name used to uniquely identify a durable subscription within a given client identifier.

For durable subscriptions created by message-driven beans, these values are set on the JMS activationSpec. For other durable subscriptions, the client identifier is set on the JMS connection factory, and the subscription name is set by the application on the createDurableSubscriber operation.

To create a durable subscription to a topic, an application uses the createDurableSubscriber operation defined in the JMS API:

```
public TopicSubscriber createDurableSubscriber(Topic topic,
                          java.lang.String subName,
                          java.lang.String messageSelector,
                          boolean noLocal)
                          throws JMSException
```

topic The name of the JMS topic to subscribe to. This is the name of an object supporting the javax.jms.Topic interfaces, such as found by looking up a suitable JNDI entry.

subName
The name used to identify this subscription.

messageSelector
Only messages with properties matching the message selector expression are delivered to consumers. A value of null or an empty string indicates that all messages should be delivered.

noLocal
If set to true, this prevents the delivery of messages published on the same connection as the durable subscriber.

Applications can use a two argument form of createDurableSubscriber that takes only topic and subName parameters. This alternative call directly invokes the four argument version shown above, but sets messageSelector to null (so all messages are delivered) and sets noLocal to false (so messages published on the connection are delivered). For example, to create a durable subscription to the topic called myTopic, with the subscription name of mySubscription:

```
session.createDurableSubscriber(myTopic,"mySubscription");
```

If the createDurableSubscription operation fails, it throws a JMS exception that provides a message and linked exception to give more detail about the cause of the problem.

To delete a durable subscription, an application uses the unsubscribe operation defined in the JMS API

In normal operation there can be at most one active (connected) subscriber for a durable subscription at a time. However, the subscriber application can be running in a cloned application server, for failover and load balancing purposes. In this case the “one active subscriber” restriction is lifted to provide a shared durable subscription that can have multiple simultaneous consumers.

For more information about application use of durable subscriptions, see the section “Using Durable Subscriptions” in the JMS specification.

- Decide what message selectors are needed. You can use the JMS message selector mechanism to select a subset of the messages on a queue so that this subset is returned by a receive call. The selector can refer to fields in the JMS message header and fields in the message properties.
- Acting on messages received. When a message is received, you can act on it as needed by the business logic of the application. Some general JMS actions are to check that the message is of the correct type and extract the content of the message. To extract the content from the body of the message, you need to cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as TextMessage. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

  In this example, the instanceof operator is used to check that the message received is of the TextMessage type. The message content is then extracted by casting to the TextMessage subclass.

```
      if ( inMessage instanceof TextMessage )

...
          String replyString = ((TextMessage) inMessage).getText();
```

- JMS applications using the default messaging provider can access, without any restrictions, the content of messages that have been received from WebSphere Application Server Version 5 embedded messaging or WebSphere MQ.
- JMS applications can access the full set of JMS_IBM* properties. These properties are of value to JMS applications that use resources provided by the default messaging provider, the V5 default messaging provider, or the WebSphere MQ provider.

  For messages handled by WebSphere MQ, the JMS_IBM* properties are mapped to equivalent WebSphere MQ Message Descriptor (MQMD) fields. For more information about the JMS_IBM* properties and MQMD fields, see the *WebSphere MQ: Using Java* book, SC34-6066.
- JMS applications can use report messages as a form of managed request/response processing, to give remote feedback to producers on the outcome of their send operations and the fate of their messages. JMS applications can request a full range of report options using JMS_IBM_Report_Xxxx message properties. For more information about using JMS report messages, see "JMS report messages" on page 1245.
- JMS applications can use the JMS_IBM_Report_Discard_Msg property to control how a request message is disposed of if it cannot be delivered to the destination queue.

  MQRO_Dead_Letter_Queue
  This is the default. The request message should be written to the dead letter queue.

  MQRO_Discard
  The request message should be discarded. This is usually used in conjunction with MQRO_Exception_With_Full_Data to return an undeliverable request message to its sender.
- Using a listener to receive messages asynchronously. In a client, not in a servlet or enterprise bean, an alternative to making calls to QueueReceiver.receive() is to register a method that is called automatically when a suitable message is available; for example:

```
...
MyClass listener =new MyClass();
queueReceiver.setMessageListener(listener);
//application continues with other application-specific behavior.
...
```

  When a message is available, it is retrieved by the onMessage() method on the listener object.

```
import javax.jms.*;
public class MyClass implements MessageListener
{
public void onMessage(Message message)
{
System.out.println("message is "+message);
//application specific processing here
...

}
}
```

  For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to receive() methods. To cope with this situation, you can register an ExceptionListener, which is an instance of a class that implements the onException()method. When an error occurs, this method is called with the JMSException passed as its only parameter.

  For more details about using listeners to receive messages asynchronously, see the Java Message Service Documentation.

  Note: An alternative to developing your own JMS listener class, you can use a message-driven bean, as described in Programming with message-driven beans.
- Take care when performing a JMS receive() from a server-side application component if that receive() invocation is waiting on a message produced by another application component that is deployed in the same server. Such a JMS receive() is synchronous, so blocks until the response message is received.

This type of application design can lead to the consumer/producer problem where the entire set of work threads can be exhausted by the receiving component, which has been blocked waiting for responses, leaving no available worker thread for which to dispatch the application component that would generate the response JMS message.

To illustrate this problem, picture a servlet and a message-driven bean deployed in the same server. When this servlet dispatches a request it sends a message to a queue which is serviced by the message-driven bean (that is, messages produced by the servlet are consumed by the message-driven bean's onMessage() method). The servlet subsequently issues a receive(), waiting for a reply on a temporary ReplyTo queue. The message-driven bean's onMessage() method performs a database query and sends back a reply to the servlet on the temporary queue. If a large number of servlet requests occur at once (relative to the number of server worker threads), then it is likely that all available server worker threads will be used to dispatch a servlet request, send a message, and wait for a reply. The application server enters a deadly-embrace condition whereby no threads remain to process any of the message-driven beans that are now pending. Since the servlets are waiting in blocking recieves, the server hangs, likely leading to application failure.

Possible solutions are:

1. Ensure that the number of worker threads (# of threads per server region * # of server regions per server) exceeds the number of concurrent dispatches of the application component doing the receive() so that there is always a worker thread available to dispatch the message producing component.
2. Use an application topology that places the receiver application component in a separate server than the producer application component. While worker thread usage can still need to be carefully considered under such a deployment scenario, this separation ensures that there are always be threads that cannot be blocked by the message receiving component. There can be other interactions to consider, such as an application server that has multiple applications installed.
3. Refactor your application to do the message receives from a client component, which will not compete with the producer component for worker threads. Furthermore, the client component can do asynchronous (non-blocking) receives, which are prohibited from J2EE servers. So, for example, the example application above could be refactored to have a client sending messages to a queue and then waiting for a response from the MDB.

- If you want to use authentication with WebSphere MQ or the Version 5 Embedded Messaging support, you cannot have user IDs longer than 12 characters. For example, the default Windows NT user ID, **administrator**, is not valid for use with WebSphere internal messaging, because it contains 13 characters.
- The following points, as defined in the EJB specification, apply to the use of flags on create*xxx*Session calls:
  - The transacted flag passed on create*xxx*Session is ignored inside a global transaction and all work is performed as part of the transaction. Outside of a transaction the transacted flag is used and, if set to true, the application should use session.commit() and session.rollback() to control the completion of the work. In an EJB2.0 module, if the transacted flag is set to true and outside of an XA transaction, then the session is involved in the WebSphere local transaction and the unresolved action attribute of the method applies to the JMS work if it is not committed or rolled back by the application.
  - Clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.
- If you want your application to use WebSphere MQ as an external JMS provider, then send messages within a container-managed transaction.

  When you use WebSphere MQ as an external JMS provider, messages sent within a user-managed transaction can arrive before the transaction commits. This occurs only when you use WebSphere MQ as an external JMS provider, and you send messages to a WebSphere MQ queue within a user-managed transaction. The message arrives on the destination queue before the transaction commits.

The cause of this problem is that the WebSphere MQ resource manager has not been enlisted in the user-managed transaction.

The solution is to use a container-managed transaction.

Transaction context impact on non-durable subscribers:

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. A non-durable subscriber is invalidated whenever a sharing boundary (in general, a local or global transaction boundary) is crossed, resulting in a javax.jms.IllegalStateException with message text `Non-durable subscriber invalidated on transaction boundary`.

For example, in the following scenario the non-durable subscriber is invalidated at the begin user transaction. This is because the local transaction context in which the subscriber was created ends when the user transaction begins:

```
...
create subscriber
...
begin user transaction -
...
complete user transaction -
...
use subscriber
...
```

If you want to cache a subscriber (to wait to receive messages that arrived since it was created), then use a durable subscriber (for which this restriction does not apply). Do not cache non-durable subscribers.

JMS report messages:

JMS applications can use report messages as a form of managed request/response processing, to give remote feedback to producers on the outcome of their send operations and the fate of their messages.

JMS applications can request the following types of report message by setting appropriate JMS_IBM_Report_Xxxx message properties and options. The options have the same general syntax and meaning:

MQRO_*report-type*
: A report message of the indicated type is generated that contains the MQMD of the original message. It does not contain any message body data.

MQRO_*report-type*_**WITH_DATA**
: A report message of the indicated type is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_*report-type*_**WITH_FULL_DATA**
: A report message of the indicated type is generated that contains all data from the original message.

For example, to request a COD report message with full data, the JMS application must set JMS_IBM_Report_COD to the value MQRO_COD_WITH_FULL_DATA.

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Exception	Send a report message if the request message cannot be put to the target queue. The exception report messages are generated when a message has been rerouted to an exception destination.	JMS_IBM_Report_Exception • MQRO_EXCEPTION • MQRO_EXCEPTION_WITH_DATA • MQRO_EXCEPTION_WITH_FULL_DATA
Expiration	Send a report message if the request message passes its expiry time.	JMS_IBM_Report_Expiration • MQRO_EXPIRATION • MQRO_EXPIRATION_WITH_DATA • MQRO_EXPIRATION_WITH_FULL_DATA
Confirm on arrival (COA)	Send a report message when the request message has been put to the target queue. For publish/subscribe messaging, the COA report message is generated only on the producers messaging engine. Therefore, such reports are relevant only to local subscriptions. For point-to-point messaging, COA messages are generated when the message arrives at the final destination. For partitioned queues, the report message is generated only when the put operation has committed and a final destination has therefore been selected. Any With_Data or With_Full_Data report options specified are ignored; the COA report message deals only with message headers. If a forward-routing path is used, the COA message are generated when the message arrives at the final destination in the path.	JMS_IBM_Report_COA • MQRO_COA • MQRO_COA_WITH_DATA • MQRO_COA_WITH_FULL_DATA
Confirm on delivery (COD)	Send a report message when the request message has been removed from the queue or topic space by a message consumer. For publish/subscribe messaging, the COD message is generated when all subscribers have received the request message. Therefore, there is one COD message generated for every COA. When a message is consumed by a subscriber, the reference count of the message on the topic space is reduced. When the reference count reaches zero, the message is removed from the topic space then a COD report message is generated. For point-to-point messaging, the COD message is generated after the message has been successfully received by a consuming application. Any With_Data or With_Full_Data report options specified are ignored; the COD report message deals only with message headers.	JMS_IBM_Report_COD • MQRO_COD • MQRO_COD_WITH_DATA • MQRO_COD_WITH_FULL_DATA

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Positive action notification (PAN)	Ask the consumer application to send a report message when it has successfully processed the request message.	JMS_IBM_Report_PAN • MQRO_PAN
Negative action notification (NAN)	Ask the consumer application to send a report message if it has not successfully processed the request message.	JMS_IBM_Report_NAN • MQRO_NAN

The requesting application can control other aspects of the report message as follows:

- How the message Id is generated for the report message and any reply message:

  MQRO_New_Msg_Id
  This the default. A new message Id is generated for the report message.

  MQRO_Pass_Msg_Id
  The message Id of the report message is set to the message Id of the request message.

- How the correlation Id of the report or reply message is to be set.

  MQRO_Copy_Msg_Id_To_Correl_Id
  This the default. the correlation Id of the report message is set to the message Id of the request message.

  MQRO_Pass_Correl_Id
  The correlation Id of the report message is set to the correlation Id of the request message.

For more information about report messages and the associated properties and options, see the *WebSphere MQ: Using Java* book, SC34-6066.

Developing an enterprise application to use JMS

Use this task to develop an enterprise application to use the JMS API directly for asynchronous messaging.

About this task

This topic gives an overview of the steps needed to develop an enterprise application (servlet or enterprise bean) to use the JMS API directly for asynchronous messaging.

This topic only describes the JMS-related considerations; it does not describe general enterprise application programming, which you should already be familiar with. For detailed information about these steps, and for examples of developing an enterprise application to use JMS, see the Java Message Service Documentation

Details of JMS resources that are used by enterprise applications are defined to WebSphere Application Server and bound into the JNDI namespace by the WebSphere administrative support.

To use JMS, complete the following general steps:

1. Import JMS packages. An enterprise application that uses JMS starts with a number of import statements for JMS, which should include at least the following:

```
import javax.jms.*;       //JMS interfaces
import javax.naming.*;    //Used for JNDI lookup of administered objects
```

2. Get an initial context.

```
   try     {
        ctx = new InitialContext(env);
...
```

3. Retrieve administered objects from the JNDI namespace. The InitialContext.lookup() method is used to retrieve administered objects (a JMS connection factory and JMS destinations); for example, to receive a message from a queue

```
qcf = (QueueConnectionFactory)ctx.lookup( qcfName );
...
inQueue = (Queue)ctx.lookup( qnameIn );
...
```

An alternative, but less manageable, approach to obtaining administratively-defined JMS destination objects by JNDI lookup is to use the Session.createQueue(String) method or Session.createTopic(String) method. For example,

```
Queue q = mySession.createQueue("Q1");
```

creates a JMS Queue instance that can be used to reference the existing destination Q1.

In its simplest form, the parameter to these create methods is the name of an existing destination. For more complex situations, applications can use a URI-based format, which allows an arbitrary number of name value pairs to be supplied to set various properties of the JMS destination object.

4. Create a connection to the messaging service provider. The connection provides access to the underlying transport, and is used to create sessions. The createQueueConnection() method on the factory object is used to create the connection.

```
connection = qcf.createQueueConnection();
```

The JMS specification defines that connections should be created in the stopped state. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. To start the connection, issue the following command:

```
connection.start();
```

5. Create a session, for sending or receiving messages. The session provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers. The createQueueSession method is used on the connection to obtain a session. The method takes two parameters:
   - A boolean that determines whether or not the session is transacted.
   - A parameter that determines the acknowledge mode.

```
boolean transacted = false;
session = connection.createQueueSession( transacted,
                                         Session.AUTO_ACKNOWLEDGE);
```

In this example, the session is not transacted, and it should automatically acknowledge received messages. With these settings, a message is backed out only after a system error or if the application terminates unexpectedly.

The following points, as defined in the EJB specification, apply to these flags:

- The transacted flag passed on createQueueSession is ignored inside a global transaction and all work is performed as part of the transaction. Outside of a transaction the transacted flag is used and, if set to true, the application should use session.commit() and session.rollback() to control the completion of the work. In an EJB2.0 module, if the transacted flag is set to true and outside of an XA transaction, then the session is involved in the WebSphere local transaction and the unresolved action attribute of the method applies to the JMS work if it is not committed or rolled back by the application.
- Clients cannot use Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

6. Send a message.
   a. Create MessageProducers to create messages. For point-to-point messaging the MessageProducer is a QueueSender that is created by passing an output queue object (retrieved earlier) into the createSender method on the session. A QueueSender is normally created for a specific queue, so that all messages sent using that sender are sent to the same destination.

```
QueueSender queueSender = session.createSender(inQueue);
```

b. Create the message. Use the session to create an empty message and add the data passed.

   JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the vendor-specific class names for the message types, methods are provided on the Session object for message creation.

   In this example, a text message is created from the outString property:

   ```
   TextMessage outMessage = session.createTextMessage(outString);
   ```

c. Send the message.

   To send the message, the message is passed to the send method on the QueueSender:

   ```
   queueSender.send(outMessage);
   ```

7. Receive replies.

   a. Create a correlation ID to link the message sent with any replies. In this example, the client receives reply messages that are related to the message that it has sent, by using a provider-specific message ID in a JMSCorrelationID.

      ```
      messageID = outMessage.getJMSMessageID();
      ```

      The correlation ID is then used in a message selector, to select only messages that have that ID:

      ```
      String selector = "JMSCorrelationID = '"+messageID+"'";
      ```

   b. Create a MessageReceiver to receive messages. For point-to-point the MessageReceiver is a QueueReceiver that is created by passing an input queue object (retrieved earlier) and the message selector into the createReceiver method on the session.

      ```
      QueueReceiver queueReceiver = session.createReceiver(outQueue, selector);
      ```

   c. Retrieve the reply message. To retrieve a reply message, the receive method on the QueueReceiver is used:

      ```
      Message inMessage = queueReceiver.receive(2000);
      ```

      The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. If you omit this parameter, the call blocks indefinitely. If you do not want any delay, use the receiveNoWait()method. In this example, the receive call returns when the message arrives, or after 2000ms, whichever is sooner.

   d. Act on the message received. When a message is received, you can act on it as needed by the business logic of the client. Some general JMS actions are to check that the message is of the correct type and extract the content of the message. To extract the content from the body of the message, it is necessary to cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as TextMessage. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

      In this example, the instanceof operator is used to check that the message received is of the TextMessage type. The message content is then extracted by casting to the TextMessage subclass.

      ```
      if ( inMessage instanceof TextMessage )

   ...
           String replyString = ((TextMessage) inMessage).getText();
      ```

8. Closing down. If the application needs to create many short-lived JMS objects at the Session level or lower, it is important to close all the JMS resources used. To do this, you call the close() method on the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.

   ```
   queueReceiver.close();
   ...
   queueSender.close();
   ...
   session.close();
   ```

```
        session = null;
...
        connection.close();
        connection = null;
```

9. Publishing and subscribing to messages. To use JMS Publish/Subscribe support instead of point-to-point messaging, the general actions are the same; for example, to create a session and connection. The exceptions are that topic resources are used instead of queue resources (such as TopicPublisher instead of QueueSender), as shown in the following example to publish a message:

```
// Creating a TopicPublisher
        TopicPublisher pub = session.createPublisher(topic);
...
        pub.publish(outMessage);
...
        // Closing TopicPublisher
        pub.close();
```

10. Handling errors Any JMS runtime errors are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

    Unlike normal Java exceptions, a JMSException can contain another exception embedded in it. The implementation of JMSException does not include the embedded exception in the output of its toString()method. Therefore, you need to check explicitly for an embedded exception and print it out, as shown in the following example:

```
catch (JMSException je)
{
   System.out.println("JMS failed with "+je);
   Exception le = je.getLinkedException();
   if (le != null)
   {
       System.out.println("linked exception "+le);
   }
}
```

What to do next

After you have packaged your application, you can next deploy the application into WebSphere Application Server, as described in Deploying a J2EE application to use JMS.

Developing a JMS client

Use this task to develop a JMS client application to use messages to communicate with enterprise applications.

About this task

This topic gives an overview of the steps needed to develop a JMS client application. This topic only describes the JMS-related considerations; it does not describe general client programming, which you should already be familiar with. For detailed information about these steps, and for examples of developing JMS clients, see the Java Message Service Documentation and the WebSphere MQ *Using Java* book, SC34-5456.

A JMS client assumes that the JMS resources (such as a queue connection factory and queue destination) already exist. A client application can obtain suitable JMS resources either by JNDI lookup or programmatically without using JNDI.

For information about the Thin Client for JMS with WebSphere Application Server, which is an embeddable technology that provides JMS V1.1 connections to a WebSphere Application Server default messaging provider messaging engine, see Using JMS to connect to a WebSphere Application Server default messaging provider messaging engine.

For more information about developing client applications and configuring JMS resources for them, see Developing J2EE application client code and related tasks.

To use JMS, a typical JMS client program completes the following general steps. This example is based on the use of JNDI lookups to obtain JMS resources.

1. Import JMS packages. An enterprise application that uses JMS starts with a number of import statements for JMS; for example:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.jms.*;
```

2. Get an initial context.

```
   try     {
        ctx = new InitialContext(env);
...
```

3. Define the parameters that the client wants to use; for example, to identify the queue connection factory and to assemble a message to be sent.

```
public class JMSppSampleClient
{
  public static void main(String[] args)
  throws JMSException, Exception

  {
    String  messageID                = null;
    String  outString                = null;
    String  qcfName                  = "java:comp/env/jms/ConnectionFactory";
    String  qnameIn                  = "java:comp/env/jms/Q1";
    String  qnameOut                 = "java:comp/env/jms/Q2";
    boolean verbose                  = false;

    QueueSession           session    = null;
    QueueConnection        connection = null;
    Context                ctx        = null;

    QueueConnectionFactory qcf        = null;
    Queue                  inQueue    = null;
    Queue                  outQueue   = null;

...
```

4. Retrieve administered objects from the JNDI namespace. The InitialContext.lookup() method is used to retrieve administered objects (a queue connection factory and the queue destinations):

```
       qcf = (QueueConnectionFactory)ctx.lookup( qcfName );
...
       inQueue = (Queue)ctx.lookup( qnameIn );
       outQueue = (Queue)ctx.lookup( qnameOut );
...
```

5. Create a connection to the messaging service provider. The connection provides access to the underlying transport, and is used to create sessions. The createQueueConnection() method on the factory object is used to create the connection.

```
       connection = qcf.createQueueConnection();
```

   The JMS specification defines that connections should be created in the stopped state. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. To start the connection, issue the following command:

```
       connection.start();
```

6. Create a session, for sending and receiving messages. The session provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers. The createQueueSession method is used on the connection to obtain a session. The method takes two parameters:

- A boolean that determines whether or not the session is transacted.
- A parameter that determines the acknowledge mode.

```
boolean transacted = false;
session = connection.createQueueSession( transacted,
                                         Session.AUTO_ACKNOWLEDGE);
```

In this example, the session is not transacted, and it should automatically acknowledge received messages. With these settings, a message is backed out only after a system error or if the client application terminates unexpectedly.

7. Send the message.
   a. Create MessageProducers to create messages. For point-to-point the MessageProducer is a QueueSender that is created by passing an output queue object (retrieved earlier) into the createSender method on the session. A QueueSender is normally created for a specific queue, so that all messages sent using that sender are sent to the same destination.

      ```
      QueueSender queueSender = session.createSender(inQueue);
      ```

   b. Create the message. Use the session to create an empty message and add the data passed.

      JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the vendor-specific class names for the message types, methods are provided on the Session object for message creation.

      In this example, a text message is created from the outString property, which could be provided as an input parameter on invocation of the client program or constructed in some other way:

      ```
      TextMessage outMessage = session.createTextMessage(outString);
      ```

   c. Send the message.

      To send the message, the message is passed to the send method on the QueueSender:

      ```
      queueSender.send(outMessage);
      ```

8. Receive replies.
   a. Create a correlation ID to link the message sent with any replies. In this example, the client receives reply messages that are related to the message that it has sent, by using a provider-specific message ID in a JMSCorrelationID.

      ```
      messageID = outMessage.getJMSMessageID();
      ```

      The correlation ID is then used in a message selector, to select only messages that have that ID:

      ```
      String selector = "JMSCorrelationID = '"+messageID+"'";
      ```

   b. Create a MessageReceiver to receive messages. For point-to-point the MessageReceiver is a QueueReceiver that is created by passing an input queue object (retrieved earlier) and the message selector into the createReceiver method on the session.

      ```
      QueueReceiver queueReceiver = session.createReceiver(outQueue, selector);
      ```

   c. Retrieve the reply message. To retrieve a reply message, the receive method on the QueueReceiver is used:

      ```
      Message inMessage = queueReceiver.receive(2000);
      ```

      The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. If you omit this parameter, the call blocks indefinitely. If you do not want any delay, use the receiveNoWait()method. In this example, the receive call returns when the message arrives, or after 2000ms, whichever is sooner.

   d. Act on the message received. When a message is received, you can act on it as needed by the business logic of the client. Some general JMS actions are to check that the message is of the correct type and extract the content of the message. To extract the content from the body of the message, you need to cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as TextMessage. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

In this example, the instanceof operator is used to check that the message received is of the TextMessage type. The message content is then extracted by casting to the TextMessage subclass.

```
       if ( inMessage instanceof TextMessage )

...
            String replyString = ((TextMessage) inMessage).getText();
```

9. Closing down. If the application needs to create many short-lived JMS objects at the Session level or lower, it is important to close all the JMS resources used. To do this, you call the close() method on the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.

```
       queueReceiver.close();
...
       queueSender.close();
...
       session.close();
       session = null;
...
       connection.close();
       connection = null;
```

10. Publishing and subscribing messages. To use publish/subscribe support instead of point-to-point messaging, the general client actions are the same; for example, to create a session and connection. The exceptions are that topic resources are used instead of queue resources (such as TopicPublisher instead of QueueSender), as shown in the following example to publish a message:

```
// Creating a TopicPublisher
       TopicPublisher pub = session.createPublisher(topic);
...
       pub.publish(outMessage);
...
       // Closing TopicPublisher
       pub.close();
```

11. Handling errors Any JMS runtime errors are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

    Unlike normal Java exceptions, a JMSException can contain another exception embedded in it. The implementation of JMSException does not include the embedded exception in the output of its toString()method. Therefore, you need to check explicitly for an embedded exception and print it out, as shown in the following example:

```
catch (JMSException je)
{
   System.out.println("JMS failed with "+je);
   Exception le = je.getLinkedException();
   if (le != null)
   {
       System.out.println("linked exception "+le);
   }
}
```

What to do next

For information about running a client against a specific remote server: “Running application clients” on page 375.

Deploying an enterprise application to use JMS

This topic describes how to deploy an enterprise application to use JMS.

About this task

This task description assumes that you have an .EAR file, which contains an application enterprise bean with code for JMS, that can be deployed in WebSphere Application Server.

To deploy an enterprise application to use JMS, complete the following steps:

1. Configure the deployment attributes for the application, as described in Assembling applications.
2. Use the WebSphere Application Server administrative console to install the application.
   This stage is a standard WebSphere Application Server task, as described in Installing applications.

Programming to use message-driven beans

Applications can use message-driven beans (a type of enterprise bean defined in the EJB specification) as asynchronous message consumers.

About this task

A client sends messages to the destination (or *endpoint*) for which the message-driven bean is deployed as the message listener. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination. The message-driven bean implements some business logic to process incoming messages on the destination.

EJB 2.0 message-driven beans support only Java Message Service (JMS) messaging. EJB 2.1 message-driven beans can handle other messaging types in addition to JMS. The message-driven bean class must implement the message listener interface for the messaging type that the message-driven bean handles. For example, an EJB 2.1 message-driven bean class used for JMS messaging must implement the javax.jms.MessageListener interface.

You can use Rational Application Developer to develop applications that use message-driven beans. You can use the WebSphere Application Server runtime tools, like the administrative console, to deploy and administer applications that use message-driven beans.

If you are developing message-driven bean applications for use with WebSphere MQ as an external JMS provider, you must write them so that they consume each message before the maximum retry limit is reached. If you do not do this, the listener port stops when the maximum retry limit is reached for any given message. It then becomes necessary to manually remove the message from the WebSphere MQ queue.

For more information about implementing WebSphere enterprise applications that use message-drive beans, see the following topics:

- Designing an enterprise application to use a message-driven bean
- Developing an enterprise application to use a message-driven bean
- Deploying an enterprise application to use a message-driven bean

Designing an enterprise application to use message-driven beans

This topic describes things to consider when designing an enterprise application to use message-driven beans.

About this task

The considerations in this topic are based on a generic enterprise application that uses one message-driven bean to retrieve messages from a JMS queue destination and passes the messages on to another enterprise bean that implements the business logic.

To design an enterprise application to use message-driven beans, complete the following steps:

1. Identify the message listener interface for the message type that the message-driven beans is to handle. The message-driven bean class must implement this message listener interface. For example, an EJB 2.1 message-driven bean class used for JMS messaging must implement the javax.jms.MessageListener interface.
2. Identify the resources that the application is to use. This helps to identify the properties of resources that need to be used within the application and configured as application deployment descriptors or within WebSphere Application Server.

JMS resource type	Properties (for example)
JMS connection factory	**Name:** SamplePtoPQueueConnectionFactory **JNDI Name:** Sample/JMS/QCF
JMS destination	**Name:** Q1 **JNDI Name:** Sample/JMS/Q1
J2C activation specification properties	**Name:** MyMDBsActivationSpec **JNDI Name:** eis/MyMDBsActivationSpec **Destination JNDI Name:** MyQueue **Destination type:** javax.jms.Queue
Message-driven bean (deployment properties)	**Name:** JMSppSampleMDBBean **Transaction type:** Container **Message selector:** JMSType='car' **Acknowledge mode:** Dups OK Acknowledge **Destination type:** javax.jms.Queue **ActivationSpec JNDI name:** MyMDBsActivationSpec
Business logic bean	**Name:** MyLogicBean

   Ensure that you use consistent values where needed; for example, the JNDI name for the J2C activation specification must be the same in both the activation specification and the Message-driven bean's deployment properties.
3. Separation of business logic. You are recommended to develop a message-driven bean to delegate the business processing of incoming messages to another enterprise bean. This provides clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client.
4. Security considerations. Messages arriving at a destination being processed by a listener have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component. For more information about EJB security, see EJB component security.
5. Discarding of best effort nonpersistent messages with the default messaging provider. If you have a non-transactional message-driven bean, the system either deletes the message when the bean starts, or when the bean completes. If the bean generates an exception, and therefore does not complete:
   - If the system is configured to delete the message when the bean completes, then the message is despatched to a new instance of the bean, so the message has another opportunity to be processed.
   - If the system is configured to delete the message when the bean starts, the message is lost.

   The message is deleted when the bean starts if the quality of service is set to best effort nonpersistent. For all other qualities of service, the message is deleted when the bean completes.

Developing an enterprise application to use message-driven beans

Use this task to develop an enterprise application to use a message-driven bean. The message-driven bean is invoked by a J2C activation specification or a JMS listener when a message arrives on the input destination that the listener is monitoring.

About this task

You are recommended to develop the message-driven bean to delegate the business processing of incoming messages to another enterprise bean, to provide clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client. Responses can be handled by another enterprise bean acting as a sender bean, or handled in the message-driven bean.

You develop an enterprise application to use a message-driven bean like any other enterprise bean, except that a message-driven bean does not have a home interface or a remote interface.

For more information about writing the message-driven bean class, see *Creating a message-driven bean* in the Rational Application Developer help bookshelf.

To develop an enterprise application to use a message-driven bean, complete the following steps:

1. Create the Enterprise Application project.
2. Create the message-driven bean class.

   You can use the New Enterprise Bean wizard of Rational Application Developer to create an enterprise bean with a bean type of Message-driven bean. The wizard creates appropriate methods for the type of bean.

   By convention, the message bean class is named *name*Bean, where *name* is the name you assign to the message bean; for example:

   ```
   public class MyJMSppMDBBean implements MessageDrivenBean, javax.jms.MessageListener
   ```

   All message-driven beans must implement the MessageDrivenBean interface. For JMS messaging, a message-driven bean must also implement the message listener interface, javax.jms.MessageListener. Other Java EE Connector Architecture (JCA)-compliant Resource Adapters may provide their own message listener interface that needs to be implemented.

   A message-driven bean can be registered with the EJB timer service for time-based event notifications if it also implements the javax.ejb.TimedObject interface and the timer callback method void ejbTimeout(Timer). At the scheduled time, the container invokes the message-driven bean's ejbTimeout method.

   The message-driven bean class must define and implement the following methods:

   - onMessage(message), which must meet the following requirements:
     - The method must have a single argument of type `javax.jms.Message`.
     - The throws clause must *not* define any application exceptions.
     - If the message-driven bean is configured to use bean-managed transactions, it must call the javax.transaction.UserTransaction interface to scope the transactions. Because these calls occur inside the onMessage() method, the transaction scope does not include the initial message receipt. For more information, see "Message-driven beans - transaction support" on page 1168.

     To handle the message within the onMessage() method (for example, to pass the message on to another enterprise bean), you use standard JMS. (This is known as bean-managed messaging.)

     If you are using a JCA-compliant Resource Adapter with a different message listener interface, another method besides onMessage() may be needed. For information about the message listener interface needed, see the documentation that was provided with your JCA Resource Adapter.
   - ejbCreate()

     You must define and implement an ejbCreate method for each way in which you want a new instance of an enterprise bean to be created.
   - ejbRemove()

     This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface from the javax.ejb.EJBHome interface. This method must contain any code that you want to execute before an enterprise bean instance is removed from the container (and the associated data is removed from the data source).
   - ejbTimeout(Timer)

This method is needed only to support notifications from the timer service, and contains the business logic that handles time events received.

For example, the following code extract shows how to access the text and the JMS MessageID, from a JMS message of type TextMessage:

The result of this step is a message-driven bean that can be assembled into an EAR file for

```
public void onMessage(javax.jms.Message msg)
{
        String text      = null;
        String messageID = null;

        try
        {
                text = ((TextMessage)msg).getText();

                System.out.println("senderBean.onMessage(), msg text2: "+text);

                //
                // store the message id to use as the Correlator value
                //
                messageID = msg.getJMSMessageID();

                // Call a private method to put the message onto another queue
                putMessage(messageID, text);
        }
        catch  (Exception err)
        {
                err.printStackTrace();
        }
        return;
}
```

Figure 9. Code example: The onMessage() method of a message bean. This figure shows a code extract for a basic onMessage() method of a sample message-driven bean. The method unpacks the incoming text message to extract the text and message identifier and calls a private putMessage method (defined within the same message bean class) to put the message onto another queue.

deployment.

3. Optional: Use the EJB deployment descriptor editor to review and, if needed, change the deployment properties. You can use the EJB deployment descriptor editor to review deployment properties that you specified on the EJB Creation Wizard (like Transaction type and Message selector) and other default deployment properties.

   If needed, you can override the values of these properties later, after the enterprise application has been exported into an EAR file for deployment.

   a. In the property pane, select the Beans tab.

   b. Specify general deployment properties.

      Transaction type

      Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean.

      Bean The message bean manages its own transactions

      Container

      The container manages transactions on behalf of the bean

   c. Specify advanced deployment properties.

      Under Activation Configuration, review the following properties:

      Acknowledge mode

      How the session acknowledges any messages it receives.

      This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).

Auto Acknowledge
The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

Dups OK Acknowledge
The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

Note: The acknowledgement is sent when the message is deleted. If you have a non-transactional message-driven bean, the system either deletes the message when the bean starts, or when the bean completes. If the bean generates an exception, and therefore does not complete:

- If the system is configured to delete the message when the bean completes, then the message is despatched to a new instance of the bean, so the message has another opportunity to be processed.
- If the system is configured to delete the message when the bean starts, the message is lost.

The message is deleted when the bean starts if the quality of service is set to best effort nonpersistent. For all other qualities of service, the message is deleted when the bean completes.

Destination type
Whether the message bean uses a queue or topic destination.

Queue
The message bean uses a queue destination.

Topic The message bean uses a topic destination.

Durability
Whether a JMS topic subscription is durable or non-durable.

Durable
A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.

Nondurable
Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

Message selector
The JMS message selector to be used to determine which messages the message bean receives; for example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

For more details about these properties, see “Message-driven bean deployment descriptor properties.”

d. Specify bindings deployment properties.

Under WebSphere Bindings, select the JCA Adapter option then specify the bindings deployment properties:

ActivationSpec JNDI name
Type the JNDI name of the J2C activation specification that is to be used to deploy this message-driven bean. This name must match the name of a J2C activation specification that you define to WebSphere Application Server.

ActivationSpec Authorization Alias
The name of a J2C authentication alias used for authentication of connections to the JCA resource adapter. A J2C authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.

Destination JNDI name
Type the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI name space.

4. Assemble and package the application for deployment.

Results

The result of this task is an EAR file, containing the message-driven bean, for the enterprise application that can be deployed in WebSphere Application Server.

What to do next

After you have developed an enterprise application to use message-driven beans, configure and deploy the application; for example, define J2C activation specifications for the message-driven beans and, optionally, change the deployment descriptor attributes for the application. For more information about configuring and deploying an application that uses message-driven beans, see Deploying an enterprise application to use message-driven beans

Message-driven bean deployment descriptor properties:

Here are the deployment descriptor properties that are used for message-driven beans.

You can configure JMX extension MBean providers to be used to extend the existing WebSphere managed resources in the core administrative system. Each MBean provider is a library containing an implementation of a JMX MBean and its MBean XML Descriptor file.

To view this administrative console page, click **Servers > Application Servers >** *server_name* **> Administration > Administration Services > Extension MBean Providers**

Transaction type:

Whether the message-driven bean manages its own transactions or the container manages transactions on behalf of the bean.

Bean The message-driven bean manages its own transactions

Container
The container manages transactions on behalf of the bean

Message selector:

The JMS message selector to be used to determine which messages the message-driven bean receives.

For example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

Acknowledge mode:

How the session acknowledges any messages it receives.

This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).

Auto Acknowledge
: The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

Dups OK Acknowledge
: The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

Destination type:

Whether the message-driven bean uses a queue or topic destination.

Queue
: The message-driven bean uses a queue destination.

Topic The message-driven bean uses a topic destination.

Subscription durability:

Whether a JMS topic subscription is durable or nondurable.

Durable
: A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.

Nondurable
: Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

  A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

ActivationSpec name:

Type the JNDI name of the J2C activation specification that is to be used to deploy this message-driven bean. This name must match the name of an activation specification that you define to WebSphere Application Server.h

Extension MBean Provider settings:

Use this page to view and change the configuration for a JMX extension MBean provider.

You can configure a library containing an implementation of a JMX MBean, and its MBean XML Descriptor file, to be used to extend the existing WebSphere managed resources in the core administrative system

To view this administrative console page, click **Servers > Application Servers >** *server_name* **> Administration > Administration Services > Extension MBean Providers >** *provider_library_name*

Name:

The name used to identify the Extension MBean provider library.

Data type String

Classpath:

The path to the Java archive (JAR) file that contains the Extension MBean provider library. This class path is automatically added to the Application Server class path. The class loader needs this information to load and parse the Extension MBean XML Descriptor file.

Data type String

Description:

An arbitrary descriptive text for the Extension MBean Provider configuration. Use this field for any text that helps identify or differentiate the provider configuration.

Data type String

Deploying an enterprise application to use message-driven beans against JCA 1.5-compliant resources

Use this task to deploy an enterprise application to use EJB 2.1 or EJB 2.0 message-driven beans for use with a Java EE Connector Architecture (JCA) 1.5-compliant resource adapter.

About this task

Message-driven beans can be configured as listeners on a JCA 1.5 resource adapter, such as the default messaging provider in WebSphere Application Server.

You deploy EJB 2.1 message-driven beans against JCA 1.5-compliant resources, and configure the resources as deployment descriptor properties. Although you can continue to deploy an EJB 2.0 message-driven bean against a listener port (as in WebSphere Application Server Version 5), you are recommended to deploy EJB 2.0 message-driven beans against JCA 1.5-compliant resources and to upgrade them to be EJB 2.1 message-driven beans.

This task description assumes that you have an .EAR file, which contains an application enterprise bean with code for message-driven beans, that can be deployed to use the default messaging provider in WebSphere Application Server.

To deploy an enterprise application to use message-driven beans against JCA 1.5-compliant resources, complete the following steps:

1. For each message-driven bean in the application, configure a J2C activation specification.

For example, for a message-driven bean to listen on a JMS destination of the default messaging provider, see Configuring a JMS activation specification for MDBs used by the default messaging provider.

2. For each message-driven bean in the application, configure the J2C deployment attributes, as described in Configuring deployment attributes.
3. Use the WebSphere Application Server administrative console to install the application.

   This stage is a standard WebSphere Application Server task, as described in Installing a new application.

Configuring deployment properties for a JCA 1.5-compliant message-driven bean:

Use this task to configure the message-driven bean deployment properties for a J2EE Connector Architecture (JCA) 1.5-compliant enterprise bean, to override the deployment properties defined within the application EAR file.

About this task

You can configure the deployment attributes of an application by using an assembly tool such as Rational Application Developer.

This topic describes the use of an assembly tool to configure the deployment attributes of an application that is to use message-driven beans against JCA 1.5-compliant resources. If you want to configure the deployment attributes for a message-driven bean against a listener port, see “Configuring deployment attributes for an EJB 2.0 message-driven bean against a listener port” on page 1267.

This task description assumes that you have an EAR file, which contains an application enterprise bean developed as a message-driven bean, that can be deployed in WebSphere Application Server. For more details about assembling applications, see assembling applications.

1. Start an assembly tool.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, use the import wizard to import the EAR file into the assembly tool. To start the import wizard:
   a. Click **File-> Import-> EAR file**
   b. Click **Next**, then select the EAR file.
   c. Click **Finish**
3. In the Java EE Hierarchy view, right-click the EJB module for the message-driven bean, then click **Open With > Deployment Descriptor Editor**. A property dialog notebook for the message-driven bean is displayed in the property pane.
4. Use the EJB deployment descriptor editor to review and, if needed, change the deployment properties.
   a. In the property pane, select the Beans tab.
   b. Under Activation Configuration, review the following properties:
      Acknowledge mode
      How the session acknowledges any messages it receives.

      This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).
      Auto Acknowledge
      The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.
      Dups OK Acknowledge
      The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

Destination type

Whether the message bean uses a queue or topic destination.

Queue

The message bean uses a queue destination.

Topic The message bean uses a topic destination.

Durability

Whether a JMS topic subscription is durable or non-durable.

Durable

A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.

Nondurable

Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

Message selector

The JMS message selector to be used to determine which messages the message bean receives; for example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

For more details about these properties, see "Message-driven bean deployment descriptor properties" on page 1259.

c. Under WebSphere Bindings, select the JCA Adapter option then specify the bindings deployment properties:

ActivationSpec JNDI name

Type the JNDI name of the J2C activation specification that is to be used to deploy this message-driven bean. This name must match the name of a J2C activation specification that you define to WebSphere Application Server.

ActivationSpec Authorization Alias

The name of a J2C authentication alias used for authentication of connections to the JCA resource adapter. A J2C authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.

Destination JNDI name

Type the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI name space.

5. Save your changes to the deployment descriptor.
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.
6. Verify the archive files with an assembly tool.
7. From the popup menu of the project, click **Deploy** to generate EJB deployment code.

8. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard. Note that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

   Note: Use **Run On Server** for unit testing only. When an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console as described in Deploying and managing applications.

Configuring security for EJB 2.1 message-driven beans:

Use this task to configure resource security and security permissions for Enterprise JavaBeans (EJB) Version 2.1 message-driven beans.

About this task

The association between connection factories, destinations, and message-driven beans is provided by listener ports. A listener port allows a deployed message-driven bean associated with the port to retrieve messages from the associated destination. You create listener ports by specifying their administrative name, the connection factory JNDI name, and the destination name (other optional properties are also configurable). Listener ports provide simplified administration of the associations between connection factories, destinations and message-driven beans, and are managed by a listener manager. The listener manager is provided by the message listener service to control and monitor the JMS listeners that are monitoring JMS destinations on behalf of deployed message-driven beans. For more information about listener ports, see Message-driven beans - listener port components

Messages handled by message-driven beans have no client credentials associated with them. The messages are anonymous.

To call secure enterprise beans from a message-driven bean, the message-driven bean needs to be configured with a RunAs Identity deployment descriptor. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component.

For more information about EJB security, see EJB component security. For more information about configuring security for your application, see Assembling secured applications.

Connections used by message-driven beans can benefit from the added security of using J2C container-managed authentication. To enable the use of J2C container authentication aliases and mapping, define an authentication alias on the J2C activation specification that the message-driven bean is configured with. If defined, the message-driven bean uses the authentication alias for its JMSConnection security credentials instead of any application-managed alias.

To set the authentication alias, you can use the administrative console to complete the following steps. This task description assumes that you have already created an activation specification. If you want to create a new activation specification, see the related tasks.

- For a message-driven bean listening on a JMS destination of the default messaging provider, set the authentication alias on a JMS activation specification.
  1. To display the JMS activation specification settings, click **Resources** → **JMS** → **JMS providers**
  2. In the content pane, click the name of a default messaging provider.

3. In the content pane, under Additional Resources, click **Activation specifications**.
4. If you have already created a JMS activation specification, click its name in the list displayed. Otherwise, click **New** to create a new JMS activation specification.
5. Set the **Authentication alias** property.
6. Click **OK**
7. Save your changes to the master configuration.

- For a message-driven bean listening on a destination (or endpoint) of another Java EE Connector Architecture (JCA) provider, set the authentication alias on a J2C activation specification.
  1. To display the J2C activation specification settings, click **Resources** → **Resource Adapters** → **adapter_name** → **J2C Activation specifications** → **activation specification_name**
  2. Set the **Authentication alias** property.
  3. Click **OK**
  4. Save your changes to the master configuration.

Throttling inbound message flow for JCA 1.5 message-driven beans:

This topic describes how to throttle message delivery for message-driven beans (MDB) which are deployed as message endpoints for Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) Version 1.5 inbound resource adapters.

About this task

For installations that use resource adapters that implement the Java EE Connector Architecture (JCA) Version 1.5 message delivery support, the WebSphere Application Server provides message throttling support to control the delivery of messages to endpoint message-driven beans (MDB). You can use this support to avoid overloading the server with a flood of inbound messages, except in the following two cases:

- The default messaging provider (the SIB JMS Resource Adapter) uses a special type of message throttling. You should not use the JCA 1.5 throttling of messages, described in this topic, for message-driven beans that you have deployed as JCA 1.5 resources on the default messaging provider. You can leave the message-driven bean pools to the default size of 500. For more information about the message throttling support of the default messaging provider (the SIB JMS Resource Adapter), see the related tasks.
- If you want to throttle message delivery for a message-driven bean deployed on a JMS provider that does not have a JCA 1.5 resource adapter (such as the V5 Default Messaging and WebSphere MQ) you can configure message throttling support as described in the related tasks.

Message delivery is throttled on an message-driven bean basis by limiting the maximum number of endpoint instances that can be created by the adapter that the MDB is bound to. When the adapter attempts to create an endpoint instance, a proxy for the MDB instance is created and returned as allowed by the JCA 1.5 architecture. There is a one-to-one correspondence between proxies and MDB instances, and like the MDB instances, the proxies are pooled based on the minimum and maximum pool size values associated with the message-driven bean. Throttling is performed through the management of the proxy pool.

At the time the adapter attempts to create an endpoint, if the number of endpoint proxies currently created is equal to the maximum size of the pool, adapter *createEndPoint* processing returns an Unavailable Exception. When this happens, the adapter is not allowed to issue any more *createEndPoint()* requests until it has released at least one endpoint back to the server for reuse. Installations can thus control the throttling of message delivery to a JCA 1.5 MDB based on the setting of the maximum size of the pool associated with each JCA 1.5 message-driven bean.

You can specify the poolsize by using the *com.ibm.websphere.ejbcontainer.poolsize jvm* System property to define the minimum and maximum poolsize of stateless, message-driven, and entity beans. In the case of an message-driven bean that supports JCA 1.5, the maximum poolsize value specified limits how many message endpoint instances can be created for that message-driven bean. For example, if the installation sets the maximum size of a JCA 1.5 MDB pool to 5, then at most 5 messages can be concurrently delivered to 5 instances of the message-driven bean. This property can be specified using command-line scripting (see EJB container system properties) or by specifying it under the Administrative Console as an environmental variable.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. Under Server Infrastructure, expand **Java and Process Management**.
6. Select **Process Definition**.
7. Select **Servant**.
8. Under Additional Properties, select **Java Virtual Machine**.
9. Under Additional Properties, select **Custom Properties**.
10. Select **New**. A panel with three General Properties fields appears. This is where you set the property.
11. In the Name field, enter **com.ibm.websphere.ejbcontainer.poolsize**.
12. To fill in the Value field, refer to EJB container system properties for possible values.
13. After defining the value of the property, select **OK**. You are now prompted to save the changes you have just made.
14. Select **Save**.

Using message-driven beans with JCA version 1.5 resource adapters:

You must redefine the servant region for any message-driven beans that are connected to Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) Version 1.5 resource adapters that support inbound message processing.

About this task

If the application server is configured with one or more message-driven beans (MDBs) that are bound to a J2EE Connector Architecture (JCA) Version 1.5 resource adapter, and the resource adapter supports inbound message processing, the adapter must be defined to run only a maximum of 1 servant region. If you use the IMS Connect for Java (IC4J) resource adapter, however, you can configure this adapter to run multiple servant regions.

A Servant can be configured to a maximum of 1 servant region by using the administrative console.

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. Under Server Infrastructure, expand **Java and Process Management**.
6. Select **Process Definition**.
7. Select**Servant**.
8. Under Additional Properties, select **Environment Entries**. This causes a panel with the heading Application servers > *server* > Process Definition > Servant > Custom Properties to appear.
9. Select **New**. A panel with three General Properties fields appears. This is where you create the environment variable for controlling the number of servant regions.

10. In the Name field, enter **wlm_maximumSRcount**.
11. In the Value field, enter **1**.
12. Select **Apply**.

Deploying an enterprise application to use EJB 2.0 message-driven beans with listener ports

Use this task to deploy an enterprise application to use EJB 2.0 message-driven beans with listener ports.

About this task

Although you can continue to deploy an EJB 2.0 message-driven bean against a listener port (as in WebSphere Application Server Version 5), you are recommended to deploy such beans as J2EE Connector Architecture (JCA) 1.5-compliant resources and to upgrade them to be EJB 2.1 message-driven beans.

This task description assumes that you have an .EAR file, which contains an application enterprise bean with code for EJB 2.0 message-driven beans, that can be deployed in WebSphere Application Server.

Note: From WebSphere Application Server Version 7 listener ports are deprecated. For information about the facilities available to aid migration of configuration information from a listener port to an activation specification for use with the Websphere MQ messaging provider, refer to related tasks.

To deploy an enterprise application to use EJB 2.0 message-driven beans with listener ports, complete the following steps:

1. Use the WebSphere Application Server administrative console to define the listener ports for the application, as described in Adding a new listener port.
2. For each message-driven bean in the application, configure the deployment attributes to match the listener port definitions, as described in Configuring deployment attributes.
3. Use the WebSphere Application Server administrative console to install the application.

   This stage is a standard WebSphere Application Server task, as described in Installing a new application.

   When you install the application, you are prompted to specify the name of the listener port that the application is to use for late responses. Select the listener port, then click **OK**.

Configuring deployment attributes for an EJB 2.0 message-driven bean against a listener port:

Use this task to configure the message-driven beans deployment attributes for an enterprise bean, to override the deployment attributes defined within the application EAR file.

About this task

You can configure the deployment attributes of an application by using an assembly tool such as Rational Application Developer.

This topic describes the use of the Rational Application Developer to configure the deployment attributes of an application. This task description assumes that you have an EAR file, which contains an application enterprise bean developed as a message-driven bean, that can be deployed in WebSphere Application Server. For more details about assembling applications, see Assembling applications.

To configure the message-driven beans deployment attributes for an enterprise bean, use the assembly tool to configure the deployment attributes of the application to match the listener port definitions:

1. Start an assembly tool.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, use the import wizard to import the EAR file into the assembly tool. To start the import wizard:

a. Click **File-> Import-> EAR file**
b. Click **Next**, then select the EAR file.
c. Click **Finish**

3. In the Java EE Hierarchy view, right-click the EJB module for the message-driven bean then click **Open With > Deployment Descriptor Editor**. A property dialog notebook for the message-driven bean is displayed in the property pane.
4. Specify general deployment properties.
   a. In the property pane, select the Beans tab.
   b. Specify the following properties:
      Transaction type
      Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean.
      Bean The message bean manages its own transactions
      Container
      The container manages transactions on behalf of the bean
5. Specify advanced deployment properties.
   a. Under Activation Configuration, review the following properties:
      Acknowledge mode
      How the session acknowledges any messages it receives.

      This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).
      Auto Acknowledge
      The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.
      Dups OK Acknowledge
      The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

      As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.
      Destination type
      Whether the message bean uses a queue or topic destination.
      Queue
      The message bean uses a queue destination.
      Topic The message bean uses a topic destination.
      Durability
      Whether a JMS topic subscription is durable or non-durable.
      Durable
      A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.
      Nondurable
      Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

Message selector

The JMS message selector to be used to determine which messages the message bean receives; for example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

For more details about these properties, see “Message-driven bean deployment descriptor properties” on page 1259.

6. Specify bindings deployment properties.
   a. Under WebSphere Bindings, specify the following property:
      Listener port name
      Type the name of the listener port for this message-driven bean.
7. Save your changes to the deployment descriptor.
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.
8. Verify the archive files.
9. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
10. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

    Note: **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only. You cannot deploy remotely to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Note: Use **Run on Server** for unit testing only. The assembly tool controls the WebSphere Application Server installation. When an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console as described in Deploying and managing applications.

JMS interfaces

WebSphere Application Server supports applications that use JMS 1.1 domain-independent interfaces (referred to as “common interfaces” in the JMS specification) and JMS 1.0.2 domain-specific interfaces.

With JMS 1.1, the preferred approach for implementing applications is to use common interfaces because they provide a simpler programming model than domain-specific interfaces. Also, applications can create both queues and topics in the same session and coordinate their use in the same transaction. Common interfaces are parents of domain-specific interfaces.

Note: Domain-specific interfaces (provided for JMS 1.0.2 in WebSphere Application Server Version 5) are supported only to provide backward compatibility for applications that have already been implemented to use those interfaces.

Common interfaces	point-point interfaces	publish/subscribe interfaces
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

For more information about JMS interfaces, see the JMS documentation at http://java.sun.com/products/jms/docs.html.

JMS and WebSphere MQ message structures

You need to consider how the JMS message structure is mapped onto a WebSphere MQ message if you want to transmit messages between JMS applications and traditional WebSphere MQ applications. This includes scenarios where you want to use WebSphere MQ to manipulate messages transmitted between two JMS applications; for example, using WebSphere MQ as a message broker.

By default, JMS messages held on WebSphere MQ queues use an MQRFH2 header to hold some of the JMS message header information. Many traditional WebSphere MQ applications cannot process messages with these headers, and require their own characteristic headers, for example the MQWIH for WebSphere MQ Workflow applications. For more information about how the JMS message structure is mapped onto a WebSphere MQ message, see the section ″Mapping JMS to a native WebSphere MQ application″ in the chapter ″JMS Messages″ of the WebSphere MQ Using Java book.

Chapter 10. Mail, URLs, and other J2EE resources

Using mail

You can enable your Java Platform, Enterprise Edition (Java EE) applications to use mail resources with the JavaMail API.

Before you begin

Using the JavaMail API, a code segment can be embedded in any Java EE application component, such as an EJB or a servlet, allowing the application to send a message and save a copy of the mail to the Sent folder.

The following is a code sample that you would embed in a Java EE application:

```
javax.naming.InitialContext ctx = new javax.naming.InitialContext();

   javax.mail.Session mail_session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession3");
   MimeMessage msg = new MimeMessage(mail_session);

   msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse("bob@coldmail.net"));

   msg.setFrom(new InternetAddress("alice@mail.eedge.com"));

   msg.setSubject("Important message from eEdge.com");

   msg.setText(msg_text);

   Transport.send(msg);


   Store store = mail_session.getStore();

   store.connect();

   Folder f = store.getFolder("Sent");

   if (!f.exists()) f.create(Folder.HOLDS_MESSAGES);

   f.appendMessages(new Message[] {msg});
```

About this task

Java EE applications can use JavaMail APIs by looking up references to logically named mail connection factories through the `java:comp/env/mail` subcontext that is declared in the application deployment descriptor and mapped to installation specific mail session resources. As in the case of other Java EE resources, this can be done in order to eliminate the need for the application to hard code references to external resources.

1. Locate a resource through Java Naming and Directory Interface (JNDI). The Java EE specification considers a mail session instance as a resource, or a factory from which mail transport and store connections can be obtained. Do not hard code mail sessions (namely, fill up a Properties object, then use it to create a javax.mail.Session object). Instead, you must follow the Java EE programming model of configuring resources through the system facilities and then locating them through JNDI lookups.

   In the previous sample code, the line `javax.mail.Session mail_session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession3");` is an example of not hard coding a mail session and using a resource name located through JNDI. You can consider the lookup name, `mail/MailSession3`, as an indirect reference to the real resource.

2. Define resource references while assembling your application. You must define a resource reference for the mail resource in the deployment descriptor of the component, because a mail session is referenced in the JNDI lookup. Typically, you can use an assembly tool that shipped with the application server.

   When you create this reference, be sure that the name of the reference matches the name used in the code. For example, the previous code uses `java:comp/env/mail/MailSession3` in its lookup. Therefore the name of this reference must be mail/Session3, and the type of the resource must be javax.mail.Session. After configuration, the deployment descriptor contains the following entry for the mail resource reference:

```
<resource-reference>
   <description>description</description>
   <res-ref-name>mail/MailSession3</res-ref-name>
   <res-type>javax.mail.Session</res-type>
   <res-auth>Container</res-auth>
</resource-reference>
```

3. Configure mail providers and sessions. The sample code references a mail resource, the deployment descriptor declares the reference, but the resource itself does not exist yet. Now you need to configure the mail resource that is referenced by your application component. Notice that the mail session you configure must have both its transport and mail access portions defined; the former required because the code is sending a message, the latter because it also saves a copy to the local mail store. When you configure the mail session, you need to specify a JNDI name. This is an important name for installing your application and linking up the resource references in your application with the real resources that you configure.
4. Install your application. You can install your application using either the administrative console or the scripting tool. During installation, the application server inspects all resource references and requires you to supply a JNDI name for each of them. This is not an arbitrary JNDI name, but the JNDI name given to a particular, configured resource that is the target of the reference.
5. Manage existing mail providers and sessions. You can update and remove mail providers and sessions.

   To update mail providers and sessions:

   a. Open the administrative console.
   b. Click **Resources** → **Mail** in the console navigation tree.
   c. Select the appropriate Java Mail resource to modify by clicking either **Mail Providers** or **Mail Sessions**.
   d. Select the specific resource to modify. To remove a mail provider or mail session, select the check box next to the appropriate resource and click **Delete**.
   e. Click **Apply** or **OK**.
   f. Save the configuration.
6. Optional: Debug a mail session.

What to do next

If your application has a client, you can update mail providers and mail sessions using the Application Client Resource Configuration Tool (ACRCT).

JavaMail API

The JavaMail APIs provide a framework that is platform and protocol independent for building mail client applications that are based on Java. The JavaMail APIs are generic for sending and reading mail. They require service providers, known in the application server as protocol providers, to interact with mail servers that run on pertaining protocols. For example, Simple Mail Transfer Protocol (SMTP) is a popular transport protocol for sending mail. Mail applications can connect to an SMTP server and send mail through it by using this SMTP protocol provider.

The application server supports the JavaMail API, Version 1.4. In the application server, the JavaMail API is supported in all Web application components, namely:

- servlets
- JavaServer Pages (JSP) files
- enterprise beans
- application clients

In addition to service providers, the JavaMail API requires the JavaBeans Application Framework (JAF) to handle mail content that is not plain text, including Multipurpose Internet Mail Extensions (MIME), URL pages, and file attachments.

The JavaMail APIs, the JAF, the service providers, and the protocols are shipped as part of the application server. The API and related specifications are repackaged from materials that are licensed to Sun Microsystems.

Note: If you are using Java 5 to run your code, you will need the JAF 1.1 package. For Java 6 and later, the JAF package is part of the run time environment.

Mail providers and mail sessions

A mail service provider is a driver that supports mail interaction with mail servers that use a particular mail protocol. The application server includes service providers, which are also known as protocol providers, for mail protocols.

A mail provider encapsulates a collection of protocol providers. For example, the application server has a built-in mail provider that encompasses the most common protocol providers. These protocol providers are installed as the default and suffice for most applications. If you have a particular application that requires custom protocol providers, follow the steps that are outlined in the chapter on mail sessions in the JavaMail API Design Specification to install your own protocol providers.

Mail sessions are represented by the javax.mail.Session class. A mail session object authenticates users and controls access to messaging systems.

To create mail applications that are platform independent, use a resource factory reference to create a mail session. A resource factory is an object that provides access to resources in the deployed environment of a program. Resource factories use the naming conventions that are defined by the Java Naming and Directory Interface (JNDI).

Note: Ensure that every mail session is defined under a parent mail provider. Select a mail provider first and then create your new mail session.

JavaMail security permissions best practices

In many of its activities, the JavaMail API needs to access certain configuration files. The JavaMail and JavaBeans Activation Framework binary packages themselves already contain the necessary configuration files. However, the JavaMail API allows the user to define user-specific and installation-specific configuration files to meet special requirements.

The two locations where you can place these configuration files are the <user.home> and <java.home>/lib directories. For example, if the JavaMail API needs to access a file named mailcap when it sends a message, the API:

1. Tries to access <user.home>/mailcap.
2. If the first attempt fails due to a lack of security permission or a nonexistent file, the API searches in <java.home>/lib/mailcap.

3. If the second attempt also fails, the API searches in the META-INF/mailcap location in the class path. This location actually leads to the configuration files contained in the mail-impl.jar and activation-impl.jar files.

Application Server uses JavaMail API configuration files that are contained in the mail-impl.jar and activation-impl.jar files, and there are no mail configuration files in <user.home> and <java.home>/lib directories. To ensure proper functioning of the JavaMail API, Application Server grants *file read* permission for both the mail-impl.jar and activation-impl.jar files to all of the installed applications.

JavaMail code attempts to access configuration files at <user.home> and <java.home>/lib, which can cause an access control exception to be thrown, since the default configuration does not grant file read permission for those two locations by default. This activity does not affect the proper functioning of the JavaMail API, but you might see a large amount of mail-related security exceptions reported in the system log, and these errors could overshadow harmful errors for which you are looking. This is a sample of the security message, SECJ0314W:

```
[02/31/08 12:55:38:188 PDT] 00000058 SecurityManag W   SECJ0314W: Current Java 2 Security policy reported a
potential violation of Java 2 Security Permission.
Please refer to Problem Determination Guide for further information.

Permission:

      D:\o063919\java\jre\lib\javamail.providers : access denied (java.io.FilePermission
D:\o063919\java\jre\lib\javamail.providers read)


Code:

     com.ibm.ws.mail.SessionFactory  in  {file:/D:/o063919/lib/runtime.jar}



Stack Trace:

java.security.AccessControlException: access denied (java.io.FilePermission D:\o063919\java\jre\lib\javamail.providers read)
 at java.security.AccessControlContext.checkPermission(AccessControlContext.java(Compiled Code))
 at java.security.AccessController.checkPermission(AccessController.java(Compiled Code))
 at java.lang.SecurityManager.checkPermission(SecurityManager.java(Compiled Code))
 at com.ibm.ws.security.core.SecurityManager.checkPermission(SecurityManager.java(Compiled Code))
 at java.lang.SecurityManager.checkRead(SecurityManager.java(Compiled Code))
 at java.io.FileInputStream.<init>(FileInputStream.java(Compiled Code))
 at java.io.FileInputStream.<init>(FileInputStream.java:89)
 at javax.mail.Session.loadFile(Session.java:1004)
 at javax.mail.Session.loadProviders(Session.java:861)
 at javax.mail.Session.<init>(Session.java:191)
 at javax.mail.Session.getInstance(Session.java:213)
 at com.ibm.ws.mail.SessionFactory.getObjectInstance(SessionFactory.java:67)
 at javax.naming.spi.NamingManager.getObjectInstance(NamingManager.java:314)
 at com.ibm.ws.naming.util.Helpers.processSerializedObjectForLookupExt(Helpers.java:894)
 at com.ibm.ws.naming.util.Helpers.processSerializedObjectForLookup(Helpers.java:701)
 at com.ibm.ws.naming.jndicos.CNContextImpl.processResolveResults(CNContextImpl.java:1937)
 at com.ibm.ws.naming.jndicos.CNContextImpl.doLookup(CNContextImpl.java:1792)
 at com.ibm.ws.naming.jndicos.CNContextImpl.doLookup(CNContextImpl.java:1707)
 at com.ibm.ws.naming.jndicos.CNContextImpl.lookupExt(CNContextImpl.java:1412)
 at com.ibm.ws.naming.jndicos.CNContextImpl.lookup(CNContextImpl.java:1290)
 at com.ibm.ws.naming.util.WsnInitCtx.lookup(WsnInitCtx.java:145)
 at javax.naming.InitialContext.lookup(InitialContext.java:361)
 at emailservice.com.onlinebank.bpel.EmailService20060907T224337EntityAbstractBase$JSE_6.
execute(EmailService20060907T224337EntityAbstractBase.java:32)
 at com.ibm.bpe.framework.ProcessBase6.executeJavaSnippet(ProcessBase6.java:256)
 at emailservice.com.onlinebank.bpel.EmailService20060907T224337EntityBase.invokeSnippet
(EmailService20060907T224337EntityBase.java:40)
```

Note: If this situation is a problem, consider adding more read access permissions for more locations. This should eliminate most, if not all, JavaMail-related harmless security exceptions from the log file.

The permissions required by JavaMail are as follows:

```
grant codeBase "file:${application}" {
  // Allow access to default configuration files
  permission java.io.FilePermission "${java.home}${/}jre${/}lib${/}javamail.address.map", "read";
```

```
  permission java.io.FilePermission "${java.home}${/}jre${/}lib${/}javamail.providers", "read";
  permission java.io.FilePermission "${java.home}${/}jre${/}lib${/}mailcap", "read";
  permission java.io.FilePermission "${java.home}${/}lib${/}javamail.address.map", "read";
  permission java.io.FilePermission "${java.home}${/}lib${/}javamail.providers", "read";
  permission java.io.FilePermission "${java.home}${/}lib${/}mailcap", "read";
  permission java.io.FilePermission "${user.home}${/}.mailcap", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}plugins${/}com.ibm.ws.prereq.javamail.jar", "read";
  // If using an isolated mail provider,
  // add additional file read permissions for each jar defined
  // for the isolated mail provider
  // permission java.io.FilePermission "path${/}mail.jar, "read";

  // Allow connection to mail server using SMTP
  permission java.net.SocketPermission "*:25", "connect,resolve";
  // Allow connection to mail server using SMTPS
  permission java.net.SocketPermission "*:465", "connect,resolve";

  // Allow connection to mail server using IMAP
  permission java.net.SocketPermission "*:143", "connect,resolve";
  // Allow connection to mail server using IMAPS
  permission java.net.SocketPermission "*:993", "connect,resolve";

  // Allow connection to mail server using POP3
  permission java.net.SocketPermission "*:110", "connect,resolve";
  // Allow connection to mail server using POP3S
  permission java.net.SocketPermission "*:995", "connect,resolve";

  // Allow System.getProperties() to be used
  // permission java.util.PropertyPermission "*", "read,write";
  // Otherwise use the following to allow system properties to be read
  permission java.util.PropertyPermission "*", "read";
};
```

Mail: Resources for learning

Use the following links to find relevant supplemental information about the JavaMail API. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming model and decisions

- JavaMail documentation

Programming specifications

- JavaMail 1.3 API documentation (Sun Java specifications)

JavaMail support for IPv6

WebSphere Application Server and its JavaMail component support Internet Protocol Version 6.0 (IPv6), meaning that:

- Both can run on a pure IPv4 network, a pure IPv6 network, *or* a mixed IPv4 and IPv6 network.
- On either the pure IPv6 network or the mixed network, the JavaMail component works with mail servers (such as the SMTP mail transfer agent, and the IMAP and POP3 mail stores) that are also IPv6 compatible. Additionally, a JavaMail component that is run on the mixed IPv4 and IPv6 network can communicate with mail servers using IPv4.

Use of brackets with IPv6 addresses

When you configure a mail session, you can specify the mail server hosts (also known as mail transport and mail store hosts) with domain-qualified host names or numerical IP addresses. Using host names is generally the preferred method. If you use IP addresses, however, consider enclosing IPv6 addresses in square brackets to prevent parsing inaccuracies. See the following example:

```
[fe80::202:57ff:fec4:2334]
```

The JavaMail API requires a combination of many host names or IP addresses with a port number, using the `host:port number` syntax . This extra colon can cause the port number to be read as part of an IPv6 address. Using brackets prevents your JavaMail implementation from processing the extra characters erroneously.

Debugging a mail session

When you need to debug a mail application, you can use the mail debugging feature. The mail component will generate debugging information, on a per session basis, that can be used for problem determination or tuning.

About this task

Enabling the debug mode triggers the mail component of the application server to print the following data to the standard output stream:

- interactions with the mail servers
- properties of the mail session

This output stream is redirected to the SystemOut.log file for the specific application server.

1. Open the administrative console.
2. Click **Resources** → **Mail** → **Mail sessions** → ***mail session***.
3. Click **Enable debug mode**. Debugging is enabled for that session only.
4. Click **Apply** or **OK**.

Example

The following example shows sample mail debugging output:

```
ResourceMgrIm I   WSVR0049I: Binding Test as mail/test
SystemOut     O *** In SessionReferenceable.getReference:
SystemOut     O  added StringRefAddr: type=ws.transport.password, content=****
SystemOut     O  added StringRefAddr: type=ws.isolated.class.loader, content=false
SystemOut     O  added StringRefAddr: type=mail.transport.protocol, content=smtp
SystemOut     O  added StringRefAddr: type=mail.imaps.class, content=com.sun.mail.imap.IMAPSSLStore
SystemOut     O  added StringRefAddr: type=mail.smtp.host, content=smtp.coldmail.com
SystemOut     O  added StringRefAddr: type=mail.debug, content=true
SystemOut     O  added StringRefAddr: type=mail.pop3s.class, content=com.sun.mail.pop3.POP3SSLStore
SystemOut     O  added StringRefAddr: type=mail.from, content=smith@coldmail.com
SystemOut     O  added StringRefAddr: type=mail.smtp.class, content=com.sun.mail.smtp.SMTPTransport
SystemOut     O  added StringRefAddr: type=mail.smtps.class, content=com.sun.mail.smtp.SMTPSSLTransport
SystemOut     O  added StringRefAddr: type=mail.imap.class, content=com.sun.mail.imap.IMAPStore
SystemOut     O  added StringRefAddr: type=mail.smtp.user, content=smith
SystemOut     O  added StringRefAddr: type=mail.pop3.class, content=com.sun.mail.pop3.POP3Store
SystemOut     O  added StringRefAddr: type=mail.mime.address.strict, content=true


SystemOut     O DEBUG: JavaMail version 1.4ea
SystemOut     O DEBUG: java.io.FileNotFoundException:
C:\Program Files\IBM\WebSphere\AppServer\java\jre\lib\javamail.providers
(The system cannot find the file specified.)
SystemOut     O DEBUG: !anyLoaded
SystemOut     O DEBUG: not loading resource: /META-INF/javamail.providers
SystemOut     O DEBUG: successfully loaded resource: /META-INF/javamail.default.providers
SystemOut     O DEBUG: Tables of loaded providers
SystemOut     O DEBUG: Providers Listed By Class Name:
     {com.sun.mail.smtp.SMTPSSLTransport=javax.mail.Provider
[TRANSPORT,smtps,com.sun.mail.smtp.SMTPSSLTransport,Sun Microsystems, Inc],
com.sun.mail.smtp.SMTPTransport=javax.mail.Provider
[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun Microsystems, Inc],
com.sun.mail.imap.IMAPSSLStore=javax.mail.Provider
[STORE,imaps,com.sun.mail.imap.IMAPSSLStore,Sun Microsystems, Inc],
com.sun.mail.pop3.POP3SSLStore=javax.mail.Provider
```

```
[STORE,pop3s,com.sun.mail.pop3.POP3SSLStore,Sun Microsystems, Inc],
com.sun.mail.imap.IMAPStore=javax.mail.Provider
[STORE,imap,com.sun.mail.imap.IMAPStore,Sun Microsystems, Inc],
com.sun.mail.pop3.POP3Store=javax.mail.Provider
[STORE,pop3,com.sun.mail.pop3.POP3Store,Sun Microsystems, Inc]}
SystemOut     O DEBUG: Providers Listed By Protocol:
{imaps=javax.mail.Provider[STORE,imaps,com.sun.mail.imap.IMAPSSLStore,Sun Microsystems,Inc],
imap=javax.mail.Provider[STORE,imap,com.sun.mail.imap.IMAPStore,Sun Microsystems, Inc],
smtps=javax.mail.Provider[TRANSPORT,smtps,com.sun.mail.smtp.SMTPSSLTransport,Sun Microsystems,Inc],
pop3=javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Sun Microsystems, Inc],
pop3s=javax.mail.Provider[STORE,pop3s,com.sun.mail.pop3.POP3SSLStore,Sun Microsystems, Inc],
smtp=javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun Microsystems, Inc]}
SystemOut     O DEBUG: successfully loaded resource: /META-INF/javamail.default.address.map
SystemOut     O DEBUG: !anyLoaded
SystemOut     O DEBUG: not loading resource: /META-INF/javamail.address.map
SystemOut     O DEBUG: java.io.FileNotFoundException:
     C:\Program Files\IBM\WebSphere\AppServer\java\jre\lib\javamail.address.map
     (The system cannot find the file specified.)
SystemOut     O *** In SessionFactory.setPasswordAuthentication,
   TRANSPORT PasswordAuthentication is based on:
SystemOut     O  url=smtp://smith@smtp.coldmail.com
SystemOut     O  user=smith
SystemOut     O  password=****
SystemOut     O *** In SessionFactory.getObjectInstance, session properties:
SystemOut     O  mail.transport.protocol=smtp
SystemOut     O  mail.imaps.class=com.sun.mail.imap.IMAPSSLStore
SystemOut     O  mail.smtp.host=smtp.coldmail.com
SystemOut     O  mail.debug=true

SystemOut     O  mail.pop3s.class=com.sun.mail.pop3.POP3SSLStore
SystemOut     O  mail.from=smith@coldmail.com
SystemOut     O  mail.smtp.class=com.sun.mail.smtp.SMTPTransport
SystemOut     O  mail.smtps.class=com.sun.mail.smtp.SMTPSSLTransport
SystemOut     O  mail.imap.class=com.sun.mail.imap.IMAPStore
SystemOut     O  mail.smtp.user=smith
SystemOut     O  mail.pop3.class=com.sun.mail.pop3.POP3Store
SystemOut     O  mail.mime.address.strict=true
SystemOut     O DEBUG: mail.smtp.class property exists and points to com.sun.mail.smtp.SMTPTransport
SystemOut     O DEBUG SMTP: useEhlo true, useAuth false
SystemOut     O DEBUG SMTP: trying to connect to host "smtp.coldmail.com", port 25, isSSL false


javax.mail.MessagingException: Unknown SMTP host: smtp.coldmail.com;
  nested exception is:
 java.net.UnknownHostException: smtp.coldmail.com
 at com.sun.mail.smtp.SMTPTransport.openServer(SMTPTransport.java:1280)
 at com.sun.mail.smtp.SMTPTransport.protocolConnect(SMTPTransport.java:370)
 at javax.mail.Service.connect(Service.java:275)
 at javax.mail.Service.connect(Service.java:156)
 at javax.mail.Service.connect(Service.java:105)
 at javax.mail.Transport.send0(Transport.java:168)
 at javax.mail.Transport.send(Transport.java:98)
 at com.ibm.ws.mail.ut.TestServlet.doTask(TestServlet.java:104)
 at com.ibm.ws.mail.ut.TestServlet.doGet(TestServlet.java:65)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:707)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:820)
 at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:1397)
 at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:759)
 at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:429)
 at com.ibm.ws.webcontainer.servlet.ServletWrapperImpl.handleRequest(ServletWrapperImpl.java:175)
 at com.ibm.ws.webcontainer.webapp.WebApp.handleRequest(WebApp.java:3512)
 at com.ibm.ws.webcontainer.webapp.WebGroup.handleRequest(WebGroup.java:273)
 at com.ibm.ws.webcontainer.WebContainer.handleRequest(WebContainer.java:896)
 at com.ibm.ws.webcontainer.WSWebContainer.handleRequest(WSWebContainer.java:1530)
 at com.ibm.ws.webcontainer.channel.WCChannelLink.ready(WCChannelLink.java:161)
 at com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.handleDiscrimination(HttpInboundLink.java:455)
 at com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.handleNewInformation(HttpInboundLink.java:384)
 at com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.ready(HttpInboundLink.java:272)
 at com.ibm.ws.tcp.channel.impl.NewConnectionInitialReadCallback.sendToDiscriminators(NewConnectionInitialReadCallback.java:214)
 at com.ibm.ws.tcp.channel.impl.NewConnectionInitialReadCallback.complete(NewConnectionInitialReadCallback.java:113)
 at com.ibm.ws.tcp.channel.impl.AioReadCompletionListener.futureCompleted(AioReadCompletionListener.java:165)
 at com.ibm.io.async.AbstractAsyncFuture.invokeCallback(AbstractAsyncFuture.java:217)
 at com.ibm.io.async.AsyncChannelFuture.fireCompletionActions(AsyncChannelFuture.java:161)
 at com.ibm.io.async.AsyncFuture.completed(AsyncFuture.java:138)
 at com.ibm.io.async.ResultHandler.complete(ResultHandler.java:202)
 at com.ibm.io.async.ResultHandler.runEventProcessingLoop(ResultHandler.java:766)
 at com.ibm.io.async.ResultHandler$2.run(ResultHandler.java:896)
 at com.ibm.ws.util.ThreadPool$Worker.run(ThreadPool.java:1487)
Caused by: java.net.UnknownHostException: smtp.coldmail.com
 at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:196)
 at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:366)
 at java.net.Socket.connect(Socket.java:519)
 at java.net.Socket.connect(Socket.java:469)
 at com.sun.mail.util.SocketFetcher.createSocket(SocketFetcher.java:232)
 at com.sun.mail.util.SocketFetcher.getSocket(SocketFetcher.java:189)
 at com.sun.mail.smtp.SMTPTransport.openServer(SMTPTransport.java:1250)
 ... 32 more
```

This output illustrates a connection failure to a Simple Mail Transfer Protocol (SMTP) server because a fictitious name, smtp.coldmail.com, is specified as the server name.

The following list provides tips on reading the previous sample of debugger output:

- The lines headed by `DEBUG` are printed by the mail provider at run time, while the two lines headed by `***` are printed by the application server at run time.
- In the second paragraph of code, the first few lines state that some configuration files are skipped. The mail component attempts to load a number of configuration files from different locations at run time. All those files are not required. If a required file cannot be accessed, however, the mail component creates an exception. In this sample, there is no exception and the third-line announces that default providers are loaded.
- The next few lines, headed by either `Providers Listed by Class Name` or `Providers Listed by Protocols`, show the protocol providers that are loaded. The six providers that are listed are the default protocol providers that come under the built-in mail provider for the application server. If you install special service providers, and these providers are used in the current mail session, those providers will be listed here with the default providers.
- The two lines headed by `***` and the few lines below them are printed by the application server to show the configuration properties of the current mail session. Although these properties are listed by their internal name rather than the name you establish in the administrative console, you can easily recognize the relationships between them. For example, the mail.store.protocol property corresponds to the **Protocol** property in the **Incoming Mail Properties** section of the console panel for mail session configuration. Review the listed properties and values to verify that they correspond.
- The few lines above the exception stack show the mail activities when sending a message. First, the JavaMail API recognizes that the transport protocol is set to SMTP and that the com.sun.mail.smtp.SMTPTransport provider exists. Next, the output log displays the useEhlo and useAuth parameters, which are used by SMTP. Finally, the log shows the SMTP provider trying to connect to the smtp.coldmail.com mail server.
- The output log show the exception stack next. This data indicates that the specified mail server either does not exist or is not functioning.

Using URL resources within an application

Java Platform, Enterprise Edition (Java EE) applications can use Uniform Resource Locators (URLs) by looking up references to logically named URL connection factories through the `java:comp/env/url` subcontext, which is declared in the application deployment descriptor and mapped to installation specific URL resources.

About this task

As in the case of other Java EE resources, this can be done in order to eliminate the need for the application to hard code references to external resources. The process is the same used with other Java EE resources, such as JDBC objects and JavaMail sessions.

1. Develop an application that relies on naming features.
2. Define resource references while assembling your application. A URL resource that uses a built-in protocol, such as HTTP, FTP, or file, can use the default URL provider. URL resources that use other protocols need to use a custom URL provider.
3. Configure your URL resources within an application.
   a. Open the administrative console.
   b. Click **Resources**>**URL** in the console navigation tree.
   c. Click either **URL Providers** or **URLs** to modify the appropriate resource.
4. Optional: Configure URL providers and URLs within an application client using the Application Client Resource Configuration Tool (ACRCT).

5. Manage URL providers and URL resources used by the deployed application. To update or remove existing URL configurations:
   a. Open the administrative console.
   b. Click **Resources** > **URL** in the console navigation tree.
   c. Click either **URL Providers** or **URLs** to modify the appropriate resource.
   d. Select the URL to modify.
   e. Modify the URL properties.
   f. Click **Apply** or **OK**.

   To remove URL providers and URLs, after step 2, click *URL_provider* > **URLs**. Select the URL you want to remove and click **Delete**. Then, click **Apply** or **OK**.

URLs

A Uniform Resource Locator (URL) is an identifier that points to an electronically accessible resource, such as a directory file on a machine in a network, or a document stored in a database.

URLs appear in the format *scheme*:*scheme_information*.

You can represent a *scheme* as HTTP, FTP, file, or another term that identifies the type of resource and the mechanism by which you can access the resource.

In a World Wide Web browser location or address box, a URL for a file available using HyperText Transfer Protocol (HTTP) starts with `http:`. An example is `http://www.ibm.com`. Files available using File Transfer Protocol (FTP) start with `ftp:`. Files available locally start with `file:`.

The *scheme_information* commonly identifies the Internet machine making a resource available, the path to that resource, and the resource name. The *scheme_information* for HTTP, FTP and file generally starts with two slashes (//), then provides the Internet address separated from the resource path name with one slash (/). For example,

```
http://www.ibm.com/software/webservers/appserv/library.html.
```

For HTTP and FTP, the path name ends in a slash when the URL points to a directory. In such cases, the server generally returns the default index for the directory.

URL provider collection

Use this page to view existing URL providers, which supply the implementation classes that are necessary for WebSphere Application Server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit for the Java™ Platform, compatible with the Java 2 Standard Edition Platform 1.3.1. These protocols include HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP), which work for must URLs.

To view this administrative console page, click **Resources** > **URL** > **URL Providers**.

Name

Specifies the administrative name for the URL provider.

Scope

Specifies the scope of this URL provider, which can support multiple URL configurations. All of the URL configurations that are supported by this provider inherit this scope.

Description

Describes the URL provider for your administrative records.

URL provider settings

Use this page to configure URL providers, which support WebSphere Application Server connections to a URL over a specific protocol.

To view this administrative console page, click **Resources** > **URL** > **URL Providers** > *URL_provider*.

Scope

Specifies the scope of this URL provider, which can support multiple URL configurations. All of the URL configurations that are supported by this provider inherit this scope.

Name

Specifies the administrative name for the URL provider.

Description

Describes the URL provider, for your administrative records.

Class path

Specifies paths or JAR file names which together form the location for the resource provider classes.

Stream handler class name

Specifies fully qualified name of a user-defined Java class that extends the `java.net.URLStreamHandler` class for a particular URL protocol, such as FTP.

Protocol

Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.

URL collection

Use this page to view existing Uniform Resource Locator (URL) configurations, which are sets of properties that define WebSphere Application Server connections to URLs. URLs are location names that represent electronically accessible resources, such as a directory file on a machine in a network or a document stored in a database.

You can access this administrative console page in one of two ways:

- **Resources** > **URL Providers** > *URL_provider* > **URLs**
- **Resources** > **URL** > **URLs**

Name

Specifies the display name for the resource.

JNDI Name

Specifies the JNDI name.

Scope

Specifies the scope of the URL provider that supports this URL configuration. Only applications that are installed within this scope can use this URL configuration to access URL resources.

Provider

Specifies the URL provider that supplies the implementation classes for using a specific protocol to access this URL.

Description

Specifies the description of the resource.

Category

Specifies the category string, which you can use to classify or group the resource.

URL configuration settings

Use this page to define connections to Uniform Resource Locators (URLs), which are location names that represent electronically accessible resources. A collection of URL connection properties is often called a URL configuration in the WebSphere Application Server environment. The targeted resources are remote to your Application Server installation.

You can access this administrative console page in one of two ways:

- **Resources** > **URL** > **URLs** > *URL*
- **Resources** > **URL** > **URL Providers** > *URL_provider* > **URLs** > *URL*

Scope

Specifies the scope of the URL provider that supports this URL configuration. Only applications that are installed within this scope can use this URL configuration to access URL resources.

Provider

Specifies the URL provider that WebSphere Application Server uses for this URL configuration.

Note: If you previously defined one or more URL providers at the relevant scope, you see a list from which you can select an existing URL provider for your new URL configuration.

Create New Provider

Provides the option of configuring a new URL provider for the new URL configuration.

Create New Provider is displayed only when you create a new URL from the **Resources** > **URL** > **URLs** path. In this flow, you can create a new URL provider if needed. The URL provider can not be changed during an edit.

Clicking **Create New Provider** triggers the console to display the URL provider configuration page, where you create a new provider. After you click **OK** to save your settings, you see the URL collection page. Click **New** to define a new URL configuration for use with the new provider; the console now displays a configuration page that lists the new provider as the URL configuration Provider.

Name

Specifies the display name for the resource.

JNDI Name

Specifies the JNDI name.

Note: Adhere to the following requirements for JNDI names:

- Do not assign duplicate JNDI names across different resource types (such as mail sessions versus URL configurations).
- Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.

Description

Specifies the description of the resource.

Category

Specifies the category string, which you can use to classify or group the resource.

Specification

Specifies the string from which to form a URL.

URLs: Resources for learning

Use the following links to find relevant supplemental information about URLs. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming specifications

- W3C Architecture - Naming and Addressing: URIs, URLs
- URL API documentation

Mapping logical names of environment resources to their physical names

This topic provides instructions on configuring *new* resource environment entries, which define environment resources that are the binding targets for resource-environment-references in an application's deployment descriptor.

1. Configure a resource environment provider, which is a library that provides the implementation for an environment resource factory. In the administration console, begin by clicking **Resources > Resource Environment > Resource Environment Providers > New**. (See the New Resource Environment Provider topic for more information.)
2. After saving your resource environment provider, go to the Additional Properties heading and click **Resource environment entries**. Click **New** to define a new resource environment entry. Refer to the "Resource environment entry settings" on page 1284 topic for descriptions of the required fields.
3. You also might need to create a referenceable, which specifies the factory class name that converts information in the name space into a class instance for your resource. To view the appropriate administrative console page for referenceables, click **Resources > Resource Environment > Resource Environment Providers >** *your_resource_environment_provider* **> Referenceables**. Click **New** to begin the configuration process. See the "Referenceables settings" on page 1286 topic for descriptions of the required fields.

Resource environment providers and resource environment entries

A resource environment reference maps a logical name used by the client application to the physical name of an object.

Not all objects bound into the server JNDI namespace are intended for use by an application client. For example, the WebSphere Application Server client run time does not support the use of Java 2 Connector (J2C) objects on the client. The object needs to be remotable, and the client-side implementations must be made available on the application client run-time classpath.

Resource environment references are different than resource references. Resource environment references allow your application client to use a logical name to look up a resource bound into the server JNDI namespace. A resource reference allows your application to use a logical name to look up a local J2EE resource. The J2EE specification does not specify a particular implementation of a resource.

Resource environment provider collection

Use this page to view resource environment providers, which encapsulate the referenceables that convert resource environment entry data into resource objects.

To view this administrative console page, click **Resources > Resource Environment > Resource Environment Providers**.

Name

Specifies a text identifier for the resource environment provider.

Data type String

Scope
Specifies the scope of this resource environment provider, which automatically becomes the scope of the resource environment entries that you define with this provider.

Description
Specifies a text string describing the resource environment provider.

Data type String

Resource environment provider settings
Use this page to create settings for a resource environment provider.

To view this administrative console page, click **Resources** > **Resource environment** > **Resource environment providers** > *resource environment provider*.

Scope:

Specifies the scope of this resource environment provider, which automatically becomes the scope of the resource environment entries that you define with this provider.

Name:

Specifies the name of the resource provider.

Data type String

Description:

Specifies a text description for the resource provider.

Data type String

New Resource environment provider
Use this page to define the configuration for a library that provides the implementation for a environment resource factory.

To view this administrative console page, click **Resources > Resource Environment > Resource Environment Providers > New**.

Scope:

Specifies the scope of this resource environment provider, which automatically becomes the scope of the resource environment entries that you define with this provider.

Name:

Specifies a text identifier for the resource environment provider.

Data type String

Description:

Specifies a text string describing the resource environment provider.

Data type String

Resource environment entries collection

Use this page to view configured resource environment entries. Within an application server name space, the data contained in a resource environment entry is converted into an object that represents a physical resource. This resource is frequently called an *environment resource*.

An environment resource can be of any arbitrary type. See the latest EJB specification for more information about resource environment references and environment resources.

You can access this administrative console page in one of two ways:

- **Resources** > **Resource Environment** > **Resource environment entries**
- **Resources** >**Resource Environment** > **Resource Environment Providers** > *resource_environment_provider* > **Resource Environment Entries**

Name

Specifies a text identifier that helps distinguish this resource environment entry from others.

For example, you can use *My Resource* for the name.

Data type String

JNDI Name

Specifies the string to be used when looking up this environment resource using JNDI.

This is the string to which you bind resource environment reference deployment descriptors.

Data type String

Scope

Specifies the resource environment entry scope, which is inherited from the resource environment provider.

Provider

Specifies the resource environment provider for this entry. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.

Description

Specifies text for information to help further identify and distinguish this resource

Data type String

Category

Specifies a category you can use to group environment resources according to some common feature.

It is strictly an organizational property and has no effect on the function of the environment resource.

Data type String

Resource environment entry settings

Use this page to configure resource environment entries. Within an application server name space, the data contained in a resource environment entry is converted into an object that represents a physical

resource. Rather than represent a connection factory, which provides connections to a resource, this object *directly* represents a resource. This design can make the resource available to application modules that do not run entirely on the application server. Examples include some application clients and Web modules.

You can access this administrative console page in one of two ways:

- **Resources** > **Resource Environment** > **Resource environment entries** > *resource_environment_entry*
- **Resources** >**Resource Environment** > **Resource Environment Providers** > *resource_environment_provider* > **Resource Environment Entries** > *resource_environment_entry*

Scope:

Specifies the scope of the resource environment provider, which is a library that supplies the implementation class for a resource environment factory. Within a JNDI name space, WebSphere Application Server uses the factory to transform your resource environment entry into an object that directly represents a physical resource.

Provider:

Specifies the resource environment provider.

Provider shows all of the existing resource environment providers that are defined at the relevant scope. Select one from the list if you want to use an existing resource environment provider as Provider.

Name:

Specifies a display name for the resource.

Data type String

JNDI name:

Specifies the JNDI name for the resource, including any naming subcontexts.

This name is used as the linkage between the platform's binding information for resources defined by a module's deployment descriptor and actual resources bound into JNDI by the platform.

Data type String

Note: Adhere to the following requirements for JNDI names:

- Do not assign duplicate JNDI names across different resource types (such as resource environment entries versus J2C connection factories).
- Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.

Description:

Specifies a text description for the resource.

Data type String

Category:

Specifies a category string that you can use to classify or group the resource.

Data type String

Referenceables:

Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.

Data type Drop-down menu

Referenceables collection

Use this page to view configured referenceables, which encapsulate the class name of the factory that converts information in the name space into a class instance for a physical resource.

To view this administrative console page, click **Resources** > **Resource environment** > **Resource Environment Providers** > *resource_environment_provider* > **Referenceables**.

Factory Class name

Specifies a javax.naming.spi.ObjectFactory implementation name

Data type String

Class name

Specifies the package name of the referenceable, for example: `javax.naming.Referenceable`

Data type String

Referenceables settings

Use this page to set the class name of the factory that converts information in the name space into a class instance of a physical resource.

To view this administrative console page, click **Resources > Resource Environment > Resource Environment Providers >** *resource_environment_provider* **> Referenceables >** *referenceable*.

Factory class name:

Specifies a javax.naming.ObjectFactory implementation class name

Data type String

Class name:

Specifies the Java type to which a Referenceable provides access, for binding validation and to create the reference.

Data type String

Resource environment references

Use this page to designate how the resource environment references of application modules map to remote resources, which are represented in the product as resource environment entries.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Resource environment references**.

Each row of the table depicts a resource environment reference within a specific module of your application. If you bound any references to resource environment entries during application assembly, you see the JNDI names of those resource environment entries in the applicable rows.

To set the mapping relationships between your resource environment references and resource environment entries:

1. Select a row. Be aware that if you check multiple rows on this page, the resource mapping target that you select in step 2 applies to all of those references.
2. Click **Browse** to select a resource environment entry from the new page that is displayed, the Available Resources page. The Available Resources page shows all resource environment entries that are available mapping targets for your application references.
3. Click **Apply**. The console displays the Resource environment references page again. In the rows that you previously selected, you now see the JNDI name of the new resource mapping target.
4. Repeat the previous steps as necessary.
5. Click **OK**. You now return to the general configuration page for your enterprise application.

Table column heading descriptions:

Select
Select the check boxes of the rows that you want to edit.

Module
The name of a module in the application.

EJB
The name of an enterprise bean that is accessed by the module.

URI
Specifies location of the module relative to the root of the application EAR file.

Reference binding
The name of a resource environment reference that is declared in the deployment descriptor of the application module. The reference corresponds to a resource that is bound as a resource environment entry into the JNDI name space of the application server.

JNDI name
The Java Naming and Directory Interface (JNDI) name of the resource environment entry that is the mapping target of the resource environment reference.

Data type String

Chapter 11. Security

Task overview: Securing resources

WebSphere Application Server supports the Java Platform, Enterprise Edition (Java EE) model for creating, assembling, securing, and deploying applications. Applications are often created, assembled, and deployed in different phases and by different teams.

About this task

You can secure resources in a Java EE environment by following the required high-level steps. Consult the Java EE specifications for complete details.

- Set up and enable security. You must address several issues prior to authenticating users, authorizing access to resources, securing applications, and securing communications. These security issues include migration, interoperability, and installation. After installing WebSphere Application Server, you must determine the proper level of security that is needed for your environment. For more information, see Setting up and enabling security.
- Configure multiple domains. Security domains enable you to define multiple security configurations for use in your environment. For example, you can define different security (such as a different user registry) for user applications than for administrative applications. You can also define separate security configurations for user applications deployed to different servers and clusters. For more information, see Configuring multiple security domains
- Authenticate users. The process of authenticating users involves a user registry and an authentication mechanism. Optionally, you can define trust between WebSphere Application Server and a proxy server, configure single sign-on capability, and specify how to propagate security attributes between application servers. For more information, see Authenticating users.
- Authorize access to resources. WebSphere Application Server provides many different methods for authorizing accessing resources. For example, you can assign roles to users and configure a built-in or external authorization provider. For more information, see Authorizing access to resources.
- Secure communications. WebSphere Application Server provides several methods to secure communication between a server and a client. For more information, see Securing communications.
- Develop extensions to the WebSphere security infrastructure. WebSphere Application Server provides various plug points so that you can extend the security infrastructure. For more information, see “Developing extensions to the WebSphere security infrastructure” on page 1290.
- Use the Auditing Facility to track and archive auditable events to ensure the integrity of your system. For more information, see Auditing the security infrastructure
- Secure various types of WebSphere applications. See **Securing WebSphere applications** for tasks involving developing, deploying, and administering secure applications, including Web applications, Web services, and many other types. This section highlights the security concerns and tasks that are specific to each type of application.
- Tune, harden, and maintain security configurations. After you have installed WebSphere Application Server, there are several considerations for tuning, strengthening, and maintaining your security configuration. For more information, see Tuning, hardening, and maintaining.
- Troubleshoot security configurations. For more information, see Troubleshooting security configurations.

Results

Your applications and production environment are secured.

Example

See Security: Resources for learning for more information on the WebSphere Application Server security architecture.

Developing extensions to the WebSphere security infrastructure

WebSphere Application Server provides various plug points so that you can extend the security infrastructure. Extending this security infrastructure involves several activities including: Developing custom user registries, developing applications that us programmatic security, and customizing Web application login forms.

About this task

The following topics are covered in this section:

- Developing custom user registries
- Developing applications that use programmatic security
- Customizing Web application login forms
- Customizing application login forms with Java Authentication and Authorization Service (JAAS)
- Securing transports with Java Secure Sockets Extension (JSSE) and Java Cryptography Extension (JCE) programming interfaces
- Implementing tokens for security attribute propagation

Developing standalone custom registries

This development provides considerable flexibility in adapting WebSphere Application Server security to various environments where some notion of a user registry, other than LDAP or Local OS, already exists in the operational environment.

Before you begin

WebSphere Application Server security supports the use of standalone custom registries in addition to the local operating system registry, standalone Lightweight Directory Access Protocol (LDAP) registries, and federated repositories for authentication and authorization purposes. A standalone custom-implemented registry uses the UserRegistry Java interface as provided by WebSphere Application Server. A standalone custom-implemented registry can support virtually any type or notion of an accounts repository from a relational database, flat file, and so on.

Implementing a standalone custom registry is a software development effort. Implement the methods that are defined in the com.ibm.websphere.security.UserRegistry interface to make calls to the appropriate registry to obtain user and group information. The interface defines a general set of methods for encapsulating a wide variety of registries. You can configure a standalone custom registry as the selected repository when configuring WebSphere Application Server security on the Global security panel.

In WebSphere Application Server Version 7.0, make sure that your implementation of the standalone custom registry does not depend on any WebSphere Application Server components such as data sources, Enterprise JavaBeans (EJB) and Java Naming and Directory Interface (JNDI). You can not have this dependency because security is initialized and enabled prior to most of the other WebSphere Application Server components during startup. If your previous implementation used these components, make a change that eliminates the dependency. For example, if your previous implementation used data sources to connect to a database, instead use the JDBC java.sql.DriverManager interface to connect to the database.

Note: The user registry is used in controllers and servants. There is an increased risk of integrity exposure in that configuration if the registry implementation is not secured.

Refer to the Migrating custom user registries for more information on migrating. If your previous implementation uses data sources to connect to a database, change the implementation to use Java database connectivity (JDBC) connections.

1. Implement all the methods in the interface except for the CreateCredential method, which is implemented by WebSphere Application Server. FileRegistrySample.java file is provided for reference.

   Note: The sample provided is intended to familiarize you with this feature. Do not use this sample in an actual production environment.

2. Build your implementation.

   To compile your code, you need the *app_server_install_root*/*Base*/plugins/com.ibm.ws.runtime.jar and the *app_server_install_root*/*Base*/plugins/com.ibm.ws.security.crypto.jar files in your class path. For example:

   ```
   %install_root%/java/bin/javac -classpath
   app_server_install_root/Base/plugins/com.ibm.ws.runtime.jar;
   app_server_install_root/Base/plugins/com.ibm.ws.security.crypto.jar your_implementation_file.java
   ```

3. Copy the class files that are generated in the previous step to the product class path.

   The preferred location is the *%install_root%*/lib/ext directory. Copy these class files to all of the product process class paths.

4. Follow the steps in Configuring standalone custom registries to configure your implementation using the administrative console. This step is required to implement custom user registries.

What to do next

If you enable security, make sure that you complete the remaining steps:

1. Save and synchronize the configuration and restart all of the servers.
2. Try accessing some J2EE resources to verify that the custom registry implementation is correct.

Example: Standalone custom registries

Use these links to view registry examples.

A *standalone custom registry* is a customer-implemented registry that implements the UserRegistry Java interface, as provided by WebSphere Application Server. A custom-implemented registry can support virtually any type or form of an accounts repository from a relational database, flat file, and so on. The custom registry provides considerable flexibility in adapting WebSphere Application Server security to various environments where some form of a registry, other than a federated repository, Lightweight Directory Access Protocol (LDAP) registry, or local operating system registry, already exist in the operational environment.

To view a sample standalone custom registry, refer to the following files:

- FileRegistrySample.java file
- users.props file
- groups.props file

Result.java file

This module is used by user registries in WebSphere Application Server when calling the getUsers and getGroups methods. The user registries use this method to set the list of users and groups and to indicate if more users and groups in the user registry exist than requested.

```
//
//  5639-D57, 5630-A36, 5630-A37, 5724-D18
// (C) COPYRIGHT International Business Machines Corp. 1997, 2005
//  All Rights Reserved * Licensed Materials - Property of IBM
//
package com.ibm.websphere.security;

import java.util.List;
```

```
public class Result implements java.io.Serializable {
    /**
      Default constructor
    */
    public Result() {
    }

    /**
       Returns the list of users and groups
       @return the list of users and groups
    */
    public List getList() {
      return list;
    }

    /**
       indicates if there are more users and groups in the registry
    */
    public boolean hasMore() {
      return more;
    }
    /**
       Set the flag to indicate that there are more users and groups
       in the registry to true
    */
    public void setHasMore() {
      more = true;
    }

    /*
      Set the list of users and groups
      @param list   list of users/groups
    */
    public void setList(List list) {
      this.list = list;
    }

    private boolean more = false;
    private List list;
}
```

UserRegistry.java files

The following file is a custom property that is used with a custom user registry.

For more information, see Configuring standalone custom registries.

```
// 5639-D57, 5630-A36, 5630-A37, 5724-D18
// (C) COPYRIGHT International Business Machines Corp. 1997, 2005
// All Rights Reserved * Licensed Materials - Property of IBM
//
// DESCRIPTION:
//
//    This file is the UserRegistry interface that custom registries in WebSphere
//    Application Server implement to enable WebSphere security to use the custom
//    registry.
//
package com.ibm.websphere.security;

import java.util.*;
import java.rmi.*;
import java.security.cert.X509Certificate;
import com.ibm.websphere.security.cred.WSCredential;/**
 * Implementing this interface enables WebSphere Application Server Security
 * to use custom registries. This interface extends java.rmi.Remote because the
 * registry can be in a remote process.
```

```
 *
 * Implementation of this interface must provide implementations for:
*
* initialize(java.util.Properties)
* checkPassword(String,String)
* mapCertificate(X509Certificate[])
* getRealm
* getUsers(String,int)
* getUserDisplayName(String)
* getUniqueUserId(String)
* getUserSecurityName(String)
* isValidUser(String)
* getGroups(String,int)
* getGroupDisplayName(String)
* getUniqueGroupId(String)
* getUniqueGroupIds(String)
* getGroupSecurityName(String)
* isValidGroup(String)
* getGroupsForUser(String)
* getUsersForGroup(String,int)
* createCredential(String)
**/

public interface UserRegistry extends java.rmi.Remote
{

  /**
   * Initializes the registry. This method is called when creating the
   * registry.
   *
   * @param props the registry-specific properties with which to
   *               initialize the  custom registry
   * @exception CustomRegistryException
   *                     if there is any registry specific problem
   * @exception RemoteException
   *    as this extends java.rmi.Remote
   **/
   public void initialize(java.util.Properties props)
      throws CustomRegistryException,
             RemoteException;  /**
   * Checks the password of the user. This method is called to authenticate a
   * user when the user's name and password are given.
   *
   * @param userSecurityName the name of the user
   * @param password the password of the user
   * @return a valid userSecurityName. Normally this is
   *   the name of same user whose password was checked but if the
   *  implementation wants to return any other valid
   *  userSecurityName in the registry it can do so
   * @exception CheckPasswordFailedException if userSecurityName/
   *  password combination does not exist in the registry
   * @exception CustomRegistryException if there is any registry specific
   *            problem
   * @exception RemoteException as this extends java.rmi.Remote
   **/
   public String checkPassword(String userSecurityName, String password)
      throws PasswordCheckFailedException,
             CustomRegistryException,
             RemoteException;  /**
   * Maps a certificate (of X509 format) to a valid user in the registry.
   * This is used to map the name in the certificate supplied by a browser
   * to a valid userSecurityName in the registry
   *
```

```
* @param cert the X509 certificate chain
* @return the mapped name of the user userSecurityName
* @exception CertificateMapNotSupportedException if the particular
*            certificate is not supported.
* @exception CertificateMapFailedException if the mapping of the
*            certificate fails.
* @exception CustomRegistryException if there is any registry specific
*            problem
* @exception RemoteException as this extends java.rmi.Remote
**/
public String mapCertificate(X509Certificate[] cert)
   throws CertificateMapNotSupportedException,
          CertificateMapFailedException,
          CustomRegistryException,
          RemoteException;  /**
* Returns the realm of the registry.
*
* @return the realm. The realm is a registry-specific string indicating
*            the realm or domain for which this registry
*            applies.  For example, for OS400 or AIX this would be the
*            host name of the system whose user registry this object
*            represents.
*            If null is returned by this method realm defaults to the
*            value of "customRealm". It is recommended that you use
*            your own value for realm.
* @exception CustomRegistryException if there is any registry specific
*            problem
* @exception RemoteException as this extends java.rmi.Remote
**/
public String getRealm()
   throws CustomRegistryException,
          RemoteException;  /**
* Gets a list of users that match a pattern in the registry.
* The maximum number of users returned is defined by the limit
* argument.
* This method is called by administrative console and by scripting (command
* line) to make available the users in the registry for adding them (users)
* to roles.
*
* @parameter pattern the pattern to match. (For example., a* will match all
*   userSecurityNames starting with a)
* @parameter limit the maximum number of users that should be returned.
*  This is very useful in situations where there are thousands of
*            users in the registry and getting all of them at once is not
*            practical. A value of 0 implies get all the users and hence
*            must be used with care.
* @return a Result object that contains the list of users
*   requested and a flag to indicate if more users exist.
* @exception CustomRegistryException if there is any registry specific
*            problem
* @exception RemoteException as this extends java.rmi.Remote
**/
public Result getUsers(String pattern, int limit)
   throws CustomRegistryException,
          RemoteException;  /**
* Returns the display name for the user specified by userSecurityName.
*
* This method is called only when the user information displays
* (information purposes only, for example, in the administrative console) and not used
* in the actual authentication or authorization purposes. If there are no
* display names in the registry return null or empty string.
*
* In WebSphere Application Server Version 4.0 custom registry, if you had a display
```

```
 * name for the user and  if it was different from the security name, the display name
 * was returned for the EJB methods getCallerPrincipal() and the servlet methods
 * getUserPrincipal() and  getRemoteUser().
 * In WebSphere Application Server Version 5.0 for the same methods the security
 * name is returned by default. This is the recommended way as the display name
 * is not unique and might create security holes.
 *
 * See the documentation for more information.
 *
 * @parameter userSecurityName the name of the user.
 * @return the display name for the user. The display name
 *   is a registry-specific string that represents a descriptive, not
 *  necessarily unique, name for a user. If a display name does
 *            not exist return null or empty string.
 * @exception EntryNotFoundException if userSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUserDisplayName(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;  /**
 * Returns the unique ID for a userSecurityName. This method is called when
 * creating a credential for a user.
 *
 * @parameter userSecurityName the name of the user.
 * @return the unique ID of the user. The unique ID for a user is
 *   the stringified form of some unique, registry-specific, data
 *  that serves to represent the user.  For example, for the UNIX
 *  user registry, the unique ID for a user can be the UID.
 * @exception EntryNotFoundException if userSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUniqueUserId(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;  /**
 * Returns the name for a user given its unique ID.
 *
 * @parameter uniqueUserId the unique ID of the user.
 * @return the userSecurityName of the user.
 * @exception EntryNotFoundException if the uniqueUserID does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUserSecurityName(String uniqueUserId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Determines if the userSecurityName exists in the registry
 *
 * @parameter userSecurityName the name of the user
 * @return true if the user is valid. false otherwise
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
```

```
 public boolean isValidUser(String userSecurityName)
    throws CustomRegistryException,
           RemoteException;

/**
 * Gets a list of groups that match a pattern in the registry.
 * The maximum number of groups returned is defined by the limit
 * argument.
 * This method is called by the administrative console and scripting
 * (command line) to make available the groups in the registry for adding
 * them (groups) to roles.
 *
 * @parameter pattern the pattern to match. (For e.g., a* will match all
 *   groupSecurityNames starting with a)
 * @parameter limit the maximum number of groups to return.
 *  This is very useful in situations where there are thousands of
 *            groups in the registry and getting all of them at once is not
 *            practical. A value of 0 implies get all the groups and hence
 *            must be used with care.
 * @return a Result object that contains the list of groups
 *   requested and a flag to indicate if more groups exist.
 * @exception CustomRegistryException if there is any registry-specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public Result getGroups(String pattern, int limit)
    throws CustomRegistryException,
           RemoteException;

/**
 * Returns the display name for the group specified by groupSecurityName.
 *
 * This method may be called only when the group information displayed
 * (for example, the administrative console) and not used in the actual
 * authentication or authorization purposes. If there are no display names
 * in the registry return null or empty string.
 *
 * @parameter groupSecurityName the name of the group.
 * @return the display name for the group. The display name
 *   is a registry-specific string that represents a descriptive, not
 *  necessarily unique, name for a group. If a display name does
 *            not exist return null or empty string.
 * @exception EntryNotFoundException if groupSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getGroupDisplayName(String groupSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Returns the unique ID for a group.

 * @parameter groupSecurityName the name of the group.
 * @return the unique ID of the group. The unique ID for
 *  a group is the stringified form of some unique,
 *  registry-specific, data that serves to represent the group.
 *  For example, for the UNIX user registry, the unique ID might
 *  be the GID.
 * @exception EntryNotFoundException if groupSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
```

```
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUniqueGroupId(String groupSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;


/**
 * Returns the unique IDs for all the groups that contain the unique ID of
 * a user.
 * Called during creation of a user's credential.
 *
 * @parameter uniqueUserId the unique ID of the user.
 * @return a list of all the group unique IDs that the unique user ID
 *   belongs to. The unique ID for an entry is the stringified
 *  form of some unique, registry-specific, data that serves
 *  to represent the entry.  For example, for the
 *   UNIX user registry, the unique ID for a group could be the GID
 *  and the unique ID for the user could be the UID.
 * @exception EntryNotFoundException if unique user ID does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public List getUniqueGroupIds(String uniqueUserId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Returns the name for a group given its unique ID.
 *
 * @parameter uniqueGroupId the unique ID of the group.
 * @return the name of the group.
 * @exception EntryNotFoundException if the uniqueGroupId does not exist.
 * @exception CustomRegistryException if there is any registry-specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getGroupSecurityName(String uniqueGroupId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Determines if the groupSecurityName exists in the registry
 *
 * @parameter groupSecurityName the name of the group
 * @return true if the groups exists, false otherwise
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public boolean isValidGroup(String groupSecurityName)
    throws CustomRegistryException,
           RemoteException;

/**
 * Returns the securityNames of all the groups that contain the user
 *
 * This method is called by administrative console and scripting
```

```
 * (command line) to verify the user entered for RunAsRole mapping belongs
 * to that role  in the roles to user mapping. Initially, the check is done
 * to see if the role contains the user. If the role does not contain the user
 * explicitly, this method is called to get the groups that this user
 * belongs to so that checks are made on the groups that the role contains.
 *
 * @parameter userSecurityName the name of the user
 * @return a List of all the group securityNames that the user
 *   belongs to.
 * @exception EntryNotFoundException if user does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public List getGroupsForUser(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Gets a list of users in a group.
 *
 * The maximum number of users returned is defined by the limit
 * argument.
 *
 * This method is used by the WebSphere Business Integration
 * Server Foundation process choreographer when staff assignments
 * are modeled using groups.
 *

 * In rare situations where you are working with a user registry and it is not
 * practical to get all of the users from any of your groups (for example if
 * a large number of users exist) you can create the NotImplementedException
 * for those particular groups. Make sure that if the WebSphere Business
 * Integration Server Foundation Process Choreographer is installed (or
 * if installed later) that the users are not modeled using these particular groups.
 * If no concern exists about the staff assignments returning the users from
 * groups in the registry it is recommended that this method be implemented
 * without throwing the NotImplemented exception.
 *
 * @parameter groupSecurityName that represents the name of the group
 * @parameter limit the maximum number of users to return.
 *            This option is very useful in situations where lots of
 *            users are in the registry and getting all of them at
 *            once is not practical. A value of 0 means  get all of
 *            the users and must be used with care.
 * @return    a Result object that contains the list of users
 *            requested and a flag to indicate if more users exist.
 * @deprecated This method will be deprecated in the future.
 * @exception NotImplementedException create this exception in rare situations
 *            if it is not practical to get this information for any of the
 *            groups from the registry.
 * @exception EntryNotFoundException if the group does not exist in
 *            the registry
 * @exception CustomRegistryException if any registry-specific
 *            problem occurs
 * @exception RemoteException as this extends java.rmi.Remote interface
 **/
 public Result getUsersForGroup(String groupSecurityName, int limit)
    throws NotImplementedException,
           EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

```
  /**
   * This method is implemented internally by the WebSphere Application Server
   * code in this release. This method is not called for the custom registry
   * implementations for this release. Return null in the implementation.
   *
   * Note that because this method is not called you can also return the
   * NotImplementedException as the previous documentation says.
   *
   **/
   public com.ibm.websphere.security.cred.WSCredential
                                createCredential(String userSecurityName)
      throws NotImplementedException,
      EntryNotFoundException,
             CustomRegistryException,
             RemoteException;
}
```

Developing a custom SAF EJB role mapper

WebSphere Application Server for z/OS allows an installation to map Java Platform, Enterprise Edition (Java EE) role names to SAF EJBRole profile names.

Before you begin

WebSphere Application Server for z/OS supports the use of a custom SAF EJB role mapper. The custom SAF EJB role mapper allows an installation to map J2EE role names to SAF EJBRole profile names. Without the SAF EJB role mapper, you must deploy an application by using a role in the deployment descriptor of a component that is identical to the name of an EJBROLE class profile. The security administrator defines EJBROLE profiles and provides the permission to these profiles to SAF users or groups.

Using SAF EJBROLE class profiles can conflict with the standard Java EE role naming conventions. Java EE role names are Unicode strings of any length. RACF class profiles are restricted to 240 characters in length and cannot be defined if these profiles contain any white spaces or extended code page characters.

If a Java EE role name for an installation conflicts with these RACF restrictions, an installation can use the SAF EJB role mapper exit to map the desired Java EE role name to an acceptable class profile name.

The custom SAF role mapper is a Java-based exit to replace the EJBROLE class profile construction algorithm. The custom SAF role mapper is called to generate a profile for authorization and delegation requests. The role mapper passes the name of the application and the name of the role then passes back the appropriate class profile name. Information about the server name, cell name, and the SAF profile prefix (previously referred to as the z/OS security domain) is provided to the implementation during initialization.

You can set the com.ibm.websphere.security.SAF.RoleMapper custom property on the z/OS SAF authorization panel in the administrative console. You also can enable the role mapper by setting the custom property com.ibm.websphere.security.SAF.RoleMapper to the name of the class that is to be given control.

1. Build your custom SAF role mapper. The SAFRoleMapper example (below) can be used as a reference.

```
public class SAFRoleMapperImpl1 {
 String domainPrefix = null;

 public void initialize(Properties context) {
  domainPrefix = context.get(SAFRoleMapper.DOMAIN_NAME);
 }
```

```
 public String getProfileNameFromRole(String app, String role) {
  String profile = app + "." + role;
  if (domainPrefix != null) {
   profile = domainPrefix + "." + profile;
  }
  profile = profile.replaceAll("\\%", "#");
  profile = profile.replaceAll("\\&", "#");
  profile = profile.replaceAll("\\*", "#");
  profile = profile.replaceAll("\\s","#");

  return profile;
 }
}
```

2. Click **Security > Global security > z/OS SAF authorization** and enable the role mapper by providing the name of the class that you want to give control in the **SAF profile mapper** field. You also can set this property as a custom property by entering `com.ibm.websphere.security.SAF.RoleMapper` as the name and providing the name of the class in the value field.
3. Click **Security > Global security > External authorization providers** and select the **System Authorization Facility (SAF) authorization** option to enable SAF as the authorization provider. After you select this option, click **z/OS SAF authorization** under Related items to configure the SAF authorization options.

   You also can set this property as a custom property by entering `com.ibm.websphere.security.SAF.authorization` as the name and `true` as the value.

Implementing custom password encryption

WebSphere Application Server supports the use of custom password encryption.

Before you begin

An installation can implement any password encryption algorithm it chooses.

About this task

Complete the following steps to implement custom password encryption:

1. Build your custom password encryption class. An example of a custom password encryption class follows.

```
// CustomPasswordEncryption
// Encryption and decryption functions
public interface CustomPasswordEncryption {
    public EncryptedInfo encrypt(byte[] clearText) throws PasswordEncryptException;
    public byte[] decrypt(EncryptedInfo cipherTextInfo) throws PasswordEncryptException;
    public void initialize(HashMap initParameters);
};
// Encapsulation of cipher text and label
public class EncryptedInfo {
    public EncryptedInfo(byte[] bytes, String keyAlias);
    public byte[] getEncryptedBytes();
    public String getKeyAlias();
};
```

2. Enable custom password encryption.
   a. Set the custom property **com.ibm.wsspi.security.crypto.customPasswordEncryptionClass** to the name of the class that is to be given control.
   b. Enable the function. Set the custom property, **com.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled** to `true`.

Results

Custom password encryption at the installation is complete.

Developing applications that use programmatic security

For some applications, declarative security is not sufficient to express the security model of the application. Use this topic to develop applications that use programmatic security.

About this task

IBM WebSphere Application Server provides security components that provide or collaborate with other services to provide authentication, authorization, delegation, and data protection. WebSphere Application Server also supports the security features that are described in the Java Platform, Enterprise Edition (Java EE) specification. An application goes through three stages before it is ready to run:

- Development
- Assembly
- Deployment

Most of the security for an application is configured during the assembly stage. The security that is configured during the assembly stage is called *declarative security* because the security is *declared* or *defined* in the deployment descriptors. The declarative security is enforced by the security runtime. For some applications, declarative security is not sufficient to express the security model of the application. For these applications, you can use *programmatic security*.

1. Develop secure Web applications. For more information, see “Developing with programmatic security APIs for Web applications” on page 1324.
2. Develop servlet filters for form login processing. For more information, see “Developing servlet filters for form login processing” on page 1338.
3. Develop form login pages. For more information, see “Customizing Web application login” on page 1333.
4. Develop enterprise bean component applications. For more information, see “Developing with programmatic APIs for EJB applications” on page 1329.
5. Develop with Java Authentication and Authorization Service to log in programmatically. For more information, see Developing programmatic logins with the Java Authentication and Authorization Service.
6. Develop your own Java EE security mapping module. For more information, see Configuring programmatic logins for Java Authentication and Authorization Service.
7. Develop custom user registries. For more information, see “Developing standalone custom registries” on page 1290.
8. Develop a custom interceptor for trust associations.

Protecting system resources and APIs (Java 2 security)

Java 2 security is a programming model that is very pervasive and has a huge impact on application development.

Before you begin

Java 2 security is orthogonal to Java Platform, Enterprise Edition (Java EE) role-based security; you can disable or enable it independently of administrative security.

However, it does provide an extra level of access control protection on top of the Java EE role-based authorization. It particularly addresses the protection of system resources and application programming interfaces (API). Administrators need to consider the benefits against the risks of disabling Java 2 security.

The following recommendations are provided to help enable Java 2 security in a test or production environment:

1. Make sure the application is developed with the Java 2 security programming model. Developers have to know whether or not the APIs that are used in the applications are protected by Java 2 security. It is very important that the required permissions for the APIs used are declared in the policy file

(was.policy), or the application fails to run when Java 2 security is enabled. Developers can reference the Web site for Development Kit APIs that are protected by Java 2 security. See the Programming model and decisions section of the Security: Resources for learning topic to visit this Web site.

2. Make sure that migrated applications from previous releases are given the required permissions. Because Java 2 security is not supported or partially supported in previous WebSphere Application Server releases, applications developed prior to Version 5 most likely are not using the Java 2 security programming model. No easy way to find out all the required permissions for the application is available. The following are activities you can perform to determine the extra permissions that are required by an application:
   - Code review and code inspection
   - Application documentation review
   - Sandbox testing of migrated enterprise applications with Java 2 security enabled in a preproduction environment. Enable tracing in WebSphere Java 2 security manager to help determine the missing permissions in the application policy file. The trace specification is: com.ibm.ws.security.core.SecurityManager=all=enabled.
   - Use the com.ibm.websphere.java2secman.norethrow system property to aid debugging. Do not use this property in a production environment.

     Refer to Java 2 security

The default permission set for applications is the recommended permission set that is defined in the J2EE 1.3 Specification. The default is declared in the *app_server_root*/profiles/*profile_name*/config/cells/*cell_name*/nodes/*node_name*/app.policy policy file with permissions defined in the Development Kit (*JAVA_HOME*/jre/lib/security/java.policy) policy file that grant permissions to everyone. However, applications are denied permissions that are declared in the profiles/*profile_name*/config/cells/*cell_name*/filter.policy file. Permissions that are declared in the *filter.policy* file are filtered for applications during the permission check.

Define the required permissions for an application in a was.policy file and embed the was.policy file in the application enterprise archive (EAR) file as *YOURAPP.ear/META-INF/was.policy*, see “Configuring Java 2 security policy files” on page 1304 for details.

The following steps describe how to enforce Java 2 security on the cell level for WebSphere Application Server Network Deployment and the server level for WebSphere Application Server and WebSphere Application Server Express:

1. Click **Security > Global security**. The Global security panel is displayed.
2. Select the **Use Java 2 security to restrict application access to local resources** option.
3. Click **OK** or **Apply**.
4. Click **Save** to save the changes.
5. Restart the server for the changes to take effect.

Results

Java 2 security is enabled and enforced for the servers. Java 2 security permission is selected when a Java 2 security protected API is called.

When to use Java 2 security

1. Enable protection on system resources, for example when opening or listening to a socket connection, reading or writing to operating system file systems, reading or writing Java virtual machine system properties, and so on.
2. Prevent application code from calling destructive APIs, for example, calling the System.exit method brings down the application server.
3. Prevent application code from obtaining privileged information (passwords) or gaining extra privileges (obtaining server credentials).

What to do next

You can enforce Java 2 security on the server level for WebSphere Application Server Network Deployment by completing the following steps.

Note: Changes to Java 2 security settings on the server level override the settings on the cell level.

1. Click **Servers > Application servers >** *server_name*.
2. Under Security, click **Server security**.
3. Select the **Security settings for this server override cell settings** option.
4. Select the **Use Java 2 security to restrict application access to local resources** option.
5. Click **OK** or **Apply**.
6. Click **Save** to save the changes.
7. Restart the server for the changes to take effect.

The Java 2 security manager is enhanced to dump the Java 2 security permissions that are granted to all classes on the call stack when an application is denied access to a resource. The java.security.AccessControlException exception is created. However, this tracing capability is disabled by default. You can enable this capability by specifying the server trace service with the `com.ibm.ws.security.core.SecurityManager=all=enabled` trace specification. When the exception is created, the trace dump provides hints to determine whether the application is missing permissions or the product runtime code or the third-party libraries that are used are not properly marked as `privileged` when accessing Java 2 security-protected resources. See the Security Problem Determination Guide for details.

Using PolicyTool to edit policy files:

Use the **PolicyTool** utility to update policy files.

Before you begin

Java 2 security uses several policy files to determine the granted permission for each Java program. The Java Development Kit provides the **PolicyTool** tool to edit these policy files. This tool is recommended for editing any policy file to verify the syntax of its contents. Syntax errors in the policy file cause an AccessControlException exception when the application runs, including the server start. Identifying the cause of this exception is not easy because the user might not be familiar with the resource that has an access violation. Be careful when you edit these policy files.

See Java 2 security policy files for the list of available policy files.

To use the **PolicyTool** utility with WebSphere Application Server for z/OS, choose one of the following two options:

- Copy the policy files to another platform such as Microsoft Windows and modify the files. To use this option, you must issue the FTP command to transfer the files to the other platform, invoke the **PolicyTool**, and transfer the updated files back to the z/OS system in binary mode.
- Invoke the **PolicyTool** that is supplied with the Software Development Kit (SDK) installed on your z/OS system.

  1. Invoke the **PolicyTool** that is supplied with the Software Development Kit (SDK) installed on your z/OS system.
     a. Export the display to an Xwindows-enabled device. For example, in Open MVS (OMVS), type `export DISPLAY=<IP_address_of_the_Xwindows_device>:0.0`
     b. Enable the z/OS system to access the display of the Xwindows-enabled device. For example, on AIX systems, type `xhost + address_of_the_MVS_system`.
     c. Convert the policy file to the Extended Binary Coded Decimal Interchange Code (EBCDIC) format.

d. Invoke the **PolicyTool** on OMVS by typing $*JAVA_HOME*/policytool. The *JAVA_HOME* variable represents the directory in which the SDK is installed.

2. Click **File** > **Open**.
3. Navigate the directory tree in the **Open** window to pick up the policy file that you need to update. After selecting the policy file, click **Open**. The code base entries are listed in the window.
4. Create or modify the code base entry.
   a. Modify the existing code base entry by double-clicking the code base, or click the code base and click **Edit Policy Entry**. The Policy Entry window opens with the permission list defined for the selected code base.
   b. Create a new code base entry by clicking **Add Policy Entry**.

      The Policy Entry window opens. At the code base column, enter the code base information as a URL format.

      For example, you can enter:

      app_server_root/InstalledApps/testcase.ear

      where the *app_server_root* variable represents your installation location.
5. Modify or add the permission specification.
   a. Modify the permission specification by double-clicking the entry that you want to modify, or by selecting the permission and clicking **Edit Permission**. The Permissions window opens with the selected permission information.
   b. Add a new permission by clicking **Add Permission**. The Permissions window opens. In the Permissions window are four rows for Permission, Target Name, Actions, and Signed By.
6. Select the permission from the Permission list. The selected permission displays. After a permission is selected, the Target Name, Actions, and Signed By fields automatically show the valid choices or they enable text input in the right text input area.
   a. Select **Target Name** from the list, or enter the target name in the right text input area.
   b. Select **Actions** from the list.
   c. Input **Signed By** if it is needed.

      Note: The Signed By keyword is not supported in the following policy files: app.policy, spi.policy, library.policy, was.policy, and filter.policy files. However, the Signed By keyword is supported in the following policy files: #java.policy, server.policy, and client.policy files. The Java Authentication and Authorization Service (JAAS) is not supported in the app.policy, spi.policy, library.policy, was.policy, and filter.policy files. However, the JAAS principal keyword is supported in a JAAS policy file when it is specified by the java.security.auth.policy Java virtual machine (JVM) system property.
7. Click **OK** to close the Permissions window. Modified permission entries of the specified code base display.
8. Click **Done** to close the window. Modified code base entries are listed. Repeat the previous steps until you complete editing.
9. Click **File** > **Save** after you finish editing the file.
10. Convert the policy file back from the EBCDIC format to the ASCII format.

Results

A policy file is updated. If any policy files need editing, use the **PolicyTool** utility. Do not edit the policy file manually. Syntax errors in the policy files can potentially cause application servers or enterprise applications to not start or function incorrectly. For the changes in the updated policy file to take effect, restart the Java processes.

Configuring Java 2 security policy files:

Users can configure Java 2 security policy files so that the required permission is granted for the specified WebSphere Application Server enterprise application.

Before you begin

Java 2 security uses several policy files to determine the permissions for each Java programs.

Two types of policy files are supported by WebSphere Application Server: dynamic policy files and static policy files. Static policy files provide the default permissions. Dynamic policy files provide application permissions. Six dynamic policy files are provided:

Policy file name	Description
app.policy	Contains default permissions for all of the enterprise applications in the cell. **Note:** Updates to the app.policy file only apply to the enterprise applications on the node to which the app.policy file belongs.
was.policy	Contains application-specific permissions for an WebSphere Application Server enterprise application. This file is packaged in an enterprise archive (EAR) file.
ra.xml	Contains connector application specific permissions for a WebSphere Application Server enterprise application. This file is packaged in a resource adapter archive (RAR) file.
spi.policy	Contains permissions for Service Provider Interface (SPI) or third-party resources that are embedded in WebSphere Application Server. The default contents grant everything. Update this file carefully when the cell requires more protection against SPI in the cell. This file is applied to all of the SPIs that are defined in the resources.xml file.
library.policy	Contains permissions for the shared library of enterprise applications.
filter.policy	Contains the list of permissions that require filtering from the was.policy file and the app.policy file in the cell. This filtering mechanism only applies to the was.policy and app.policy files.

In WebSphere Application Server, applications must have the appropriate thread permissions specified in the was.policy or app.policy file. Without the thread permissions specified, the application cannot manipulate threads and WebSphere Application Server creates a java.security.AccessControlException exception. The app.policy file applies to a specified node. If you change the permissions in one app.policy file, you must incorporate the new thread policy in the same file on the remaining nodes. Also, if you add the thread permissions to the app.policy file, you must restart WebSphere Application Server to enforce the new permissions. However, if you add the permissions to the was.policy file for a specific application, you do not need to restart WebSphere Application Server. An administrator must add the following code to a was.policy or app.policy file for an application to manipulate threads:

```
grant codeBase "file:${application}" {
  permission java.lang.RuntimePermission "stopThread";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
};
```

Note: The Signed By keyword is not supported in the following policy files: app.policy, spi.policy, library.policy, was.policy, and filter.policy files. However, the Signed By keyword is supported in the following policy files:java.policy, server.policy, and client.policy files. The Java Authentication and Authorization Service (JAAS) is not supported in the app.policy, spi.policy, library.policy, was.policy, and filter.policy files. However, the JAAS principal keyword is supported in a JAAS policy file when it is specified by the java.security.auth.policy Java virtual machine (JVM) system property. You can statically set the authorization policy files in java.security.auth.policy with auth.policy.url.n=*URL*, where *URL* is the location of the authorization policy.

1. Identify the policy file to update.

- If the permission is required by an application, update the static policy file. Refer to “Configuring static policy files” on page 1318.
- If the permission is required by all of the WebSphere Application Server enterprise applications in the node, refer to “spi.policy file permissions” on page 1313.
- If the permission is required only by specific WebSphere Application Server enterprise applications and the permission is required only by connector, update the `ra.xml` file. Refer to “Assembling resource adapter (connector) modules” on page 1099. Otherwise, update the `was.policy` file. Refer to “Configuring the was.policy file” on page 1310 and “Adding the was.policy file to applications” on page 1316.
- If the permission is required by shared libraries, refer to “library.policy file permissions” on page 1314.
- If the permission is required by SPI libraries, refer to “spi.policy file permissions” on page 1313.

Note: Pick up the policy file with the smallest scope. You can avoid giving an extra permission to the Java programs and protect the resources. You can update the `ra.xml` file or the `was.policy` file rather than the `app.policy` file. Use specific component symbols ($(ejbcomponent), ${webComponent},${connectorComponent} and ${jars}) than ${application} symbols. Update dynamic policy files, rather than static policy files.

Add any permission that you never want granted to the WebSphere Application Server enterprise application in the cell to the `filter.policy` file. Refer to “filter.policy file permissions” on page 1309.

2. Restart the WebSphere Application Server enterprise application.

Results

The required permission is granted for the specified WebSphere Application Server enterprise application.

app.policy file permissions:

Java 2 security uses several policy files to determine the granted permissions for each Java program. The union of the permissions that are contained in these following files is applied to the WebSphere Application Server enterprise application. This union determines the granted permissions.

For the list of available policy files that are supported by WebSphere Application Server, see the Java 2 security policy files article The `app.policy` file is a default policy file that is shared by all of the WebSphere Application Server enterprise applications. The union of the permissions that are contained in the following files is applied to the WebSphere Application Server enterprise application:

- Any policy file that is specified in the policy.url.* properties in the `java.security` file.
- The `app.policy` files, which are managed by configuration and file replication services.
- The `server.policy` file.
- The `java.policy` file.
- The application `was.policy` file.
- The permission specification of the `ra.xml` file.
- The shared library, which is the `library.policy` file.

Changes made in these files are replicated to other nodes in the Network Deployment cell.

In WebSphere Application Server, applications that manipulate threads must have the appropriate thread permissions specified in the `was.policy` or `app.policy` file. Without the thread permissions specified, the application cannot manipulate threads and WebSphere Application Server creates a java.security.AccessControlException exception. If an administrator adds thread permissions to the `app.policy` file, the permission change requires a restart of the WebSphere Application Server. An administrator must add the following code to a `was.policy` or `app.policy` file for an application to manipulate threads:

```
grant codeBase "file:${application}" {
  permission java.lang.RuntimePermission "stopThread";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
};
```

Note: The Signed By and the Java Authentication and Authorization Service (JAAS) principal keywords are not supported in the `app.policy` file. However, the Signed By keyword is supported in the following files: `java.policy`, `server.policy`, and the `client.policy` files. The JAAS principal keyword is supported in a JAAS policy file when it is specified by the java.security.auth.policy Java virtual machine (JVM) system property. You can statically set the authorization policy files in the `java.security.auth.policy` property with `auth.policy.url.n=`*URL* where *URL* is the location of the authorization policy.

If the default permissions for enterprise applications (the union of the permissions that is defined in the `java.policy` file, the `server.policy` file and the `app.policy` file) are enough; no action is required. The default `app.policy` file is used automatically. If a specific change is required to all of the enterprise applications in the cell, update the `app.policy` file. Syntax errors in the policy files cause start failures in the application servers. Edit these policy files carefully.

Note: Updates to the `app.policy` file only apply to the enterprise applications on the node to which the `app.policy` file belongs.

To extract the policy file, use a command prompt to enter the following command on one line using the appropriate variable values for your environment:

wsadmin> set obj [$AdminConfig extract cells/*cell_name*/node/*node_name*/app.policy /temp/test/app.policy]

Edit the extracted `app.policy` file with the Policy Tool. For more information, see “Using PolicyTool to edit policy files” on page 1303. Changes to the `app.policy` file are local for the node.

To check in the policy file, use a command prompt to enter the following command on one line using the appropriate variable values for your environment:

wsadmin> $AdminConfig checkin cells/*cell_name*/nodes/*node_name*/app.policy temp/test/app.policy $obj

Several product-reserved symbols are defined to associate the permission lists to a specific type of resource.

Symbol	Meaning
file:${application}	Permissions apply to all resources within the application
file:${jars}	Permissions apply to all utility Java archive (JAR) files within the application
file:${ejbComponent}	Permissions apply to enterprise bean resources within the application
file:${webComponent}	Permissions apply to Web resources within the application
file:${connectorComponent}	Permissions apply to connector resources both within the application and within standalone connector resources.

Five embedded symbols are provided to specify the path and name for the java.io.FilePermission permission. These symbols enable flexible permission specifications. The absolute file path is fixed after the installation of the application.

Symbol	Meaning
${app.installed.path}	Path where the application is installed
${was.module.path}	Path where the module is installed

Symbol	Meaning
${current.cell.name}	Current cell name
${current.node.name}	Current node name
${current.server.name}	Current server name

Note: You cannot use the `${was.module.path}` in the `${application}` entry.

The `app.policy` file supplied by WebSphere Application Server is located in the *profile_root*`/config/cells/`*cell_name*`/nodes/`*node_name*`/app.policy`, which contains the following default permissions:

Note: In the following code sample, the first two lines that are related to java.io.FilePermission permission are split into two lines for illustrative purposes only.

```
grant codeBase "file:${application}" {
  // The following are required by JavaMail
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
};

grant codeBase "file:${jars}" {
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

grant codeBase "file:${connectorComponent}" {
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

grant codeBase "file:${webComponent}" {
  permission java.io.FilePermission "${was.module.path}${/}-", "read, write";
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

grant codeBase "file:${ejbComponent}" {
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};
```

If all of the WebSphere Application Server enterprise applications in a cell require permissions that are not defined as defaults in the `java.policy` file, the `server.policy` file and the `app.policy` file, then update the `app.policy` file. The symptom of a missing permission is the java.security.AccessControlException exception.

Note: Updates to the `app.policy` file only apply to the enterprise applications on the node to which the `app.policy` file belongs.

When a Java program receives this exception and adding this permission is justified, add a permission to the `server.policy` file, for example:

The previous permission information lines are split for the illustration. You actually enter the permission on one line.

To decide whether to add a permission, refer to the AccessControlException topic.

Restart all WebSphere Application Server enterprise applications to ensure that the updated `app.policy` file takes effect.

filter.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program. Java 2 security policy filtering is only in effect when Java 2 security is enabled.

Before modifying the `filter.policy` file, you must start the wsadmin tool. See the Starting the wsadmin scripting client article for more information.

Refer to “Protecting system resources and APIs (Java 2 security)” on page 1301. The filtering policy defined in the `filter.policy` file is cell wide. The `filter.policy` file is the only policy file that is used when restricting the permission instead of granting permission. The permissions that are listed in the filter policy file are filtered out from the `app.policy` file and the `was.policy` file. Permissions that are defined in the other policy files are not affected by the `filter.policy` file.

When a permission is filtered out, an audit message is logged. However, if the permissions that are defined in the `app.policy` file and the `was.policy` file are compound permissions like the java.security.AllPermission permission, for example, the permission is not removed. A warning message is logged. If the Issue Permission Warning flag is enabled (default) and if the `app.policy` file and the `was.policy` file contain custom permissions (non-Java API permission, the permission package name begins with characters other than `java` or `javax`), a warning message is logged and the permission is not removed. You can change the value of the **Warn if applications are granted custom permissions** option on the Global security panel. It is not recommended that you use the `AllPermission` permission for the enterprise application.

Some default permissions that are defined in the `filter.policy` file. These permissions are the minimal ones that are recommended by the product. If more permissions are added to the `filter.policy` file, certain operations can fail for enterprise applications. Add permissions to the `filter.policy` file carefully.

You cannot use the Policy Tool to edit the `filter.policy` file. Editing must be completed in a text editor. Be careful and verify that no syntax errors exist in the `filter.policy` file. If any syntax errors exist in the `filter.policy` file, the file is not loaded by the product security runtime, which implies that filtering is disabled.

To extract the `filter.policy` file, enter the following command using information from your environment:

set obj [$AdminConfig extract cells/*cell_name*/filter.policy /temp/test/filter.policy]

To check in the policy file, enter the following command using information from your environment:

$AdminConfig checkin cells/*cell_name*/filter.policy /temp/test/filter.policy $obj

An updated `filter.policy` file is applied to all of the WebSphere Application Server enterprise applications after the servers are restarted. The `filter.policy` file is managed by configuration and file replication services.

Changes made in the file are replicated to other nodes in the cell.

The `filter.policy` file that is supplied by WebSphere Application Server resides at: *app_server_root*/`profiles/`*`profile_name`*`/config/cells/`*`cell_name`*`/filter.policy`.

This fill contains these permissions as defaults:

```
filterMask {
permission java.lang.RuntimePermission "exitVM";
permission java.lang.RuntimePermission "setSecurityManager";
permission java.security.SecurityPermission "setPolicy";
```

```
permission javax.security.auth.AuthPermission "setLoginConfiguration"; };
runtimeFilterMask {
permission java.lang.RuntimePermission "exitVM";
permission java.lang.RuntimePermission "setSecurityManager";
permission java.security.SecurityPermission "setPolicy";
permission javax.security.auth.AuthPermission "setLoginConfiguration"; };
```

The permissions that are defined in `filterMask` filter are for static policy filtering. The security runtime tries to remove the permissions from applications during application startup. Compound permissions are not removed, but are issued with a warning, and application deployment is stopped if applications contain permissions that are defined in the `filterMask` filter, and if scripting is used. The runtimeFilterMask filter defines permissions that are used by the security runtime to deny access to those permissions to application thread. Do not add more permissions to the runtimeFilterMask filter. Application start failure or incorrect functioning might result. Be careful when adding more permissions to the runtimeFilterMask filter. Usually, you only need to add permissions to the `filterMask` stanza.

WebSphere Application Server relies on the filter policy file to restrict or disallow certain permissions that can compromise the integrity of the system. For instance, WebSphere Application Server considers the exitVM and setSecurityManager permissions as those permissions that most applications never have. If these permissions are granted, the following scenarios are possible:

exitVM
: A servlet, JavaServer Pages (JSP) file, enterprise bean, or other library that is used by the aforementioned might call the System.exit API and cause the entire WebSphere Application Server process to terminate.

setSecurityManager
: An application might install its own security manager and either grant more permissions or bypass the default policy that the WebSphere Application Server security manager enforces.

Note: In application code, do not use the setSecurityManager permission to set a security manager. When an application uses the setSecurityManager permission, a conflict exists with the internal security manager within WebSphere Application Server. If you must set a security manager in an application for Remote Method Invocation (RMI) purposes, you also must select the **Use Java 2 security to restrict application access to local resources** option on the Global security panel within the WebSphere Application Server administrative console. WebSphere Application Server then registers a security manager, which the application code can verify is registered by using the System.getSecurityManager application programming interface (API).

For the updated `filter.policy` file to take effect, restart related Java processes.

Configuring the was.policy file:

You should update the `was.policy` file if the application has specific resources to access.

Before you begin

Java 2 security uses several policy files to determine the granted permission for each Java program. The `was.policy` file is an application-specific policy file for WebSphere Application Server enterprise applications. This file is embedded in the `META-INF/was.policy` enterprise archive (.EAR) file. The `was.policy` file is located in:

profile_root`/config/cells/`*cell_name*`/applications/`
ear_file_name`/deployments/`*application_name*`/META-INF/was.policy`

See Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server Version 6.1.

The union of the permissions that are contained in the following files is applied to the WebSphere Application Server enterprise application:

- Any policy file that is specified in the policy.url.* properties in the `java.security` file.
- The `app.policy` files, which are managed by configuration and file replication services.
- The `server.policy` file.
- The `java.policy` file.
- The application `was.policy` file.
- The permission specification of the `ra.xml` file.
- The shared library, which is the `library.policy` file.

Changes made in these files are replicated to other nodes in the cell.

Several product-reserved symbols are defined to associate the permission lists to a specific type of resources.

Symbol	Definition
file:${application}	Permissions apply to all resources used within the application.
file:${jars}	Permissions apply to all utility Java archive (JAR) files within the application
file:${ejbComponent}	Permissions apply to enterprise bean resources within the application
file:${webComponent}	Permissions apply to Web resources within the application
file:${connectorComponent}	Permissions apply to connector resources within the application

In WebSphere Application Server, applications that manipulate threads must have the appropriate thread permissions specified in the `was.policy` or `app.policy` file. Without the thread permissions specified, the application cannot manipulate threads and WebSphere Application Server creates a java.security.AccessControlException exception. If you add the permissions to the `was.policy` file for a specific application, you do not need to restart WebSphere Application Server. An administrator must add the following code to a `was.policy` or `app.policy` file for an application to manipulate threads:

```
grant codeBase "file:${application}" {
  permission java.lang.RuntimePermission "stopThread";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
};
```

An administrator can add the thread permissions to the `app.policy` file, but the permission change requires a restart of WebSphere Application Server.

Note: The Signed By and the Java Authentication and Authorization Service (JAAS) principal keywords are not supported in the `was.policy` file. The Signed By keyword is supported in the `java.policy`, `server.policy`, and `client.policy` policy file. The JAAS principal keyword is supported in a JAAS policy file when it is specified by the `java.security.auth.policy` Java virtual machine (JVM) system property. You can statically set the authorization policy files in the `java.security.auth.policy` file with the `auth.policy.url.n=`*URL*, where *URL* is the location of the authorization policy.

Other than these blocks, you can specify the module name for granular settings. For example,

```
grant codeBase "file:DefaultWebApplication.war" {
  permission java.security.SecurityPermission "printIdentity";
};

grant codeBase "file:IncCMP11.jar" {
  permission java.io.FilePermission
      "${user.install.root}${/}bin${/}DefaultDB${/}-",
      "read,write,delete";
};
```

Five embedded symbols are provided to specify the path and name for the java.io.FilePermission permission. These symbols enable flexible permission specification. The absolute file path is fixed after the application is installed.

Symbol	Definition
${app.installed.path}	Path where the application is installed
${was.module.path}	Path where the module is installed
${current.cell.name}	Current cell name
${current.node.name}	Current node name
${current.server.name}	Current server name

About this task

If the default permissions for the enterprise application are enough, an action is not required. The default permissions are a union of the permissions that are defined in the `java.policy` file, the `server.policy` file, and the `app.policy` file. If an application has specific resources to access, update the `was.policy` file. The first two steps assume that you are creating a new policy file.

Note: Syntax errors in the policy files cause the application server to fail. Use care when editing these policy files.

1. Create or edit a new `was.policy` file by using the PolicyTool. For more information, see “Using PolicyTool to edit policy files” on page 1303.
2. Package the `was.policy` file into the enterprise archive (EAR) file.

   For more information, see “Adding the was.policy file to applications” on page 1316.The following instructions describe how to import a `was.policy` file.

   a. Import the EAR file into an assembly tool.
   b. Open the Project Navigator view.
   c. Expand the EAR file and click **META-INF**. You might find a `was.policy` file in the META-INF directory. If you want to delete the file, right-click the file name and select **Delete**.
   d. At the bottom of the Project Navigator view, click **J2EE Hierarchy**.
   e. Import the `was.policy` file by right-clicking the **Modules** directory within the deployment descriptor and by clicking **Import > Import > File system**.
   f. Click **Next**.
   g. Enter the path name to the `was.policy` file in the **From directory** field or click **Browse** to locate the file.
   h. Verify that the path directory that is listed in the **Into directory** field lists the correct `META-INF` directory.
   i. Click **Finish**.
   j. To validate the EAR file, right-click the EAR file, which contains the Modules directory, and click **Run Validation**.

k. To save the new EAR file, right-click the EAR file, and click **Export > Export EAR file**. If you do not save the revised EAR file, the EAR file will contain the new `was.policy` file. However, if the workspace becomes corrupted, you might lose the revised EAR file.

l. To generate deployment code, right-click the EAR file and click **Generate Deployment Code**.

3. Update an existing installed application, if one already exists. Modify the `was.policy` file with the Policy Tool. For more information, see "Using PolicyTool to edit policy files" on page 1303.

   a. Extract the policy file. Enter the following command from a command prompt:

   ```
   wsadmin> set obj [$AdminConfig extract profiles/profile_name/cells/cell_name
   /application/ear_file_name/deployments/application_name
   /META_INF/was.policy c:/temp/test/was.policy]
   ```

   Enter the three previous lines as one continuous line. They are display here for illustration only.

   b. Edit the extracted `was.policy` file with the Policy Tool. For more information, see "Using PolicyTool to edit policy files" on page 1303.

   c. Check in the policy file. Enter the following at a command prompt:

   ```
   wsadmin> $AdminConfig checkin profiles/profile_name/cells/cell_name/application/
   ear_file_name/deployments/application_name/META_INF/was.policy
   c:/temp/test/was.policy $obj
   ```

   Enter the three previous lines as one continuous line. They are display here for illustration only.

Results

The updated `was.policy` file is applied to the application after the application restarts.

Example

```
java.security.AccessControlException: access denied (java.io.FilePermission
${was.install.root}/java/ext/mail.jar read)
```

If an application must access a specific resource that is not defined as a default in the `java.policy` file, the `server.policy` file, and the `app.policy`, delete the `was.policy` file for that application. The symptom of the missing permission is the java.security.AccessControlException exception. The missing permission is listed in the exception data:

Note: Examples that appear below are split into several lines for illustration only. You actually enter the permission on one line.

```
java.security.AccessControlException: access denied (java.io.FilePermission
${was.install.root}/java/ext/mail.jar read)
```

When a Java program receives this exception and adding this permission is justified, add the following permission to the `was.policy` file:

```
grant codeBase "file:user_client_installed_location" {
  permission java.io.FilePermission
"${was.install.root}$(/)java$(/)jre$(/)lib$(/)ext$(/)mail.jar", "read";
};
```

To determine whether to add a permission, see Access control exception.

What to do next

Restart all applications for the updated `app.policy` file to take effect.

spi.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

For the list of available policy files that are supported by WebSphere Application Server Version 6.0.x, see Java 2 security policy files.

Because the default permission for the Service Provider Interface (SPI) is the AllPermission permission, the only reason to update the `spi.policy` file is a restricted SPI permission. When a change in the `spi.policy` is required, complete the following steps.

Syntax errors in the policy files cause the application server to fail. Edit these policy files carefully.

Note: Do not place the codebase keyword or any other keyword after the filterMask and runtimeFilterMask keywords. The Signed By and the Java Authentication and Authorization Service (JAAS) Principal keywords are not supported in the `spi.policy` file. The Signed By keyword is supported in the `java.policy`, `server.policy`, and `client.policy` policy files. The JAAS Principal keyword is supported in a JAAS policy file that is specified by the java.security.auth.policy Java virtual machine (JVM) system property. You can statically set the authorization policy files in `java.security.auth.policy` with `auth.policy.url.n=`*`URL`*`,` where *`URL`* is the location of the authorization policy.

To extract the `filter.policy` file, enter the following command using information from your environment:

set obj [$AdminConfig extract profiles/*profile_name*/cells/*cell_name*/nodes/*node_name*/spi.policy c:/temp/test/spi.policy]

Edit the file using the Policy Tool. For more information, see “Using PolicyTool to edit policy files” on page 1303.

To check in the policy file, enter the following command using information from your environment:

The updated `spi.policy` is applied to the Service Provider Interface (SPI) libraries after the Java process is restarted.

$AdminConfig checkin profiles/*profile_name*/cells/*cell_name*/nodes/*node_name*/spi.policy c:/temp/test/spi.policy $obj

Examples

The `spi.policy` file is the template for SPIs or third-party resources embedded in the product. Examples of SPIs are Java Message Services (JMS) (MQSeries) and Java database connectivity (JDBC) drivers. They are specified in the `resources.xml` file. The dynamic policy grants the permissions that are defined in the `spi.policy` file to the class paths defined in the `resources.xml` file. The union of the permission that is contained in the `java.policy` file and the `spi.policy` file are applied to the SPI libraries. The `spi.policy` files are managed by configuration and file replication services.

Changes made in these files are replicated to other nodes in the cell.

You can find the `spi.policy` file that is supplied by WebSphere Application Server in the following location: *app_server_root*`/profiles/`*`profile_name`*`/config/cells/`*`cell_name`*`/nodes/`*`node_name`*`/spi.policy`. This file contains the following default permission:

```
grant {
  permission java.security.AllPermission;
};
```

Restart the related Java processes for the changes in the `spi.policy` file to become effective.

library.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

For the list of available policy files that are supported by WebSphere Application Server, see Java 2 security policy files.

The `library.policy` file is the template for shared libraries (Java library classes). Multiple enterprise applications can define and use shared libraries. Refer to Managing shared libraries for information on how to define and manage the shared libraries.

If the default permissions for a shared library (union of the permissions defined in the `java.policy` file, the `app.policy` file and the `library.policy` file) are enough, no action is required. The default library policy is picked up automatically. If a specific change is required to share a library in the cell, update the `library.policy` file.

Syntax errors in the policy files cause the application server to fail. Edit these policy files carefully.

Note: Do not place the codebase keyword or any other keyword after the grant keyword. The Signed By keyword and the Java Authentication and Authorization Service (JAAS) Principal keyword are not supported in the `library.policy` file. The Signed By keyword is supported in the `java.policy`, the `server.policy`, and the `client.policy` policy files. The JAAS Principal keyword is supported in a JAAS policy file when it is specified by the Java virtual machine (JVM) system property, java.security.auth.policy. You can statically set the authorization policy files in the `java.security.auth.policy` file with `auth.policy.url.n=`*`URL`* `where` *`URL`* is the location of the authorization policy.

To extract the policy file, use a command prompt to enter the following command using the appropriate variable values for your environment:The previous two lines were split onto two lines for illustrative purposes only.

wsadmin> set obj [$AdminConfig extract cells/*cell_name*/nodes/
node_name/library.policy /temp/test/library.policy]

Edit the extracted `library.policy` file with the Policy Tool. For more information, see “Using PolicyTool to edit policy files” on page 1303.

To check in the policy file, use a command prompt to enter the following command using the appropriate variable values for your environment:An updated `library.policy` is applied to shared libraries after the servers restart.

wsadmin> $AdminConfig checkin cells/*cell_name*/nodes/*node_name*/library.policy
temp/test/library.policy $obj

Example

The union of the permission that is contained in the `java.policy` file, the `app.policy` file, and the `library.policy` file are applied to the shared libraries. The `library.policy` file is managed by configuration and file replication services.

Changes made in the file are replicated to other nodes in the cell.

The `library.policy` file are supplied by WebSphere Application Server resides at: *app_server_root*/`config/cells/`*`cell_name`*`/nodes/`*`node_name`*`/` directory. The file contains an empty permission entry as a default. For example:

```
grant {
  };
```

If the shared library in a cell requires permissions that are not defined as defaults in the java.policy file, the app.policy file and the library.policy file, update the library.policy file. The missing permission causes the java.security.AccessControlException exception. The missing permission is listed in the exception data.

When a Java program receives this exception and adding this permission is justified, add a permission to the library.policy file.

To decide whether to add a permission, refer to Access control exception.

Restart the related Java processes for the changes in the library.policy file to become effective.

Adding the was.policy file to applications:

An application might need a was.policy file if it accesses resources that require more permissions than those granted in the default app.policy file.

About this task

When Java 2 security is enabled for a WebSphere Application Server, all the applications that run on WebSphere Application Server undergo a security check before accessing system resources. An application might need a was.policy file if it accesses resources that require more permissions than those granted in the default app.policy file. By default, the product security reads an app.policy file that is located in each node and grants the permissions in the app.policy file to all the applications. Include any additional required permissions in the was.policy file. The was.policy file is only required if an application requires additional permissions.

The default policy file for all applications is specified in the app.policy file. This file is provided by the product security, is common to all applications, and you do not change this file. Add any new permissions that are required for an application in the was.policy file.

The app.policy file supplied by WebSphere Application Server resides at *app_server_root*/config/cells/profile/profile_name/config/*cell_name*/nodes/*node_name*/app.policy. The contents of the app.policy file are presented in the following example:

Note: In the following code sample, the two permissions that are required by JavaMail are split onto two lines for illustration only. You actually enter the permission on one line.

```
// The following permissions apply to all the components under the application.

grant codeBase "file:${application}" {
   // The following are required by JavaMail

  permission java.io.FilePermission "
        ${was.install.root}${/}lib${/}activation-impl.jar",
"read";

  permission java.io.FilePermission "
        ${was.install.root}${/}lib${/}mail-impl.jar","read";

  };
   // The following permissions apply to all utility .jar files  (other
   //  than enterprise beans JAR files) in the application.
grant codeBase "file:${jars}" {
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

// The following permissions apply to connector resources within the application
grant codeBase "file:${connectorComponent}" {
  permission java.net.SocketPermission "*", "connect";
```

```
  permission java.util.PropertyPermission "*", "read";
};

// The following permissions apply to all the Web modules (.war files)
// within the application.
grant codeBase "file:${webComponent}" {
  permission java.io.FilePermission "${was.module.path}${/}-", "read, write";
       //  where "was.module.path" is the path where the Web module is
       //  installed. Refer to Dynamic policy concepts for other symbols.
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

// The following permissions apply to all the EJB modules within the application.
grant codeBase "file:${ejbComponent}" {
 permission java.lang.RuntimePermission "queuePrintJob";
 permission java.net.SocketPermission "*", "connect";
 permission java.util.PropertyPermission "*", "read";
};
```

If additional permissions are required for an application or for one or more modules of an application, use the `was.policy` file for that application. For example, use codeBase of ${application} and add required permissions to grant additional permissions to the entire application. Similarly, use codeBase of ${webComponent} and ${ejbComponent} to grant additional permissions to all the Web modules and all the enterprise bean modules in the application. You can assign additional permissions to each module (`.war` file or `.jar` file), as shown in the following example.

This example illustrates adding extra permissions for an application in the `was.policy` file:

Note: In the following code sample, the permission for the EJB module was split onto two lines for illustration only. You actually enter the permission on one line.

```
// grant additional permissions to a Web module
grant codeBase " file:aWebModule.war" {
 permission java.security.SecurityPermission "printIdentity";
};

// grant additional permission to an EJB module
grant codeBase "file:aEJBModule.jar"  {
    permission java.io.FilePermission "
       ${user.install.root}${/}bin${/}DefaultDB${/}-" ."read.write,delete";
    // where, ${user.install.root} is the system property whose value is
    // located in the app_server_root directory.
 };
```

To use a `was.policy` file for your application, perform the following steps:

1. Create a `was.policy` file using the policy tool. For more information on using the policy tool, see “Using PolicyTool to edit policy files” on page 1303.
2. Add the required permissions in the `was.policy` file using the policy tool.
3. Place the `was.policy` file in the application enterprise archive (EAR) file under the META-INF directory. Update the application EAR file with the newly created `was.policy` file by using the **jar** command.
4. Verify that the `was.policy` file is inserted and start an assembly tool.

   Note: An assembly tool is not available. Use an assembly tool on another platform such as Linux Intel® or Windows.
5. Verify that the `was.policy` file in the application is syntactically correct. In an assembly tool, right-click the enterprise application module and click **Run Validation**.

Results

An application EAR file is now ready to run when Java 2 security is enabled.

Example

This step is required for applications to run properly when Java 2 security is enabled. If the `was.policy` file is not created and it does not contain required permissions, the application might not access system resources.

The symptom of the missing permissions is the java.security.AccessControlException exception. The missing permission is listed in the exception data, for example,

```
java.security.AccessControlException: access denied (java.io.FilePermission
${was.install.root}/java/ext/mail.jar read)
```

When an application program receives this exception and adding this permission is justified, include the permission in the `was.policy` file, for example,

```
grant codeBase "file:user_client_installed_location" {
  permission java.io.FilePermission
"${was.install.root}$(/)java$(/)jre$(/)lib$(/)ext$(/)mail.jar", "read";
};
```

The previous permission information lines are split for the illustration. Enter the permission on one line.

What to do next

Install the application.

Configuring static policy files:

By configuring the static policy files, the required permission will be granted for all of the Java programs.

Before you begin

Java 2 security uses several policy files to determine the granted permission for each Java program.

See the Java 2 security policy files topic for the list of available policy files that are supported by WebSphere Application Server.

Two types of policy files are supported by WebSphere Application Server: dynamic policy files and static policy files. Static policy files provide the default permissions. Dynamic policy files provide application permissions.

Policy file name	Description
`java.policy`	Contains default permissions for all of the Java programs on the node. This file seldom changes.
`server.policy`	Contains default permissions for all of the WebSphere Application Server programs on the node. This file is rarely updated.
`client.policy`	Contains default permissions for all of the applets and client containers on the node.

The static policy file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not get replicated to the other machine.

1. Identify the policy file to update.
   - If the permission is required only by an application, update the dynamic policy file. Refer to “Configuring Java 2 security policy files” on page 1304.

- If the permission is required only by applets and client containers, update the `client.policy` file. Refer to “client.policy file permissions” on page 1322.
- If the permission is required only by WebSphere Application Server (servers, agents, managers and application servers), update the `server.policy` file. Refer to “server.policy file permissions” on page 1321.
- If the permission is required by all of the Java programs running on the Java virtual machine (JVM), update the `java.policy` file. Refer to “java.policy file permissions.”

2. Stop and restart WebSphere Application Server.

Results

The required permission is granted for all of the Java programs that run with the restarted JVM.

Example

If Java programs on a node require permissions, the policy file needs updating. If the Java program that required the permission is not part of an enterprise application, update the static policy file. The missing permission results in the creation of the java.security.AccessControlException exception. The missing permission is listed in the exception data.

For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:/WAS_HOME/lib/mail-impl.jar read)
```

When a Java program receives this exception and adding this permission is justified, add a permission to an adequate policy file.

For example:

```
grant codeBase "file:user_client_installed_location" {
  permission java.io.FilePermission
  "C:/WAS_HOME/lib/mail-impl.jar",
  "read";
};
```

To decide whether to add a permission, refer to Access control exception.

java.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

See Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server.

The `java.policy` file is a global default policy file that is shared by all of the Java programs that run in the Java virtual machine (JVM) on the node. A change to the `java.policy` file is local for the node. The default Java policy is picked up automatically. Syntax errors in the policy files cause the application server to fail. An updated `java.policy` file is applied to all the Java programs that run in all the JVMs on the local node. Restart the programs for the updates to take effect. Modifying this file is not recommended. If a specific change is required to some of the Java programs on a node and the `java.policy` file requires updating, carefully modify the `java.policy` file with the policy tool. For more information, see “Using PolicyTool to edit policy files” on page 1303.

Default permissions for the `java.policy` file

The `java.policy` file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not get replicated to the other machine. The `java.policy` file

that is supplied by WebSphere Application Server is located at *install_root*/java/jre/lib/security/java.policy. This file contains these default permissions.

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
        permission java.security.AllPermission;
};
// default permissions granted to all domains
grant {
        // Allows any thread to stop itself using the java.lang.Thread.stop()
        // method that takes no argument.
        // Note that this permission is granted by default only to remain
        // backwards compatible.
        // It is strongly recommended that you either remove this permission
        // from this policy file or further restrict it to code sources
        // that you specify, because Thread.stop() is potentially unsafe.
        // See "http://java.sun.com/notes" for more information.
        // permission java.lang.RuntimePermission "stopThread";

        // allows anyone to listen on un-privileged ports
        permission java.net.SocketPermission "localhost:1024-", "listen";

        // "standard" properties that can be read by anyone

        permission java.util.PropertyPermission "java.version", "read";
        permission java.util.PropertyPermission "java.vendor", "read";
        permission java.util.PropertyPermission "java.vendor.url", "read";
        permission java.util.PropertyPermission "java.class.version", "read";
        permission java.util.PropertyPermission "os.name", "read";
        permission java.util.PropertyPermission "os.version", "read";
        permission java.util.PropertyPermission "os.arch", "read";
        permission java.util.PropertyPermission "file.separator", "read";
        permission java.util.PropertyPermission "path.separator", "read";
        permission java.util.PropertyPermission "line.separator", "read";

        permission java.util.PropertyPermission "java.specification.version", "read";
        permission java.util.PropertyPermission "java.specification.vendor", "read";
        permission java.util.PropertyPermission "java.specification.name", "read";

        permission java.util.PropertyPermission "java.vm.specification.version","read";
        permission java.util.PropertyPermission "java.vm.specification.vendor","read";
        permission java.util.PropertyPermission "java.vm.specification.name", "read";
        permission java.util.PropertyPermission "java.vm.version", "read";
        permission java.util.PropertyPermission "java.vm.vendor", "read";
        permission java.util.PropertyPermission "java.vm.name", "read";
       };
```

If some Java programs on a node require permissions that are not defined as defaults in the `java.policy` file, consider updating the `java.policy` file. Most of the time, other policy files are updated instead of the `java.policy` file. The missing permission causes the creation of the , java.security.AccessControlException exception. The missing permission is listed in the exception data.

For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar read)
```

The previous two lines are one continuous line.

When a Java program receives this exception and adding this permission is justified, add a permission to the `java.policy` file.

For example:

```
grant codeBase "file:user_client_installed_location" {
permission java.io.FilePermission
"C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar", "read"; };
```

To decide whether to add a permission, refer to Access control exception.

Restart all of the Java processes for the updated `java.policy` file to take effect.

server.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

The `server.policy` file is a default policy file that is shared by all of the WebSphere Application Servers on a node. The `server.policy` file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not replicate to the other machine.

If the default permissions for a server (the union of the permissions that is defined in the `java.policy` file and the `server.policy` file) are enough, no action is required. The default server policy is picked up automatically. If a specific change is required to some of the server programs on a node, update the `server.policy` file with the Policy Tool. Refer to the “Using PolicyTool to edit policy files” on page 1303 topic to edit policy files. Changes to the `server.policy` file are local for the node. Syntax errors in the policy files cause the application server to fail. Edit these policy files carefully. An updated `server.policy` file is applied to all the server programs on the local node. Restart the servers for the updates to take effect.

If you want to add permissions to an application, use the `app.policy` file and the `was.policy` file.

Note: Updates to the `app.policy` file only apply to the enterprise applications on the node to which the `app.policy` file belongs.

When you do need to modify the `server.policy` file, locate this file at: *profile_root*`/properties/server.policy`. This file contains these default permissions:

```
// Allow to use sun tools
grant codeBase "file:${java.home}/lib/tools.jar" {
permission java.security.AllPermission;
};


// Allow the WebSphere deploy tool all permissions
grant codeBase "file:${was.install.root}/deploytool/-" {
permission java.security.AllPermission;
};


grant codeBase "file:${was.install.root}/plugins/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${was.install.root}/classes/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${was.install.root}/lib/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${smpe.install.root}/lib/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${smpe.install.root}/-" {
```

```
permission java.security.AllPermission;
};

// Allow to use TAM
grant codeBase "file:${was.install.root}/java/jre/lib/ext/PD.jar" {
permission java.security.AllPermission;
};
```

If some server programs on a node require permissions that are not defined as defaults in the `server.policy` file and the `server.policy` file, update the `server.policy` file. The missing permission creates the java.security.AccessControlException exception. The missing permission is listed in the exception data.

For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\WebSphere\AppServer\java\jre\lib\ext\mail-impl.jar read)
```

The previous two lines are split into two lines for illustrative purposes only.

When a Java program receives this exception and adding this permission is justified, add a permission to the `server.policy` file.

For example:

```
grant codeBase "file:user_client_installed_location" {
permission java.io.FilePermission
"C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar", "read"; };
```

To decide whether to add a permission, refer to Access control exception.

Restart all of the Java processes for the updated `server.policy` file to take effect.

client.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

For the list of available policy files that are supported by WebSphere Application Server, see Java 2 security policy files.

- The `client.policy` file is a default policy file that is shared by all of the WebSphere Application Server client containers and applets on a node.
- The union of the permissions that is contained in the `java.policy` file and the `client.policy` file are given to all of the client containers for WebSphere Application Server and applets running on the node.
- The `client.policy` file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not replicate to the other machine.
- The `client.policy` file supplied by WebSphere Application Server is located in the *profile_root*`/properties/client.policy`.
- If the default permissions for a client (union of the permissions defined in the `java.policy` file and the `client.policy` file) are enough, no action is required. The default client policy is picked up automatically.
- If a specific change is required to some of the client containers and applets on a node, modify the `client.policy` file with the Policy Tool. Refer to “Using PolicyTool to edit policy files” on page 1303, to edit policy files. Changes to the `client.policy` file are local for the node.

This file contains these default permissions:

```
grant codeBase "file:${was.install.root}/java/ext/*" {
  permission java.security.AllPermission;
};
```

```
// JDK classes
grant codeBase "file:${was.install.root}/java/ext/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/java/tools/ibmtools.jar" {
  permission java.security.AllPermission;
};
grant codeBase "file:/QIBM/ProdData/Java400/jdk14/lib/tools.jar" {
  permission java.security.AllPermission;
};

// WebSphere system classes
grant codeBase "file:${was.install.root}/lib/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/plugins/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/classes/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/installedConnectors/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${user.install.root}/installedConnectors/-" {
  permission java.security.AllPermission;
};

grant codeBase "file:${was.install.root}/installedChannels/-" {
  permission java.security.AllPermission;
};

// J2EE 1.4 permissions for client container WebSphere Application Server applications
// in $WAS_HOME/installedApps
grant codeBase "file:${user.install.root}/installedApps/-" {
  //Application client permissions
  permission java.awt.AWTPermission "accessClipboard";
  permission java.awt.AWTPermission "accessEventQueue";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission java.lang.RuntimePermission "exitVM";
  permission java.lang.RuntimePermission "loadLibrary";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.net.SocketPermission "localhost:1024-", "accept,listen";
  permission java.io.FilePermission "*", "read,write";
  permission java.util.PropertyPermission "*", "read";

};

// J2EE 1.4 permissions for client container - expanded ear file code base
grant codeBase "file:${com.ibm.websphere.client.applicationclient.archivedir}/-" {
  permission java.awt.AWTPermission "accessClipboard";
  permission java.awt.AWTPermission "accessEventQueue";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission java.lang.RuntimePermission "exitVM";
  permission java.lang.RuntimePermission "loadLibrary";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.net.SocketPermission "localhost:1024-", "accept,listen";
  permission java.io.FilePermission "*", "read,write";
  permission java.util.PropertyPermission "*", "read";
};
```

All of the client containers and applets on the local node are granted the updated permissions when they start. If some client containers or applets on a node require permissions that are not defined as defaults in

the `java.policy` file and the default `client.policy` file, update the `client.policy` file. The missing permission creates the java.security.AccessControlException exception. The missing permission is listed in the exception data, for example,

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar read)
```

The previous two lines of sample code are one continuous line, but presented as such for illustrative purposes only.

When a client program receives this exception and adding this permission is justified, add a permission to the `client.policy` file, for example, grant codebase ″file:*user_client_installed_location*″ { permission java.io.FilePermission ″`C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar`″, ″read″; };.

To decide whether to add a permission, refer to Access control exception.

If you update the policy file, you must restart the browser and any client applications.

Developing with programmatic security APIs for Web applications

Use this information to programmatically secure APIs for Web applications.

Before you begin

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application. Programmatic security consists of the following methods of the HttpServletRequest interface:

getRemoteUser
: Returns the user name that the client used for authentication. Returns `null` if no user is authenticated.

isUserInRole
: (String role name): Returns `true` if the remote user is granted the specified security role. If the remote user is not granted the specified role, or if no user is authenticated, it returns `false`.

getUserPrincipal
: Returns the java.security.Principal object that contains the remote user name. If no user is authenticated, it returns `null`.

You can configure several options for Web authentication that determine how the Web client interacts with protected and unprotected Uniform Resource Identifiers (URI). Also, you can specify whether WebSphere Application Server challenges the Web client for basic authentication information if the certificate authentication for the HTTPS client fails. For more information, see Selecting an authentication mechanism.

You can enable a login module to indicate which principal class is returned by these calls. Refer to Map a registry principal to a System Authorization Facility user ID using a Java Authentication and Authorization Services login module for more information.

When the isUserInRole method is used, declare a security-role-ref element in the deployment descriptor with a role-name subelement containing the role name that is passed to this method, or with the @DeclareRoles annotation. Because actual roles are created during the assembly stage of the application, you can use a logical role as the role name and provide enough hints to the assembler in the description of the security-role-ref element to link that role to the actual role. During assembly, the assembler creates a role-link subelement to link the role name to the actual role. Creation of a security-role-ref element is possible if an assembly tool such as Rational Application Developer (RAD) is used. You also can create the security-role-ref element during assembly stage using an assembly tool.

1. Add the required security methods in the servlet code.

2. Create a security-role-ref element with the **role-name** field. If a security-role-ref element is not created during development, make sure it is created during the assembly stage.

Results

A programmatically secured servlet application.

Example

This step is required to secure an application programmatically. This action is particularly useful when a Web application needs to access external resources and wants to control the access to external resources using its own authorization table (external-resource to remote-user mapping). In this case, use the getUserPrincipal or the getRemoteUser methods to get the remote user and then it can consult its own authorization table to perform authorization. The remote user information also can help retrieve the corresponding user information from an external source such as a database or from an enterprise bean. You can use the isUserInRole method in a similar way.

After development, a security-role-ref element can be created:

```
<security-role-ref>
   <description>Provide hints to assembler for linking this role
               name to an actual role here<\description>
   <role-name>Mgr<\role-name>
</security-role-ref>
```

During assembly, the assembler creates a role-link element:

```
<security-role-ref>
   <description>Hints provided by developer to map the role
               name to the role-link</description>
   <role-name>Mgr</role-name>
   <role-link>Manager</role-link>
</security-role-ref>
```

You can add programmatic servlet security methods inside any servlet doGet, doPost, doPut, and doDelete service methods. The following example depicts using a programmatic security API:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response) {

   ....

   // to get remote user using getUserPrincipal()
   java.security.Principal principal = request.getUserPrincipal();
   String remoteUser = principal.getName();

   // to get remote user using getRemoteUser()
   remoteUser = request.getRemoteUser();

   // to check if remote user is granted Mgr role
   boolean isMgr = request.isUserInRole("Mgr");

   // use the above information in any way as needed by
   // the application
   ....

}
```

When developing Servlet 2.5 modules, the value of the rolename argument in isCallerInRole method can be defined using Java annotations instead of declaring a security-role-ref elements in the deployment descriptor.

```
@javax.annotation.security.DeclareRoles("Mgr")
public void doGet(HttpServletRequest request,
HttpServletResponse response) {

   ....

   // to get remote user using getUserPrincipal()
   java.security.Principal principal = request.getUserPrincipal();
   String remoteUser = principal.getName();

   // to get remote user using getRemoteUser()
   remoteUser = request.getRemoteUser();

   // to check if remote user is granted Mgr role
   boolean isMgr = request.isUserInRole("Mgr");

   // use the above information in any way as needed by
   // the application
   ....

}
```

What to do next

After developing an application, use an assembly tool to create roles and to link the actual roles to role names in the security-role-ref elements. For more information, see “Securing Web applications using an assembly tool” on page 63.

getRemoteUser and getAuthType methods:

The getRemoteUser and getAuthType methods are methods of the javax.servlet.http.HttpServletRequest interface. If the user has been authenticated, the getRemoteUser method returns the login of the user that makes the request. If the user is not authenticated, the getRemoteUser method returns null. The getAuthType method returns the name of the authentication scheme that is used to protect the servlet (for example, BASIC or SSL). If the servlet is not protected, the getAuthType method returns null.

For both methods, the data that is returned depends upon whether security is enabled in the application server where the servlet is deployed. The following possibilities exist:

- If security is not enabled, a servlet is requested and it is configured with Web server protection. The getRemoteUser method returns the login and getAuthType method returns the authentication scheme.
- If security is enabled and a servlet is requested, both methods return null when WebSphere Application Server protection is not configured for the servlet.
- If security is enabled, a servlet is requested, and the servlet is configured with WebSphere Application Server protection, then the getRemoteUser method returns the login and the getAuthType method returns the configured authentication scheme.

Note: You can disable security at the application server level by overriding the administrative security setting. For more information, see Securing specific application servers.

Example: Using a programmatic security model for a Web application:

The following example depicts a Web application or servlet using the programmatic security model.

This example illustrates one use and not necessarily the only use of the programmatic security model. The application can use the information that is returned by the getUserPrincipal, isUserInRole, and the getRemoteUser methods in any other way that is meaningful to that application. Use the declarative security model whenever possible.

File : HelloServlet.java

```
public class HelloServlet extends javax.servlet.http.HttpServlet {

 public void doPost(
  javax.servlet.http.HttpServletRequest request,
  javax.servlet.http.HttpServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException {
 }
public void doGet(
  javax.servlet.http.HttpServletRequest request,
  javax.servlet.http.HttpServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException {

        String s = "Hello";


        // get remote user using getUserPrincipal()
        java.security.Principal principal = request.getUserPrincipal();
        String remoteUserName = "";
        if( principal != null )
          remoteUserName = principal.getName();
// get remote user using getRemoteUser()
        String remoteUser = request.getRemoteUser();

        // check if remote user is granted Mgr role
        boolean isMgr = request.isUserInRole("Mgr");

        // display Hello username for managers and bob.
        if ( isMgr || remoteUserName.equals("bob") )
            s = "Hello " + remoteUserName;

   String message =  "<html> \n" +
              "<head><title>Hello Servlet</title></head>\n" +
         "<body> /n +"
    "<h1> "  +s+ </h1>/n " +
  byte[] bytes = message.getBytes();

  // displays "Hello" for ordinary users
        // and displays "Hello username" for managers and "bob".
        response.getOutputStream().write(bytes);
 }

}
```

After developing the servlet, you can create a security role reference for the HelloServlet servlet as shown in the following example:

```
<security-role-ref>
     <description> </description>
     <role-name>Mgr</role-name>
</security-role-ref>
```

Web authentication settings:

Use this page to specify the Web authentication settings that are associated with a Web client.

To view this administrative console page, complete the following steps:

1. Click **Security > Global security**.
2. Under Authentication, expand **Web and SIP security** and click **General settings**.

You can override the global Web authentication setting that you select on this panel by specifying a system property on the server level. To specify the system property, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** *server_name*.
2. Under Server infrastructure, click **Java and Process Management > Process definition**.

3. Under Additional properties, click **Java Virtual Machine > Custom properties > New**

You can specify the following system properties on the server level for Web authentication.

Table 32. Web authentication system property values

Property name	Value	Explanation
com.ibm.wsspi.security.web.webAuthReq	`lazy`	This value is equivalent to the **Authenticate only when the URI is protected** option.
com.ibm.wsspi.security.web.webAuthReq	`persisting`	This value is equivalent to the **Use available authentication data when an unprotected URI is accessed** option.
com.ibm.wsspi.security.web.webAuthReq	`always`	This value is equivalent to the **Authenticate when any URI is accessed** option.
com.ibm.wsspi.security.web.failOverToBasicAuth	`true`	This value is equivalent to the **Default to basic authentication when certificate authentication for the HTTPS client fails** option.

Authenticate only when the URI is protected:

The application server challenges the Web client to provide authentication data when the Web client accesses a Uniform Resource Identifier (URI) that is protected by a Java 2 Platform, Enterprise Edition (J2EE) role. The authenticated identity is available only when the Web client accesses a protected URI.

This option is the default J2EE Web authentication behavior that is also available in previous releases of WebSphere Application Server.

Note: When you select this option, the administrative console login page is missing images. You might encounter the following error in the administrative console: ″CWLAA6003: Could not display the portlet, the portlet may not be started. Check the error logs″.

The missing images and the error message are a side-effect of this option. The images do not display because the URIs for the images now need authentication, which requires you to log in. You can ignore this error message.

Default: Enabled

Use available authentication data when an unprotected URI is accessed:

The Web client can access validated authenticated data that it previously could not access. This option enables the Web client to call the getRemoteUser, isUserInRole, and getUserPrincipal methods to retrieve an authenticated identity from an unprotected URI.

When you select this option with the **Authenticate only when the URI is protected** option, the Web client can use authenticated data when the URI is protected or not protected.

Note: This option does not challenge the Web client to provide authenticated data if the Web client accesses an unprotected URI without authenticated data.

Default: Disabled

Authenticate when any URI is accessed:

The Web client must provide authentication data regardless of whether the URI is protected.

Default: Disabled

Default to basic authentication when certificate authentication for the HTTPS client fails:

When the required HTTPS client certificate authentication fails, the application server uses the basic authentication method to challenge the Web client to provide a user ID and password.

The HTTP client certification authentication that is performed by the application server security is different from the client authentication that is performed by the Web server plug-in. If you configure the Web server plug-in for mutual authentication and client authentication fails, the following situations will occur:

- The Web server produces a error and the Web request is not processed by application server security.
- The application server cannot fail over to basic authentication.

Default: Disabled

Developing with programmatic APIs for EJB applications

Use this topic to programmatically secure your Enterprise JavaBeans (EJB) applications.

About this task

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application. The javax.ejb.EJBContext application programming interface (API) provides two methods whereby the bean provider can access security information about the enterprise bean caller.

- **IsCallerInRole**(String rolename): Returns `true` if the bean caller is granted the security role that is specified by role name. If the caller is not granted the specified role, or if the caller is not authenticated, it returns `false`. If the specified role is granted Everyone access, it always returns `true`.
- **getCallerPrincipal**: Returns the java.security. Principal object that contains the bean caller name. If the caller is not authenticated, it returns a principal that contains an unauthorized name.

You can enable a login module to indicate which principal class is returned by these calls.

When the isCallerInRole method is used, declare a security-role-ref element in the deployment descriptor with a role-name that is subelement containing the role name that is passed to this method. Because actual roles are created during the assembly stage of the application, you can use a logical role as the role name and provide enough hints to the assembler in the description of the security-role-ref element to link that role to an actual role. During assembly, the assembler creates a role-link subelement to link the role-name to the actual role. Creation of a security-role-ref element is possible if an assembly tool such as Rational Application Developer (RAD) is used. You also can create the security-role-ref element during the assembly stage using an assembly tool.

1. Add the required security methods in the EJB module code.
2. Create a security-role-ref element with a role-name field for all the role names used in the isCallerInRole method. If a security-role-ref element is not created during development, make sure it is created during the assembly stage.

Results

Performing the previous steps result in a programmatically secured EJB application.

Example

Hard coding security policies in applications is strongly discouraged. The Java Platform, Enterprise Edition (Java EE) security model capabilities of declaratively specifying security policies is encouraged wherever possible. Use these APIs to develop security-aware EJB applications.

Using Java EE security model capabilities to specify security policies declaratively is useful when an EJB application wants to access external resources and wants to control the access to these external resources using its own authorization table (external-resource to user mapping). In this case, use the getCallerPrincipal method to get the caller identity and then the application can consult its own authorization table to perform authorization. The caller identification also can help retrieve the corresponding user information from an external source, such as database or from another enterprise bean. You can use the isCallerInRole method in a similar way.

After development, you can create a security-role-ref element:

```
<security-role-ref>
<description>Provide hints to assembler for linking this role-name to
actual role here<\description>
<role-name>Mgr<\role-name>
</security-role-ref>
```

During assembly, the assembler creates a role-link element:

```
<security-role-ref>
<description>Hints provided by developer to map role-name to role-link</description>
<role-name>Mgr</role-name>
<role-link>Manager</role-link>
</security-role-ref>
```

You can add programmatic EJB component security methods for example isCallerInRole and getCallerPrincipal, inside any business methods of an enterprise bean. The following example of programmatic security APIs includes a session bean:

```
public class aSessionBean implements SessionBean {

       .....

       // SessionContext extends EJBContext. If it is entity bean use EntityContext
       javax.ejb.SessionContext context;

       // The following method will be called by the EJB container
       // automatically
       public void setSessionContext(javax.ejb.SessionContext ctx) {
              context = ctx; // save the session bean's context
       }

       ....

       private  void aBusinessMethod()  {
       ....

       // to get  bean's caller using getCallerPrincipal()
       java.security.Principal principal = context.getCallerPrincipal();
       String  callerId= principal.getName();

       // to check if  bean's caller is granted Mgr role
       boolean isMgr = context.isCallerInRole("Mgr");

       // use the above information in any way as needed by the
       //application

       ....
```

```
        }

        ....
}
```

When developing EJB 3.0 modules, the value of the rolename argument in isCallerInRole method can be defined using Java annotations instead of declaring a security-role-ref elements in the deployment descriptor.

```
@javax.annotation.security.DeclareRoles("Mgr")
public class aSessionBean implements SessionBean {

        .....

        // SessionContext extends EJBContext. If it is entity bean use EntityContext
        javax.ejb.SessionContext context;

        // The following method will be called by the EJB container
        // automatically
        public void setSessionContext(javax.ejb.SessionContext ctx) {
               context = ctx; // save the session bean's context
        }

        ....

        private  void aBusinessMethod()  {
        ....

        // to get  bean's caller using getCallerPrincipal()
        java.security.Principal principal = context.getCallerPrincipal();
        String  callerId= principal.getName();

        // to check if  bean's caller is granted Mgr role
        boolean isMgr = context.isCallerInRole("Mgr");

        // use the above information in any way as needed by the
        //application

        ....
        }

        ....
}
```

What to do next

After developing an application, use an assembly tool to create roles and to link the actual roles to role names in the security-role-ref elements. For more information, see Securing enterprise bean applications.

Example: Enterprise bean application code:

The following Enterprise JavaBeans (EJB) component example illustrates the use of the isCallerInRole and the getCallerPrincipal methods in an EJB module.

Using declarative security is recommended. The following example is one way of using the isCallerInRole and the getCallerPrincipal methods. The application can use this result in any way that is suitable.

A remote interface

```
File : Hello.java

package tests;
```

```
import java.rmi.RemoteException;
/**
 * Remote interface for Enterprise Bean: Hello
 */
public interface Hello extends javax.ejb.EJBObject {
      public abstract String getMessage()throws RemoteException;
     public abstract void setMessage(String s)throws RemoteException;
}
```

A home interface

```
File : HelloHome.java
package tests;
/**
 * Home interface for Enterprise Bean: Hello
 */
public interface HelloHome extends javax.ejb.EJBHome {
 /**
  * Creates a default instance of Session Bean: Hello
  */
 public tests.Hello create() throws javax.ejb.CreateException,
  java.rmi.RemoteException;
}
```

A bean implementation

```
File : HelloBean.java

package tests;
/**
 * Bean implementation class for Enterprise Bean: Hello
 */
public class HelloBean implements javax.ejb.SessionBean {
 private javax.ejb.SessionContext mySessionCtx;
 /**
  * getSessionContext
  */
 public javax.ejb.SessionContext getSessionContext() {
  return mySessionCtx;
 }
 /**
  * setSessionContext
  */
 public void setSessionContext(javax.ejb.SessionContext ctx) {
  mySessionCtx = ctx;
 }
 /**
  * ejbActivate
  */
 public void ejbActivate() {
 }
 /**
  * ejbCreate
  */
 public void ejbCreate() throws javax.ejb.CreateException {
 }
 /**
  * ejbPassivate
  */
 public void ejbPassivate() {
 }
```

```
/**
 * ejbRemove
 */
public void ejbRemove() {
}

public java.lang.String message;


     //business methods

     // all users can call getMessage()
     public String getMessage() {
         return message;
     }

     // all users can call setMessage() but only few users can set new message.
     public void setMessage(String s) {

       // get  bean's caller using getCallerPrincipal()
       java.security.Principal principal = mySessionCtx.getCallerPrincipal();
       java.lang.String  callerId= principal.getName();

       // check if  bean's caller is granted Mgr role
       boolean isMgr = mySessionCtx.isCallerInRole("Mgr");

       // only set supplied message if caller is "bob" or caller is granted Mgr role
       if ( isMgr || callerId.equals("bob") )
           message = s;
       else
           message = "Hello";
     }

}
```

After the development of the entity bean, create a security role reference in the deployment descriptor under the session bean, Hello:

```
<security-role-ref>
     <description>Only Managers can call setMessage() on this bean (Hello)</description>
     <role-name>Mgr</role-name>
</security-role-ref>
```

For an explanation of how to create a <security-role-ref> element, see Securing enterprise bean applications. Use the information under Map security-role-ref and role-name to role-link to create the element.

Customizing Web application login

You can create a form login page and an error page to authenticate a user.

Before you begin

A Web client or a browser can authenticate a user to a Web server using one of the following mechanisms:

- **HTTP basic authentication**: A Web server requests the Web client to authenticate and the Web client passes a user ID and a password in the HTTP header.

- **HTTPS client authentication**: This mechanism requires a user (Web client) to possess a public key certificate. The Web client sends the certificate to a Web server that requests the client certificates. This authentication mechanism is strong and uses the Hypertext Transfer Protocol with Secure Sockets Layer (HTTPS) protocol.
- **Form-based Authentication**: A developer controls the look and feel of the login screens using this authentication mechanism.

The Hypertext Transfer Protocol (HTTP) basic authentication transmits a user password from the Web client to the Web server in simple base64 encoding. Form-based authentication transmits a user password from the browser to the Web server in plain text. Therefore, both HTTP basic authentication and form-based authentication are not very secure unless the HTTPS protocol is used.

The Web application deployment descriptor contains information about which authentication mechanism to use. When form-based authentication is used, the deployment descriptor also contains entries for login and error pages. A login page can be either an HTML page or a JavaServer Pages (JSP) file. This login page displays on the Web client side when a secured resource (servlet, JSP file, HTML page) is accessed from the application. On authentication failure, an error page displays. You can write login and error pages to suit the application needs and control the look and feel of these pages. During assembly of the application, an assembler can set the authentication mechanism for the application and set the login and error pages in the deployment descriptor.

Form login uses the servlet sendRedirect method, which has several implications for the user. The sendRedirect method is used twice during form login:
- The sendRedirect method initially displays the form login page in the Web browser. It later redirects the Web browser back to the originally requested protected page. The sendRedirect(String URL) method tells the Web browser to use the HTTP GET request to get the page that is specified in the Web address. If HTTP POST is the first request to a protected servlet or JavaServer Pages (JSP) file, and no previous authentication or login occurred, then HTTP POST is not delivered to the requested page. However, HTTP GET is delivered because form login uses the sendRedirect method, which behaves as an HTTP GET request that tries to display a requested page after a login occurs.
- Using HTTP POST, you might experience a scenario where an unprotected HTML form collects data from users and then posts this data to protected servlets or JSP files for processing, but the users are not logged in for the resource. To avoid this scenario, structure your Web application or permissions so that users are forced to use a form login page before the application performs any HTTP POST actions to protected servlets or JSP files.

1. Create a form login page with the required look and feel, including the required elements to perform form-based authentication. For an example, see “Example: Form login” on page 1336.
2. Create an error page. You can program error pages to retry authentication or to display an appropriate error message.
3. Place the login page and error page in the Web archive (`.war`) file relative to the top directory. For example, if the login page is configured as `/login.html` in the deployment descriptor, place it in the top directory of the WAR file. An assembler can also perform this step using the assembly tool.
4. Create a form logout page and insert it to the application only when the Web application requires a form-based authentication mechanism.

Example: Form login

See the “Example: Form login” on page 1336 article for sample form login pages.

The WebSphere Application Server Samples Gallery provides a form login Sample that demonstrates how to use the WebSphere Application Server login facilities to implement and configure form login procedures. The Sample integrates the following technologies to demonstrate the WebSphere Application Server and Java Platform, Enterprise Edition (Java EE) login functionality:

- Java EE form-based login

- Java EE servlet filter with login
- IBM extension: form-based login

The form login sample is part of the Technology Samples package. For more information on how to access the form login sample, see Accessing the Samples (Samples Gallery).

For the authentication to proceed appropriately, the action of the login form must always have the j_security_check action. The following example shows how to code the form into the HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="text" name="j_password">
<\form>
```

Use the **j_username** input field to get the user name, and use the **j_password** input field to get the user password.

On receiving a request from a Web client, the Web server sends the configured form page to the client and preserves the original request. When the Web server receives the completed form page from the Web client, the server extracts the user name and password from the form and authenticates the user. On successful authentication, the Web server redirects the call to the original request. If authentication fails, the Web server redirects the call to the configured error page.

The following example depicts a login page in HTML (`login.html`):

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<META HTTP-EQUIV = "Pragma" CONTENT="no-cache">
<title> Security FVT Login Page </title>
<body>
<h2>Form Login</h2>
<FORM METHOD=POST ACTION="j_security_check">
<p>
<font size="2"> <strong> Enter user ID and password: </strong></font>
<BR>
<strong> User ID</strong> <input type="text" size="20" name="j_username">
<strong> Password </strong>  <input type="password" size="20" name="j_password">
<BR>
<BR>
<font size="2">  <strong> And then click this button: </strong></font>
<input type="submit" name="login" value="Login">
</p>

</form>
</body>
</html>
```

The following example depicts an error page in a JSP file:

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<head><title>A Form login authentication failure occurred</head></title>
<body>
<H1><B>A Form login authentication failure occurred</H1></B>
<P>Authentication may fail for one of many reasons. Some possibilities include:
<OL>
<LI>The user-id or password may be entered incorrectly; either misspelled or the
wrong case was used.
<LI>The user-id or password does not exist, has expired, or has been disabled.
</OL>
</P>
</body>
</html>
```

After an assembler configures the Web application to use form-based authentication, the deployment descriptor contains the login configuration as shown:

```
<login-config id="LoginConfig_1">
<auth-method>FORM<auth-method>
<realm-name>Example Form-Based Authentication Area</realm-name>
<form-login-config id="FormLoginConfig_1">
<form-login-page>/login.html</form-login-page>
<form-error-page>/error.jsp</form-error-page>
</form-login-config>
</login-config>
```

A sample Web application archive (WAR) file directory structure that shows login and error pages for the previous login configuration follows:

```
META-INF
     META-INF/MANIFEST.MF
     login.html
     error.jsp
     WEB-INF/
     WEB-INF/classes/
     WEB-INF/classes/aServlet.class
```

What to do next

After developing login and error pages, add them to the Web application. Use the assembly tool to configure an authentication mechanism and insert the developed login page and error page in the deployment descriptor of the application.

Example: Form login

This article provides several examples pertaining to form login.

For the authentication to proceed appropriately, the action of the login form must always have the j_security_check action. The following example shows how to code the form into the HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="text" name="j_password">
<\form>
```

Use the **j_username** input field to get the user name, and use the **j_password** input field to get the user password.

On receiving a request from a Web client, the Web server sends the configured form page to the client and preserves the original request. When the Web server receives the completed form page from the Web client, the server extracts the user name and password from the form and authenticates the user. On successful authentication, the Web server redirects the call to the original request. If authentication fails, the Web server redirects the call to the configured error page.

The following example depicts a login page in HTML (`login.html`):

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<META HTTP-EQUIV = "Pragma" CONTENT="no-cache">
<title> Security FVT Login Page </title>
<body>
<h2>Form Login</h2>
<FORM METHOD=POST ACTION="j_security_check">
<p>
<font size="2"> <strong> Enter user ID and password: </strong></font>
<BR>
<strong> User ID</strong> <input type="text" size="20" name="j_username">
<strong> Password </strong>  <input type="password" size="20" name="j_password">
<BR>
```

```
<BR>
<font size="2">  <strong> And then click this button: </strong></font>
<input type="submit" name="login" value="Login">
</p>

</form>
</body>
</html>
```

The following example depicts an error page in a JSP file:

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<head><title>A Form login authentication failure occurred</head></title>
<body>
<H1><B>A Form login authentication failure occurred</H1></B>
<P>Authentication may fail for one of many reasons. Some possibilities include:
<OL>
<LI>The user-id or password may be entered incorrectly; either misspelled or the
wrong case was used.
<LI>The user-id or password does not exist, has expired, or has been disabled.
</OL>
</P>
</body>
</html>
```

After an assembler configures the Web application to use form-based authentication, the deployment descriptor contains the login configuration as shown:

```
<login-config id="LoginConfig_1">
<auth-method>FORM<auth-method>
<realm-name>Example Form-Based Authentication Area</realm-name>
<form-login-config id="FormLoginConfig_1">
<form-login-page>/login.html</form-login-page>
<form-error-page>/error.jsp</form-error-page>
</form-login-config>
</login-config>
```

A sample Web application archive (WAR) file directory structure that shows login and error pages for the previous login configuration follows:

```
META-INF
     META-INF/MANIFEST.MF
     login.html
     error.jsp
     WEB-INF/
     WEB-INF/classes/
     WEB-INF/classes/aServlet.class
```

Form logout

Form logout is a mechanism to log out without having to close all Web-browser sessions. After logging out of the form logout mechanism, access to a protected Web resource requires re-authentication. This feature is not required by J2EE specifications, but it is provided as an additional feature in WebSphere Application Server security.

Suppose that you want to log out after logging into a Web application and perform some actions. A form logout works in the following manner:

1. The logout-form URI is specified in the Web browser and loads the form.
2. The user clicks **Submit** on the form to log out.
3. The WebSphere Application Server security code logs the user out. During this process, the Application Server completes the following processes:
   a. Clears the Lightweight Third Party Authentication (LTPA) / single sign-on (SSO) cookies
   b. Invalidates the HTTP session

c. Removes the user from the authentication cache
4. Upon logout, the user is redirected to a logout exit page.

Form logout does not require any attributes in a deployment descriptor. The form-logout page is an HTML or a JavaServer Pages (JSP) file that is included with the Web application. The form-logout page is like most HTML forms except that like the form-login page, the form-logout page has a special post action. This post action is recognized by the Web container, which dispatches the post action to a special internal form-logout servlet. The post action in the form-logout page must be `ibm_security_logout`.

You can specify a logout-exit page in the logout form and the exit page can represent an HTML or a JSP file within the same Web application to which the user is redirected after logging out. Additionally, the logout-exit page permits a fully qualified URL in the form of `http://`*`hostname`*`:`*`port`*`/`*`URL`*. The logout-exit page is specified as a parameter in the form-logout page. If no logout-exit page is specified, a default logout HTML message is returned to the user.

Here is a sample form logout HTML form. This form configures the logout-exit page to redirect the user back to the login page after logout.

```
<!DOCTYPE HTML PUBliC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
  <META HTTP-EQUIV = "Pragma" CONTENT="no-cache">
  <title>Logout Page </title>
  <body>
  <h2>Sample Form Logout</h2>
      <FORM METHOD=POST ACTION="ibm_security_logout" NAME="logout">
      <p>
      <BR>
      <BR>
      <font size="2"><strong> Click this button to log out: </strong></font>
      <input type="submit" name="logout" value="Logout">
      <INPUT TYPE="HIDDEN" name="logoutExitPage" VALUE="/login.html">
      </p>
      </form>
  </body>
</html>
```

The WebSphere Application Server Samples Gallery provides a form login Sample that demonstrates how to use the WebSphere Application Server login facilities to implement and configure form login procedures. The Sample integrates the following technologies to demonstrate the WebSphere Application Server and Java 2 Platform, Enterprise Edition (J2EE) login functionality:

- J2EE form-based login
- J2EE servlet filter with login
- IBM extension: form-based login

The form login Sample is part of the Technology Samples package.

Developing servlet filters for form login processing

You can control the look and feel of the login screen using the form-based login mechanism. In form-based login, you specify a login page that is used to retrieve the user ID and password information. You also can specify an error page that displays when authentication fails.

About this task

If additional authentication or additional processing is required before and after authentication, servlet filters are an option. Servlet filters can dynamically intercept requests and responses to transform or to use the information that is contained in the requests or responses. One or more servlet filters can be attached to a servlet or to a group of servlets. Servlet filters also can attach to JavaServer Pages (JSP) files and HTML pages. All of the attached servlet filters are called before the servlet is invoked.

Both form-based login and servlet filters are supported by any servlet Version 2.3 specification-complaint Web container. The form login servlet performs the authentication and servlet filters perform additional authentication, auditing, or logging information.

To perform pre-login and post-login actions using servlet filters, configure these filters for either form login page support or for the /j_security_check URL. The j_security_check is posted by a form login page with the j_username parameter that contains the user name and the j_password parameter that contains the password. A servlet filter can use the user name parameter and password information to perform more authentication or other special needs.

1. A servlet filter implements the javax.servlet.Filter class. Implement three methods in the filter class:
   - **init(javax.servlet.FilterConfig cfg)**. This method is called by the container once, when the servlet filter is placed into service. The FilterConfig passed to this method contains the init-parameters of the servlet filter. Specify the init-parameters for a servlet filter during configuration using the assembly tool.
   - **destroy**. This method is called by the container when the servlet filter is taken out of a service.
   - **doFilter(ServletRequest req, ServletResponse res, FilterChain chain)**. This method is called by the container for every servlet request that maps to this filter before invoking the servlet. The FilterChain chain that is passed to this method can be used to invoke the next filter in the chain of filters. The original requested servlet runs when the last filter in the chain calls the chain.doFilter method. Therefore, all filters call the chain.doFilter method for the original servlet to run after filtering. If an additional authentication check is implemented in the filter code and results in failure, the original servlet does not run. The chain.doFilter method is not called and can be redirected to some other error page.
2. If a servlet maps to many servlet filters, servlet filters are called in the order that is listed in the `web.xml` deployment descriptor of the application. Place the servlet filter class file in the `WEB-INF/classes` directory of the application.

Example

An example of a servlet filter follows: This login filter can map to the /j_security_check URL to perform pre-login and post-login actions.

```
import javax.servlet.*;
     public class LoginFilter implements Filter {
     protected FilterConfig filterConfig;
     // Called once when this filter is instantiated.
     // If mapped to j_security_check, called
     // very first time j_security_check is invoked.
     public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
        }
     public void destroy() {
        this.filterConfig = null;
        }
      // Called for every request that is mapped to this filter.
     // If mapped to j_security_check,
     // called for every  j_security_check action
     public void doFilter(ServletRequest request,
     ServletResponse response, FilterChain chain)
         throws java.io.IOException, ServletException   {
         // perform pre-login action here
         chain.doFilter(request, response);
         // calls the next filter in chain.
         // j_security_check if this filter is
         // mapped to j_security_check.
        // perform post-login action here.
                 }
       }
```

Example: Using servlet filters to perform pre-login and post-login processing during form login:

This example illustrates one way that the servlet filters can perform pre-login and post-login processing during form login.

Servlet filter source code: LoginFilter.java

```
/**
 * A servlet filter example: This example filters j_security_check and
 * performs pre-login action to determine if the user trying to log in
 * is in the revoked list. If the user is on the revoked list, an error is
 * sent back to the browser.
 *
 * This filter reads the revoked list file name from the FilterConfig
 * passed in the init() method. It reads the revoked user list file and
 * creates a revokedUsers list.
 *
 * When the doFilter method is called, the user logging in is checked
 * to make sure that the user is not on the revoked Users list.
 *
 */

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class LoginFilter implements Filter {

   protected FilterConfig filterConfig;

   java.util.List revokeList;


   /**
    * init() : init() method called when the filter is instantiated.
    * This filter is instantiated the first time j_security_check is
    * invoked for the application (When a protected servlet in the
    * application is accessed).
    */
   public void init(FilterConfig filterConfig) throws ServletException {
      this.filterConfig = filterConfig;

      // read revoked user list
      revokeList = new java.util.ArrayList();
      readConfig();
   }


   /**
    * destroy() : destroy() method called when the filter is taken
    * out of service.
    */
   public void destroy() {
      this.filterConfig = null;
      revokeList = null;
   }

   /**
    * doFilter() : doFilter() method called before the servlet to
    * which this filter is mapped is invoked. Since this filter is
    * mapped to j_security_check,this method is called before
    * j_security_check action is posted.
    */
   public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws java.io.IOException, ServletException {


      HttpServletRequest req = (HttpServletRequest)request;
      HttpServletResponse res = (HttpServletResponse)response;

      // pre login action

      // get username
      String username = req.getParameter("j_username");
```

```
      // if user is in revoked list send error
      if ( revokeList.contains(username) ) {
      res.sendError(javax.servlet.http.HttpServletResponse.SC_UNAUTHORIZED);
      return;
      }

      // call next filter in the chain : let j_security_check authenticate
      // user
      chain.doFilter(request, response);

      // post login action

   }

   /**
    * readConfig() : Reads revoked user list file and creates a revoked
    * user list.
    */
   private void readConfig() {
      if ( filterConfig != null ) {

         // get the revoked user list file and open it.
         BufferedReader in;
         try {
               String filename = filterConfig.getInitParameter("RevokedUsers");
               in = new BufferedReader( new FileReader(filename));
         } catch ( FileNotFoundException fnfe) {
               return;
         }

         // read all the revoked users and add to revokeList.
         String userName;
         try {
               while ( (userName = in.readLine()) != null )
                   revokeList.add(userName);
         } catch (IOException ioe) {
         }

      }

   }


}
```

Note: In the previous code sample, the line that begins `public void doFilter(ServletRequest request` is broken into two lines for illustrative purposes only. The `public void doFilter(ServletRequest request` line and the line after it are one continuous line.

An example of the `web.xml` file that shows the LoginFilter filter configured and mapped to the j_security_check URL:

```
<filter id="Filter_1">
    <filter-name>LoginFilter</filter-name>
    <filter-class>LoginFilter</filter-class>
         <description>Performs pre-login and post-login operation</description>
    <init-param>
    <param-name>RevokedUsers</param-name>
    <param-value>c:\WebSphere\AppServer\installedApps\
                            <app-name>\revokedUsers.lst</param-value>
    </init-param>
</filter-id>

<filter-mapping>
 <filter-name>LoginFilter</filter-name>
     <url-pattern>/j_security_check</url-pattern>
</filter-mapping>
```

An example of a revoked user list file:

```
user1
cn=user1,o=ibm,c=us
user99
cn=user99,o=ibm,c=us
```

Configuring servlet filters:

IBM Rational Application Developer or an assembly tool can configure the servlet filters. Two steps are involved in configuring a servlet filter.

1. Name the servlet filter and assign the corresponding implementation class to the servlet filter.

   Optionally, assign initialization parameters that get passed to the init method of the servlet filter.After configuring the servlet filter, the `web.xml` application deployment descriptor contains a servlet filter configuration similar to the following example:

   ```
   <filter id="Filter_1">
      <filter-name>LoginFilter</filter-name>
      <filter-class>LoginFilter</filter-class>
      <description>Performs pre-login and post-login
           operation</description>
      <init-param>// optional
        <param-name>ParameterName</param-name>
        <param-value>ParameterValue</param-value>
      </init-param>
   </filter>
   ```

2. Map the servlet filter to a URL or a servlet.

   After mapping the servlet filter to a URL or a servlet, the`web.xml` application deployment descriptor contains servlet mapping similar to the following example:

   ```
   <filter-mapping>
      <filter-name>LoginFilter</filter-name>
      <url-pattern>/j_security_check</url-pattern>
                  // can be servlet <servlet>servletName</servlet>
   </filter-mapping>
   ```

Example

You can use servlet filters to replace the CustomLoginServlet servlet, and to perform additional authentication, auditing, and logging.

The WebSphere Application Server Samples Gallery provides a form login sample that demonstrates how to use the WebSphere Application Server login facilities to implement and configure form login procedures. The sample integrates the following technologies to demonstrate the WebSphere Application Server and Java Platform, Enterprise Edition (Java EE) login functionality:

- Java EE form-based login
- Java EE servlet filter with login
- IBM extension: form-based login

The form login sample is part of the Technology Samples package. For more information on how to access the form login sample, see Accessing the Samples (Samples Gallery).

Secure transports with JSSE and JCE programming interfaces

This topic provides detailed information about transport security using Java Secure Socket Extension (JSSE) and Java Cryptography Extension (JCE) programming interfaces. Within this topic, there is a description of the IBM version of the Java Cryptography Extension Federal Information Processing Standard (IBMJCEFIPS).

Java Secure Socket Extension

Java Secure Socket Extension (JSSE) provides the transport security for WebSphere Application Server. JSSE provides the application programming interface (API) framework and the implementation of the APIs for Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, including functionality for data encryption, message integrity, and authentication.

JSSE APIs are integrated into the Java 2 SDK, Standard Edition (J2SDK), Version 5. The API package for JSSE APIs is `javax.net.ssl.*`. Documentation for using JSSE APIs can be found in the J2SE 6 API documentation that is located at `http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`.

Several JSSE providers ship with the Java 2 SDK Version 5 that comes with WebSphere Application Server. The IBMJSSE provider is used in previous WebSphere Application Server releases. Associated with the IBMJSSE provider is the IBMJSSEFIPS provider, which is used when FIPS is enabled on the server. Both of these providers do not work with the Java Message Service (JMS) and HTTP transports in WebSphere Application Server Version 7.0. These transports take advantage of the J2SDK Verison 5 network input/output (NIO) asynchronous channels.

For more information on the new IBMJSSE2 provider, please review the documentation located at `http://www.ibm.com/developerworks/java/jdk/security/60/`.

Customizing Java Secure Socket Extension

You can customize a number of aspects of JSSE by plugging in different implementations of Cryptography Package Provider, X509Certificate and HTTPS protocols, or specifying different default keystore files, key manager factories, and trust manager factories. The following table summarizes which aspects can be customized, what the defaults are, and which mechanisms are used to provide customization. You can customize the following key aspects:

Customizable item	Default	How to customize
X509Certificate	X509Certificate implementation from IBM	The cert.provider.x509v1 security property
HTTPS protocol	Implementation from IBM	The java.protocol.handler.pkgs system property
Cryptography Package Provider	IBMJSSE2	A security.provider.n= line in security properties file. See description.
Default keystore	None	The * javax.net.ssl.keyStore system property
Default truststore	jssecacerts, if it exists. Otherwise, cacerts	The * javax.net.ssl.trustStore system property
Default key manager factory	IbmX509	The ssl.KeyManagerFactory.algorithm security property
Default trust manager factory	IbmX509	The ssl.TrustManagerFactory.algorithm security property

For aspects that you can customize by setting a system property, statically set the system property by using the **-D** option of the **Java** command. You can set the system property using the administrative console, or set the system property dynamically by calling the java.lang.System.setProperty method in your code: System.setProperty(propertyName,″propertyValue″).

For aspects that you can customize by setting a Java security property, statically specify a security property value in the `java.security` properties file. The security property is propertyName=propertyValue. Dynamically set the Java security property by calling the java.security.Security.setProperty method in your code.

The `java.security` properties file is located in the following directory:

app_server_root`/java/jre/lib/security` directory.

Application Programming Interface

The JSSE provides a standard application programming interface (API) that is available in packages of the `javax.net` file, `javax.net.ssl` file, and the `javax.security.cert` file. The APIs cover:
- Sockets and SSL sockets
- Factories to create the sockets and SSL sockets
- Secure socket context that acts as a factory for secure socket factories
- Key and trust manager interfaces
- Secure HTTP URL connection classes
- Public key certificate API

Samples using Java Secure Socket Extension

The Java Secure Socket Extension (JSSE) also provides samples to demonstrate its functionality. The Java Secure Socket Extension (JSSE) also provides samples to demonstrate its functionality. You can access the samples in the following location:

Version 1.6

1. Access the http://www.ibm.com/developerworks/java/jdk/security/ Web site.
2. Click **Java 1.6**.
3. Click **jssedocs_samples.zip** in the Java Secure Socket Extension (JSSE) Guide section.

Look for the following files:

Files	Description
ClientJsse.java	Demonstrates a simple client and server interaction using JSSE. All enabled cipher suites are used.
OldServerJsse.java	Back-level samples
ServerPKCS12Jsse.java	Demonstrates a simple client and server interaction using JSSE with the PKCS12 keystore file. All enabled cipher suites are used.
ClientPKCS12Jsse.java	Demonstrates a simple client and server interaction using JSSE with the PKCS12 keystore file. All enabled cipher suites are used.
UseHttps.java	Demonstrates accessing an SSL or non-SSL Web server using the Java protocol handler of the com.ibm.net.ssl.www.protocol class. The URL is specified with the `http` or `https` prefix. The HTML that is returned from this site is displayed.

See more instructions in the source code. Follow these instructions before you run the samples.

Permissions for Java 2 security

You might need the following permissions to run an application with JSSE: This list is for reference only.
- java.util.PropertyPermission ″java.protocol.handler.pkgs″, ″write″
- java.lang.RuntimePermission ″writeFileDescriptor″
- java.lang.RuntimePermission ″readFileDescriptor″
- java.lang.RuntimePermission ″accessClassInPackage.sun.security.x509″
- java.io.FilePermission ″${user.install.root}${/}etc${/}.keystore″, ″read″
- java.io.FilePermission ″${user.install.root}${/}etc${/}.truststore″, ″read″

For the IBMJSSE provider:
- java.security.SecurityPermission ″putProviderProperty.IBMJSSE″

- java.security.SecurityPermission ″insertProvider.IBMJSSE″

For the SUNJSSE provider:
- java.security.SecurityPermission ″putProviderProperty.SunJSSE″
- java.security.SecurityPermission ″insertProvider.SunJSSE″

Debugging

By configuring through the javax.net.debug system property, JSSE provides the following dynamic debug tracing: `-Djavax.net.debug=true`.

A value of `true` turns on the trace facility. Use the administrative console to set the system property for debugging the application server.

Documentation

See the Security: Resources for learning topic for documentation references to JSSE.

JCE

Java Cryptography Extension (JCE) provides cryptographic, key and hash algorithms for WebSphere Application Server. JCE provides a framework and implementations for encryption, key generation, key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block and stream ciphers.

IBMJCE

The IBM version of the Java Cryptography Extension (IBMJCE) is an implementation of the JCE cryptographic service provider that is used in WebSphere Application Server. The IBMJCE is similar to SunJCE, except that the IBMJCE offers more algorithms:
- Cipher algorithm (AES, DES, TripleDES, PBEs, Blowfish, and so on)
- Signature algorithm (SHA1withRSA, MD5withRSA, SHA1withDSA)
- Message digest algorithm (MD5, MD2, SHA1, SHA-256, SHA-384, SHA-512)
- Message authentication code (HmacSHA1, HmacMD5)
- Key agreement algorithm (DiffieHellman)
- Random number generation algorithm (IBMSecureRandom, SHA1PRNG)
- Key store (JKS, JCEKS, PKCS12, JCERACFKS [z/OS only])

The IBMJCE belongs to the com.ibm.crypto.provider.* packages.

For further information, see the information on JCE on the following web site: http://www.ibm.com/developerworks/java/jdk/security/60/.

IBMJCEFIPS

The IBM version of the Java Cryptography Extension Federal Information Processing Standard (IBMJCEFIPS) is an implementation of the JCE cryptographic service provider that is used in WebSphere Application Server. The IBMJCEFIPS service provider implements the following:
- Signature algorithms (SHA1withDSA, SHA1withRSA)
- Cipher algorithms (AES, TripleDES, RSA)
- Key agreement algorithm (DiffieHellman)
- Key (pair) generator (DSA, AES, TripleDES, HmacSHA1, RSA, DiffieHellman)
- Message authentication code (MAC) (HmacSHA1)
- Message digest (MD5, SHA-1, SHA-256, SHA-384, SHA-512)
- Algorithm parameter generator (DiffieHellman, DSA)

- Algorithm parameter (AES, DiffieHellman, DES, TripleDES, DSA)
- Key factory (DiffieHellman, DSA, RSA)
- Secret key factory (AES, TripleDES)
- Certificate (X.509)
- Secure random (IBMSecureRandom)

Application Programming Interface

Java Cryptography Extension (JCE) has a provider-based architecture. Providers can be plugged into the JCE framework by implementing the APIs that are defined by the JCE. The JCE APIs cover:
- Symmetric bulk encryption, such as DES, RC2, and IDEA
- Symmetric stream encryption, such as RC4
- Asymmetric encryption, such as RSA
- Password-based encryption (PBE)
- Key agreement
- Message authentication codes

Samples using Java Cryptography Extension

There are samples located on the `http://www.ibm.com/developerworks/java/jdk/security/` Web site in the `jceDocs_samples.zip` file. Unzip the file and locate the following samples in the `jceDocs/samples` directory:

File	Description
SampleDSASignature.java	Demonstrates how to generate a pair of DSA keys (a public key and a private key) and use the key to digitally sign a message using the SHA1withDSA algorithm
SampleMarsCrypto.java	Demonstrates how to generate a Mars secret key, and how to do Mars encryption and decryption
SampleMessageDigests.java	Demonstrates how to use the message digest for MD2 and MD5 algorithms
SampleRSACrypto.java	Demonstrates how to generate an RSA key pair, and how to do RSA encryption and decryption
SampleRSASignatures.java	Demonstrates how to generate a pair of RSA keys (a public key and a private key) and use the key to digitally sign a message using the SHA1withRSA algorithm
SampleX509Verification.java	Demonstrates how to verify X509 certificates

Documentation

Refer to the Security: Resources for learning for documentation on JCE.

Using System Authorization Facility keyrings with Java Secure Sockets Extension

WebSphere Application Server for z/OS customers running server W50100*x* or later, with Java Development Kit 1.3 level SR20 or later, can modify their WebSphere Application Server systems to use System Authorization Facility (SAF) for Java Secure Sockets Extension (JSSE) as well as Secure Sockets Layer (SSL), which eliminates the need to maintain duplicate certificates in the hierarchical file system (HFS).

Before you begin

WebSphere Application Server for z/OS running at maintenance levels before W502000 stored digital certificate information in two different places because of the following Software Development Kit (SDK) restrictions:

- JSSE used digital certificates stored in hierarchical file system files
- SSL used digital certificate information stored in the SAF database

Systems customized at W502000 or above use the single SAF digital certificate repository by default, and do not need the modifications described below.

About this task

WebSphere Application Server for z/OS customers running server W50100x or later, with Java Development Kit 1.3 level SR20 or later, can modify their WebSphere Application Server systems to use SAF for JSSE as well as SSL (eliminating the need to maintain duplicate certificates in the HFS). The instructions below describe how to enable this support.

Note: Systems that are customized at maintenance levels at or after W502000 use the single (SAF digital certificate repository by default, and these systems do not need the modifications described below.

To use SAF certificates with JSSE:

1. Update the Java Management Extensions (JMX) connector settings to indicate the SAF keyring names for the node.
   a. Log in to the administrative console using an identity with administrator authority.
   b. Click **Servers > Application servers >** *server_name*.
   c. Under Server infrastructure, click **Administration > Administration services**.
   d. Under Additional properties, click **JMX connectors**.
   e. On the JMX Connectors panel, click **SOAPConnector**.
   f. Under Additional Properties, click **Custom Properties**.
   g. On the Custom properties page, click **sslConfig**.
   h. On the sslConfig page, look at the Value field. Verify that this field says *node_name/DefaultSSLSettings*, where *nodename* represents the node name where the application server resides. Record the node name for a subsequent step.
   i. Select ***node_name*/RACFJSSESettings** from the list next to the Value field, where *node_name* is the same as the node name that you previously recorded.
   j. Click **OK**. The Custom Properties page appears with a message indicating that changes are made to your local configuration. Do not click **Save** because additional changes that are required.
2. Click **Servers > Application servers** and repeat the previous substeps for each of the other application servers in the cell.
3. Update the Java Management Extensions (JMX) connector settings to indicate the SAF keyring names for the deployment manager node.
   a. Click **System administration > Deployment manager**.
   b. Under Additional properties, click **Administration services > JMX Connectors**.
   c. On the JMX Connectors panel, click **SOAPConnector**.
   d. Under Additional properties, click **Custom properties**.
   e. On the Custom properties page, click **sslConfig**.
   f. On the sslConfig page, look at the Value field. This field displays *dmnode/DefaultSSLSettings*, where *dmnode* represents the deployment manager node name. Record the node name for a subsequent step.

g. Select **dmnode/RACFJSSESettings** from the list next to the Value field, where **dmnode** represents the Deployment Manager node name.

h. Click **OK**. After a short time the Custom Properties page appears with a message at the top indicating that changes have been made to your local configuration. Do not click **Save** at this point because there are additional changes that are required.

4. Update the Java Management Extensions (JMX) connector settings to indicate the SAF keyring names for the node agent.

   a. Click **System administration > Node agents >** *Node_name*. Record the node agent name for the next step.

   b. Under Additional properties, click **Administration services > JMX Connectors**.

   c. On the JMX Connectors panel, click **SOAPConnector**.

   d. Under Additional properties, click **Custom properties**.

   e. On the Custom properties page, click **sslConfig**.

   f. On the sslConfig page, look at the Value field. This field displays *nodename/DefaultSSLSettings*, where *nodename* is the node name where the node agent resides. Record the node name for a subsequent step.

   g. Select **nodename/RACFJSSESettings** from the list next to the Value field, where **nodename** is the node name that you previously recorded.

   h. Click **OK**. The Custom Properties page is displayed with a message indicating that changes have been made to the local configuration. Do not click **Save** at this point because additional changes are required.

5. Click **System administration > Node agents** and repeat the previous substeps for each of the other node agents servers in the cell.

6. Click **Save** when the `Changes have been made to your local configuration. Click Save to apply changes to the master configuration` message is displayed.

7. On the Save page, select the **Synchronize changes with Nodes** option and click **Save**. After the changes are saved, the administrative console returns to the home page.

8. Update the `soap.client.props` file in the *profile_root*`/properties` directory to indicate the SAF keyring names that are appropriate for your configuration. The `soap.client.props` file is used by the `wsadmin.sh` script and is located in the application server or deployment manager `(user.install.root)/properties` file. The purpose of the `soap.client.props` file is to specify the values used by SOAP clients such as `wsadmin.sh`. In a cell configured before WebSphere Application Server for z/OS maintenance level W502000, the `soap.client.props` file indicates the names of the Java key stores used by JSSE. Once your cell is using SAF keyrings for JSSE administration, verify that SAF keyrings are being used for SOAP clients.

   The `soap.client.props` file is used by the `wsadmin.sh` script.

   Changes to wsadmin client SAF keyrings require updates to the `soap.client.props` file and the creation of a keyring for administrators. Specify the following values:

   ```
   com.ibm.ssl.protocol=SSL
   com.ibm.ssl.keyStoreType=JCERACFKS
   com.ibm.ssl.keyStore=safkeyring:///yourkeyringName
   com.ibm.ssl.keyStorePassword=password
   com.ibm.ssl.trustStoreType=JCERACFKS
   com.ibm.ssl.trustStore=safkeyring:///yourKeyringName
   com.ibm.ssl.trustStorePassword=password
   ```

   =

   The password value specified does not represent a real password because you can use any string. Replace the string `yourKeyringName` with your administrative SAF keyring. The keyring name used by all WebSphere administrators and the administrative started task user ID (default `WSADMSH`) must be the same. Additionally, a keyring must be created for each user that uses the `wsadmin.sh` file with the SOAP connector when using SAF keyrings and security is enabled. (A keyring is created by the customization process for your initial administrative user ID, such as `WSADMIN`.)

A description of how to create keyrings for administrative users in SAF is described in SSL considerations for WebSphere Application Server administrators.

9. Recycle the cell.

Configuring Federal Information Processing Standard Java Secure Socket Extension files

Use this topic to configure Federal Information Processing Standard Java Secure Socket Extension files.

About this task

In WebSphere Application Server, the Java Secure Socket Extension (JSSE) provider used is the IBMJSSE2 provider. This provider delegates encryption and signature functions to the Java Cryptography Extension (JCE) provider. Consequently, IBMJSSE2 does not need to be Federal Information Processing Standard (FIPS)-approved because it does not perform cryptography. However, the JCE provider requires FIPS-approval.

WebSphere Application Server provides a FIPS-approved IBMJCEFIPS provider that IBMJSSE2 can utilize. The IBMJCEFIPS provider that is shipped in WebSphere Application Server Version 7.0 supports the following SSL ciphers:

- SSL_RSA_WITH_AES_128_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_AES_128_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_AES_128_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA

Even though the IBMJSSEFIPS provider is still present, the runtime does not use this provider. If IBMJSSEFIPS is specified as a contextProvider, WebSphere Application Server automatically defaults to the IBMJSSE2 provider (with the IBMJCEFIPS provider) for supporting FIPS. When enabling the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option on the server SSL certificate and key management panel, the runtime always uses IBMJSSE2, despite the contextProvider that you specify for SSL (IBMJSSE, IBMJSSE2 or IBMJSSEFIPS). Also, because FIPS requires the SSL protocol be TLS, the runtime always uses TLS when FIPS is enabled, regardless of the SSL protocol setting in the SSL repertoire. This simplifies the FIPS configuration in Version 7.0 because an administrator needs to enable only the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option on the server SSL certificate and key management panel to enable all transports using SSL.

1. Click **Security > SSL certificate and key management**.
2. Select the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option and click **Apply**. This option makes IBMJSSE2 and IBMJCEFIPS the active providers.
3. Accommodate Java clients that must access enterprise beans.

   Change the com.ibm.security.useFIPS property value from `false` to `true` in the *profile_root*`/properties/ssl.client.props` file.
4. Ensure that the java.security includes the provider.

   Edit the java.security file to uncomment the line with the IBMJCEFIPS provider and also renumber the rest of the provider list. The IBMJCEFIPS provider must be in the java.security file provider list. The java.security file is located in the WASHOME/java/jre/lib/security directory. To edit the file, complete the following steps:

   a. Copy the java.security file to a directory that has write permissions.

b. Edit the java.security file to comment out the line with the IBMJCE provider, uncomment the line with the IBMJCEFIPS provider, and save the file.

The IBM Software Development Kit (SDK) java.security file looks like the following example prior to completing this step:

```
#security.provider.1=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.1=com.ibm.crypto.provider.IBMJCE
security.provider.2=com.ibm.jsse.IBMJSSEProvider
security.provider.3=com.ibm.jsse2.IBMJSSEProvider2
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.crypto.pkcs11.provider.IBMPKCS11
security.provider.7=com.ibm.security.cmskeystore.CMSProvider
security.provider.8=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
```

c. Configure the security.overridePropertiesFile and java.security.properties system properties for each Java Virtual Machine (JVM) in the cell. Add the following property and value pairs:

Table 33. Custom properties for specifying a new location for the java.security file

Property name	Value
security.overrideProperties	true
java.security.properties	Specify the new location of the java.security file.

You must specify the previous set of system properties for the deployment manager, the node agent, and other application servers. For the deployment manager, specify this set of system properties for both the control and the servant. For the node agent, specify this set of system properties for the control. For all application servers, specify this set of system properties for the adjunct, control, and servant. For example, complete the following steps to specify these system properties for the control on an application server:

1) In the administrative console, click **Servers > Application servers >** *server_name*.
2) Under Server infrastructure, click **Java and Process Management > Process Definition > Control**.
3) Under Additional properties, click **Java Virtual Machine > Custom properties**.
4) Enter the properties as two sets of name and value pairs.
5) Click **Save**.

What to do next

After completing these steps, a FIPS-approved JSSE or JCE provider offers increased encryption capabilities. However, when you use FIPS-approved providers:

- By default, Microsoft Internet Explorer might not have TLS enabled. To enable TLS, open the Internet Explorer browser and click **Tools > Internet Options**. On the Advanced tab, select the Use TLS 1.0 option.

  Note: Netscape Version 4.7.*x* and earlier versions might not support TLS.
- IBM Directory Server Version 5.1 (and earlier versions) do not support TLS.
- If you have an administrative client that uses a SOAP connector and you enable FIPS, add the following line to the *profile_root*/properties/soap.client.props file:

  ```
  com.ibm.ssl.contextProvider=IBMJSSEFIPS
  ```
- When you select the **Use the Federal Information Processing Standard (FIPS)** option on the SSL certificate and key management panel, the Lightweight Third-Party Authentication (LTPA) token format is not backwards-compatible with previous releases of WebSphere Application Server. However, you can import the LTPA keys from a previous version of the application server.

Note: When enabling FIPS, you cannot configure cryptographic token devices in the SSL repertoires. IBMJSSE2 must use IBMJCEFIPS when utilizing cryptographic services for FIPS.

The following FIPS 140-2 approved cryptographic providers that are the only devices that are supported with the FIPS option:

- IBMJCEFIPS (certificate 376)
- IBM Cryptography for C (ICC) (certificate 384)

The relevant certificates are listed on the NIST Web site: Cryptographic Module Validation Program FIPS 140-1 and FIPS 140-2 Pre-validation List

To unconfigure the FIPS provider, reverse the changes that you made in the previous steps. After you reverse the changes, verify that you have made the following changes to the sas.client.props, soap.client.props, and java.security files:

- In the ssl.client.props file, you must change the com.ibm.security.useFIPS value to false.
- In the java.security file, you must change the FIPS provider to a non-FIPS provider.

  If you are using the IBM SDK java.security file, you must change the first provider to a non-FIPS provider as shown in the following example:

```
#security.provider.1=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.1=com.ibm.crypto.provider.IBMJCE
security.provider.2=com.ibm.jsse.IBMJSSEProvider
security.provider.3=com.ibm.jsse2.IBMJSSEProvider2
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
#security.provider.6=com.ibm.crypto.pkcs11.provider.IBMPKCS11
```

  If you are using the Sun JDK java.security file, you must change the third provider to a non-FIPS provider as shown in the following example:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.ibm.security.jgss.IBMJGSSProvider
security.provider.3=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.4=com.ibm.crypto.provider.IBMJCE
security.provider.5=com.ibm.jsse.IBMJSSEProvider
security.provider.6=com.ibm.jsse2.IBMJSSEProvider2
security.provider.7=com.ibm.security.cert.IBMCertPath
#security.provider.8=com.ibm.crypto.pkcs11.provider.IBMPKCS11
```

Implementing tokens for security attribute propagation

As part of an extensible architecture, WebSphere Application Server enables you to implement your own tokens in which to propagate security attributes.

About this task

The following topics are covered in this section:

- Implementing a custom propagation token
- Implementing a custom authorization token
- Implementing a custom a single sign-on token
- Implementing a custom authentication token
- Propagating a custom Java serializable object

Implementing a custom propagation token

This topic explains how you might create your own propagation token implementation, which is set on the running thread and propagated downstream.

About this task

The default propagation token usually is sufficient for propagating attributes that are not user-specific. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. You must deserialize the bytes at the target and add that information back on the thread by plugging in a custom login module into the inbound system login configurations. This task also might include encryption and decryption.

To implement a custom propagation token, you must complete the following steps:

1. Write a custom implementation of the PropagationToken interface. Many different methods are available for implementing the PropagationToken interface. However, make sure that the methods that are required by the PropagationToken interface and the token interface are fully implemented.

   After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so that it has the required permissions for the server code.

   Note: All of the token types that are defined by the propagation framework have similar interfaces. The token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the propagation token, might extend the abstract class and then most of the work is complete.

   To see an implementation of the propagation token, see “Example: com.ibm.wsspi.security.token.PropagationToken implementation” on page 1353.
2. Add and receive the custom propagation token during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. You also can add the implementation from an application. However, to deserialize the information, you need to plug in a custom login module, which is discussed in “Propagating a custom Java serializable object” on page 1387. The WSSecurityPropagationHelper class has APIs that are used to set a propagation token on the thread and to retrieve the token from the thread to make updates.

   The code sample in “Example: Custom propagation token login module” on page 1358 shows how to determine if the login is an initial login or a propagation login. The difference between these login types is whether the WSTokenHolderCallback callback contains propagation data. If the callback does not contain propagation data, initialize a new custom propagation token implementation and set it on the thread. If the callback contains propagation data, look for your specific custom propagation token TokenHolder instance, convert the byte array back into your custom PropagationToken object, and set it back on the thread. The code sample shows both instances.

   You can add attributes any time your custom propagation token is added to the thread. If you add attributes between requests and the getUniqueId method changes, the Common Secure Interoperability Version 2 (CSIv2) client session is invalidated so that it can send the new information downstream. Adding attributes between requests can affect performance. In many cases, you want the downstream requests to receive the new propagation token information.

   To add the custom propagation token to the thread, call the WSSecurityPropagationHelper.addPropagationToken method. This call requires the WebSphereRuntimePerMission ″setPropagationToken″ Java 2 Security permission.
3. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module for receiving serialized versions of your custom propagation token You can also add this login module to any of the application logins where you might want to generate your custom propagation token on the

thread during the login. Alternatively, you can generate the custom PropagationToken implementation from within your application. However, to deserialize it, you need to add the implementation to the system login modules.

For information on how to add your custom login module to the existing login configurations, see Developing custom login modules for a system login configuration.

Results

After completing these steps, you have implemented a custom PropagationToken.

Example: com.ibm.wsspi.security.token.PropagationToken implementation:

Use this file to see an example of a propagation token implementation. The following sample code does not extend an abstract class, but implements the com.ibm.wsspi.security.token.PropagationToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if considerable differences exist between how you handle the various token implementations.

For information on how to implement a custom propagation token, see “Implementing a custom propagation token” on page 1351.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomPropagationTokenImpl implements com.ibm.wsspi.security.
   token.PropagationToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
   // 2 hours in millis, by default
  private static long expire_period_in_millis = 2*60*60*1000;
 private long counter = 0;

/**
 * The constructor that is used to create initial PropagationToken instance
 */

 public CustomAbstractTokenImpl ()
 {
  // set the token version
  addAttribute("version", "1");
  // set the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis() +
        expire_period_in_millis).toString());
 }

/**
 * The constructor that is used to deserialize the token bytes received
 * during a propagation login.
 */
 public CustomAbstractTokenImpl (byte[] token_bytes)
```

```
 {
  try
  {
       hashtable = (java.util.Hashtable) com.ibm.wsspi.security.token.
         WSOpaqueTokenHelper.deserialize(token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // if you set the expiration to 0, it does not expire
  if (expiration != 0)
  {
   // return if this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }

/**
 * Gets the expiration as a long type.
 * @return long
 */
 public long getExpiration()
 {
  // get the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // expiration is the first element (should only be one)
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
     // You can choose whether your token gets propagated. In some cases
     // you might want the token to be local only.
  return true;
```

```
 }

/**
 * Gets the principal that this token belongs to. If this token is an
 * authorization token, this principal string must match the authentication
 * token principal string or the message is rejected.
 * @return String
 */
 public String getPrincipal()
 {
  // It is not necessary for the PropagationToken to return a principal,
     // because it is not user-centric.
  return "";
 }

/**
 * Returns the unique identifier of the token based upon information that
 * the provider considers makes it a unique token. This identifier is used
 * for caching purposes and might be used in combination with other token
 * unique IDs that are part of the same Subject.
 *
 * This method should return null if you want the accessID of the user to
 * represent its uniqueness. This is the typical scenario.
 *
 * @return String
 */
 public String getUniqueID()
 {
     // If you want to propagate the changes to this token, change the
     // value that this unique ID returns whenever the token is changed.
     // Otherwise, CSIv2 uses an existing session when everything else is
     // the same. This getUniqueID is checked by CSIv2 to determine the
     // session lookup.
  return counter;
 }

/**
 * Gets the bytes to be sent across the wire. The information in the byte[]
 * needs to be enough to recreate the Token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
     // Do this if the object is set to read-only during login commit
     // because this guarantees that no new data is set.
    if (isReadOnly() && tokenBytes == null)
     tokenBytes = com.ibm.wsspi.security.token.WSOpaqueTokenHelper.
               serialize(hashtable);

     // You can deserialize this in the downstream login module using
         //  WSOpaqueTokenHelper.deserialize()
    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }
```

```
/**
 * Gets the name of the token, which is used to identify the byte[] in the
 * protocol message.
 * @return String
 */
 public String getName()
 {
  return this.getClass().getName();
 }

/**
 * Gets the version of the token as a short type. This code also is used
 * to identify the byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only. The implementation
 * needs to ensure that any setter methods check that this read-only flag has
 * been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is readonly
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }
```

```
/**
 * Sets the attribute name and value pair. Returns the previous values set
 * for the key, or returns null if the value is not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
 public String[] addAttribute(String key, String value)
 {
  // Gets the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // Increments the counter to change the uniqueID
   counter++;

   // Copies the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // Allocates a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // Adds the String to the current array list
   array.add(value);

   // Adds the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // Returns the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
 }


/**
 * Gets the list of all of the attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep clone of this token. This is typically used by the session
 * logic of the CSIv2 server to create a copy of the token as it exists in the
 * session.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.websphere.security.token.CustomPropagationTokenImpl deep_clone =
   new com.ibm.websphere.security.token.CustomPropagationTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();
```

```
   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }
}
```

Example: Custom propagation token login module:

This example shows how to determine if the login is an initial login or a propagation login.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
         Map sharedState, Map options)
 {
  // (For more information on what to do during initialization, see
     // Developing custom login modules for a system login configuration.)
 }

 public boolean login() throws LoginException
 {
  // (For more information on what to do during login, see
     // Developing custom login modules for a system login configuration.)

  // Handles the WSTokenHolderCallback to see if this is an initial
     // or propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // handle exception
  }

  // Receives the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

  if (authzTokenList != null)
  {
   // Iterates through the list looking for your custom token
   for (int i=0; i<authzTokenList.size(); i++)
   {
    TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

    // Looks for the name and version of your custom PropagationToken implementation
    if (tokenHolder.getName().equals("
              com.ibm.websphere.security.token.CustomPropagationTokenImpl") &&
        tokenHolder.getVersion() == 1)
    {
     // Passes the bytes into your custom PropagationToken constructor
           //  to deserialize
     customPropToken = new
      com.ibm.websphere.security.token.CustomPropagationTokenImpl(tokenHolder.
                  getBytes());

    }
   }
  }
  else // This is not a propagation login. Create a new instance of
```

```
        // your PropagationToken implementation
 {
  // Adds a new custom propagation token. This is an initial login
  customPropToken = new com.ibm.websphere.security.token.CustomPropagationTokenImpl();

  // Adds any initial attributes
  if (customPropToken != null)
  {
   customPropToken.addAttribute("key1", "value1");
   customPropToken.addAttribute("key1", "value2");
   customPropToken.addAttribute("key2", "value1");
   customPropToken.addAttribute("key3", "something different");
  }
 }

 // Note: You can add the token to the thread during commit in case
    // something happens during the login.
}

public boolean commit() throws LoginException
{
 // For more information on what to do during commit, see
    // Developing custom login modules for a system login configuration
 if (customPropToken != null)
 {
  // Sets the propagation token on the thread
  try
  {

   System.out.println(tc, "*** ADDED MY CUSTOM PROPAGATION TOKEN TO THE THREAD ***");
   // Prints out the values in the deserialized propagation token
   java.util.Enumeration keys = customPropToken.getAttributeNames();
   while (keys.hasMoreElements())
   {
    String key = (String) keys.nextElement();
    String[] list = (String[]) customPropToken.getAttributes(key);
    for (int k=0; k<list.length; k++)
    System.out.println("Key/Value: " + key + "/" + list[k]);
   }

   // This sets it on the thread using getName() + getVersion() as the key
   com.ibm.wsspi.security.token.WSSecurityPropagationHelper.addPropagationToken(
             customPropToken);
  }
  catch (Exception e)
  {
   // Handles exception
  }


  // Now you can verify that you have set it properly by trying to get
      // it back from the thread and print the values.
  try
  {
   // This gets the PropagationToken from the thread using getName()
         // and getVersion() parameters.
   com.ibm.wsspi.security.token.PropagationToken tempPropagationToken =
    com.ibm.wsspi.security.token.WSSecurityPropagationHelper.getPropagationToken
     ("com.ibm.websphere.security.token.CustomPropagationTokenImpl", 1);

   if  (tempPropagationToken != null)
   {
    System.out.println(tc, "*** RECEIVED MY CUSTOM PROPAGATION
              TOKEN FROM THE THREAD ***");
    // Prints out the values in the deserialized propagation token
    java.util.Enumeration keys = tempPropagationToken.getAttributeNames();
    while (keys.hasMoreElements())
```

```
     {
      String key = (String) keys.nextElement();
      String[] list = (String[]) tempPropagationToken.getAttributes(key);
      for (int k=0; k<list.length; k++)
      System.out.println("Key/Value: " + key + "/" + list[k]);
     }
    }
   }
   catch (Exception e)
   {
    // Handles exception
   }
  }
 }

 // Defines your login module variables
 com.ibm.wsspi.security.token.PropagationToken customPropToken = null;

}
```

Implementing a custom authorization token

This task explains how you might create your own AuthorizationToken implementation, which is set in the login Subject and propagated downstream.

About this task

The default AuthorizationToken usually is sufficient for propagating attributes that are user-specific. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. You must deserialize the bytes at the target and add that information back on the thread. This task also might include encryption and decryption.
- Affect the overall uniqueness of the Subject using the getUniqueID() application programming interface (API).

To implement a custom authorization token, you must complete the following steps:

1. Write a custom implementation of the AuthorizationToken interface. There are many different methods for implementing the AuthorizationToken interface. However, make sure that the methods required by the AuthorizationToken interface and the token interface are fully implemented.

   After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so that it has the necessary permissions that are needed by the server code.

   Note: All of the token types defined by the propagation framework have similar interfaces. Basically, the token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the AuthorizationToken, might extend the abstract class and then most of the work is completed.

   To see an implementation of AuthorizationToken, see “Example: com.ibm.wsspi.security.token.AuthorizationToken implementation” on page 1361
2. Add and receive the custom AuthorizationToken during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. However, in order to deserialize the information, you must plug in a custom login

module, which is discussed in “Propagating a custom Java serializable object” on page 1387. After the object is instantiated in the login module, you can add the object to the Subject during the commit() method.

If you only want to add information to the Subject to get propagated, see “Propagating a custom Java serializable object” on page 1387. If you want to ensure that the information is propagated, want to do you own custom serialization, or want to specify the uniqueness for Subject caching purposes, then consider writing your own AuthorizationToken implementation.

The code sample in “Example: custom AuthorizationToken login module” on page 1366 shows how to determine if the login is an initial login or a propagation login. The difference between these login types is whether the WSTokenHolderCallback contains propagation data. If the callback does not contain propagation data, initialize a new custom AuthorizationToken implementation and set it into the Subject. If the callback contains propagation data, look for your specific custom AuthorizationToken TokenHolder instance, convert the byte[] back into your custom AuthorizationToken object, and set it back into the Subject. The code sample shows both instances.

You can make your AuthorizationToken read-only in the commit phase of the login module. If you do not make the token read-only, then attributes can be added within your applications.

3. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule for receiving serialized versions of your custom authorization token.

   Because this login module relies on information in the sharedState added by the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule, add this login module after com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule. For information on how to add your custom login module to the existing login configurations, see Developing custom login modules for a system login configuration.

Results

After completing these steps, you have implemented a custom AuthorizationToken.

Example: com.ibm.wsspi.security.token.AuthorizationToken implementation:

Use this file to see an example of a AuthorizationToken implementation. The following sample code does not extend an abstract class, but rather implements the com.ibm.wsspi.security.token.AuthorizationToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if there are considerable differences between how you handle the various token implementations.

For information on how to implement a custom AuthorizationToken, see “Implementing a custom authorization token” on page 1360.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomAuthorizationTokenImpl implements com.ibm.wsspi.security.
   token.AuthorizationToken
{
```

```
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
 private static long expire_period_in_millis = 2*60*60*1000;
  // 2 hours in millis, by default

/**
 * Constructor used to create initial AuthorizationToken instance
 */

 public CustomAuthorizationTokenImpl (String principal)
 {
  // Sets the principal in the token
  addAttribute("principal", principal);
  // Sets the token version
  addAttribute("version", "1");
  // Sets the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis() +
        expire_period_in_millis).toString());
 }

/**
 * Constructor used to deserialize the token bytes received during a
 * propagation login.
 */
 public CustomAuthorizationTokenImpl (byte[] token_bytes)
 {
  try
  {
   hashtable = (java.util.Hashtable) com.ibm.wsspi.security.token.
           WSOpaqueTokenHelper.deserialize(token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // if you set the expiration to 0, it does not expire
  if (expiration != 0)
  {
   // return if this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }

/**
 * Gets the expiration as a long.
 * @return long
 */
```

```
 public long getExpiration()
 {
  // Gets the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // The expiration is the first element. There should be only one expiration.
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
  // You can choose whether your token gets propagated. In some cases,
     // you might want it to be local only.
  return true;
 }

/**
 * Gets the principal that this Token belongs to. If this is an authorization token,
 * this principal string must match the authentication token principal string or the
 * message will be rejected.
 * @return String
 */
 public String getPrincipal()
 {
  // this might be any combination of attributes
  String[] principal = getAttributes("principal");

  if (principal != null && principal[0] != null)
  {
   return principal[0];
  }

  System.out.println("getExpiration: returning null");
  return null;
 }

/**
 * Returns a unique identifier of the token based upon the information that provider
 * considers makes this a unique token. This will be used for caching purposes
 * and might be used in combination with other token unique IDs that are part of
 * the same Subject.
 *
 * This method should return null if you want the accessID of the user to represent
 * uniqueness.  This is the typical scenario.
 *
 * @return String
 */
 public String getUniqueID()
 {
  // if you don't want to affect the cache lookup, just return NULL here.
  // return null;

  String cacheKeyForThisToken = "dynamic attributes";

  // if you do want to affect the cache lookup, return a string of
     // attributes that you want factored into the lookup.
```

```
  return cacheKeyForThisToken;
 }

/**
 * Gets the bytes to be sent across the wire. The information in the byte[]
 * needs to be enough to recreate the Token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
    // Do this if the object is set to read-only during login commit,
    // because this makes sure that no new data gets set.
    if (isReadOnly() && tokenBytes == null)
     tokenBytes = com.ibm.wsspi.security.token.WSOpaqueTokenHelper.
                serialize(hashtable);

    // You can deserialize this in the downstream login module using
          // WSOpaqueTokenHelper.deserialize()
    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }

/**
 * Gets the name of the token used to identify the byte[] in the protocol message.
 * @return String
 */
 public String getName()
 {
  return this.getClass().getName();
 }

/**
 * Gets the version of the token as an short. This also is used to identify the
 * byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only. The implementation
 * needs to ensure that any setter methods check that this flag has been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
```

```
 }

/**
 * Called internally to see if the token is read-only
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name and value pair. Returns the previous values set for key,
 * or null if not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
 public String[] addAttribute(String key, String value)
 {
  // Gets the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // Copies the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // Allocates a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // Adds the String to the current array list
   array.add(value);

   // Adds the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // Returns the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
```

```
 }


/**
 * Gets the list of all attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep copying of this token, if necessary.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.websphere.security.token.CustomAuthorizationTokenImpl deep_clone =
   new com.ibm.websphere.security.token.CustomAuthorizationTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }
}
```

Example: custom AuthorizationToken login module:

This file shows how to determine if the login is an initial login or a propagation login.

For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
  // Handles the WSTokenHolderCallback to see if this is an initial or
     // propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // Handles exception
```

```
 }

 // Receives the ArrayList of TokenHolder objects (the serialized tokens)
 List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

 if (authzTokenList != null)
 {
  // Iterates through the list looking for your custom token
  for (int i=0; i
  for (int i=0; i<authzTokenList.size(); i++)
  {
   TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

         // Looks for the name and version of your custom AuthorizationToken
         // implementation
   if (tokenHolder.getName().equals("com.ibm.websphere.security.token.
            CustomAuthorizationTokenImpl") &&
       tokenHolder.getVersion() == 1)
   {
          // Passes the bytes into your custom AuthorizationToken constructor
          // to deserialize
    customAuthzToken = new
     com.ibm.websphere.security.token.CustomAuthorizationTokenImpl(
                 tokenHolder.getBytes());

   }
  }
 }
 else
    // This is not a propagation login. Create a new instance of your
    // AuthorizationToken implementation
 {
      // Gets the prinicpal from the default AuthenticationToken. This must match
      // all tokens.
  defaultAuthToken = (com.ibm.wsspi.security.token.AuthenticationToken)
   sharedState.get(com.ibm.wsspi.security.auth.callback.Constants.WSAUTHTOKEN_KEY);
  String principal = defaultAuthToken.getPrincipal();

      // Adds a new custom authorization token. This is an initial login. Pass the
      // principal into the constructor
  customAuthzToken = new com.ibm.websphere.security.token.
          CustomAuthorizationTokenImpl(principal);

  // Adds any initial attributes
  if (customAuthzToken != null)
  {
   customAuthzToken.addAttribute("key1", "value1");
   customAuthzToken.addAttribute("key1", "value2");
   customAuthzToken.addAttribute("key2", "value1");
   customAuthzToken.addAttribute("key3", "something different");
  }
 }

    // Note: You can add the token to the Subject during commit in case something
    // happens during the login.
}

public boolean commit() throws LoginException
{
 if (customAut  // (hzToken != null)
 {
  // sSets the customAuthzToken token into the Subject
  try
  {
   public final AuthorizationToken customAuthzTokenPriv = customAuthzToken;
         // Do this in a doPrivileged code block so that application code does not
         // need to add additional permissions
```

```
    java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
    {
     public Object run()
     {
      try
      {
                 // Adds the custom authorization token if it is not null
                 //  and not already in the Subject
                                 if ((customAuthzTokenPriv != null) &&
        (!subject.getPrivateCredentials().contains(customAuthzTokenPriv)))
       {
        subject.getPrivateCredentials().add(customAuthzTokenPriv);
       }
      }
      catch (Exception e)
      {
       throw new WSLoginFailedException (e.getMessage(), e);
      }

      return null;
     }
    });
   }
   catch (Exception e)
   {
    throw new WSLoginFailedException (e.getMessage(), e);
   }
  }
 }

 // Defines your login module variables
 com.ibm.wsspi.security.token.AuthorizationToken customAuthzToken = null;
 com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
 java.util.Map _sharedState = null;
}
```

Implementing a custom single sign-on token

You can create your own single sign-on token implementation. The single sign-on token implementation is set in the login Subject and added to the HTTP response as an HTTP cookie.

About this task

The cookie name is the concatenation of the SingleSignonToken.getName application programming interface (API) and the SingleSignonToken.getVersion API. There is no delimiter. When you add a single sign-on token to the Subject, it also gets propagated horizontally and downstream in case the Subject is used for other Web requests. You must deserialize your custom single sign-on token when you receive it from a propagation login. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. Encrypt the information because it is out to the HTTP response and is available on the Internet. You must deserialize or decrypt the bytes at the target and add that information back into the Subject.
- Affect the overall uniqueness of the Subject using the getUniqueID API.

To implement a custom single sign-on token, complete the following steps:

1. Write a custom implementation of the SingleSignonToken interface.

   Many different methods are available for implementing the SingleSignonToken interface. However, make sure the methods that are required by the SingleSignonToken interface and the token interface are fully implemented.

After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so that it has the required permissions for the server code.

Note: All of the token types that are defined by the propagation framework have similar interfaces. Basically, the token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the single sign-on token, might extend the abstract class and then most of the work is complete.

To see an implementation of the single sign-on token, see "Example: A com.ibm.wsspi.security.token.SingleSignonToken implementation" on page 1370

2. Add and receive the custom single sign-on token during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. However, to deserialize the information, you need to plug in a custom login module, which is discussed in a subsequent step. After the object is instantiated in the login module, you can add it to the Subject during the commit method.

   The code sample in "Example: A custom single sign-on token login module" on page 1374, shows how to determine if the login is an initial login or a propagation login. The difference is whether the WSTokenHolderCallback callback contains propagation data. If the callback does not contain propagation data, initialize a new custom single sign-on token implementation and set it into the Subject. Also, look for the HTTP cookie from the HTTP request if the HTTP request object is available in the callback. You can get your custom single sign-on token both from a horizontal propagation login and from the HTTP request. However, it is recommended that you make the token available in both places because then the information arrives at any front-end application server, even if that server does not support propagation.

   You can make your single sign-on token read-only in the commit phase of the login module. If you make the token read-only, additional attributes cannot be added within your applications.

   Note:

   - HTTP cookies have a size limitation. Size restrictions should be included in the documentation for your specific browser.
   - The WebSphere Application Server runtime does not handle cookies that it does not generate, so this cookie is not used by the runtime.
   - The SingleSignonToken object, when in the Subject, does affect the cache lookup of the Subject if you return something in the getUniqueID method.

3. Get the HTTP cookie from the HTTP request object during login or from an application. The sample code that is found in "Example: An HTTP cookie retrieval" on page 1376 shows how you can retrieve the HTTP cookie from the HTTP request, decode the cookie so that it is back to your original bytes, and create your custom SingleSignonToken object from the bytes.
4. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule for receiving serialized versions of your custom propagation token. Because this login module relies on information in the sharedState state that is added by the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module, add this login module after the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module.

   For information on adding your custom login module into the existing login configurations, see Developing custom login modules for a system login configuration.

Results

After completing these steps, you have implemented a custom single sign-on token.

Example: A com.ibm.wsspi.security.token.SingleSignonToken implementation:

Use this file to see an example of a single sign-on implementation. The following sample code does not extend an abstract class, but implements the com.ibm.wsspi.security.token.SingleSignonToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if considerable differences exist between how you handle the various token implementations.

For information on how to implement a custom single sign-on token, see “Implementing a custom single sign-on token” on page 1368.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomSingleSignonTokenImpl implements com.ibm.wsspi.security.
   token.SingleSignonToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
   // 2 hours in millis, by default
 private static long expire_period_in_millis = 2*60*60*1000;

/**
 * Constructor used to create initial SingleSignonToken instance
 */

 public CustomSingleSignonTokenImpl (String principal)
 {
  // set the principal in the token
  addAttribute("principal", principal);
  // set the token version
  addAttribute("version", "1");
  // set the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis() +
        expire_period_in_millis).toString());
 }

/**
 * Constructor used to deserialize the token bytes received during a propagation login.
 */
 public CustomSingleSignonTokenImpl (byte[] token_bytes)
 {
  try
  {
   // you should implement a decryption algorithm to decrypt the cookie bytes
   hashtable = (java.util.Hashtable) some_decryption_algorithm (token_bytes);
  }
  catch (Exception e)
```

```
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // if you set the expiration to 0, it does not expire
  if (expiration != 0)
  {
   // return if this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }

/**
 * Gets the expiration as a long.
 * @return long
 */
 public long getExpiration()
 {
  // get the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // expiration will always be the first element (should only be one)
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
  // You can choose whether your token gets propagated or not, in some cases
     // you might want it to be local only.
  return true;
 }

/**
 * Gets the principal that this Token belongs to. If this is an authorization token,
 * this principal string must match the authentication token principal string or the
 * message will be rejected.
 * @return String
```

```
 */
 public String getPrincipal()
 {
  // this could be any combination of attributes
  String[] principal = getAttributes("principal");

  if (principal != null && principal[0] != null)
  {
   return principal[0];
  }

  System.out.println("getExpiration: returning null");
  return null;
 }

/**
 * Returns a unique identifier of the token based upon information the provider
 * considers makes this a unique token.  This will be used for caching purposes
 * and may be used in combination with other token unique IDs that are part of
 * the same Subject.
 *
 * This method should return null if you want the access ID of the user to represent
 * uniqueness.  This is the typical scenario.
 *
 * @return String
 */
 public String getUniqueID()
 {
  // this could be any combination of attributes
  return getPrincipal();
 }

/**
 * Gets the bytes to be sent across the wire.  The information in the byte[]
 * needs to be enough to recreate the Token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
    // do this if the object is set read-only during login commit,
    // since this guarantees no new data gets set.
    if (isReadOnly() && tokenBytes == null)
     tokenBytes = some_encryption_algorithm (hashtable);

    // you can deserialize the tokenBytes using a similiar decryption algorithm.
    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }

/**
 * Gets the name of the token, used to identify the byte[] in the protocol message.
 * @return String
 */
 public String getName()
```

```
 {
  return "myCookieName";
 }

/**
 * Gets the version of the token as a short.  This is also used to identify the
 * byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only.  The implementation
 * needs to ensure any setter methods check that this has been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is readonly
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name/value pair.  Returns the previous values set for key,
 * or null if not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
```

```
 public String[] addAttribute(String key, String value)
 {
  // get the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // copy the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // allocate a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // add the String to the current array list
   array.add(value);

   // add the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // return the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
 }


/**
 * Gets the List of all attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep copying of this token, if necessary.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.websphere.security.token.CustomSingleSignonImpl deep_clone =
   new com.ibm.websphere.security.token.CustomSingleSignonTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }
}
```

Example: A custom single sign-on token login module:

This file shows how to determine if the login is an initial login or a propagation login.

For information on initialization and on what to do during login and commit, see Developing custom login modules for a system login configuration.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
     // Handles the WSTokenHolderCallback to see if this is an initial or
     // propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // handle exception
  }

  // Receives the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

  if (authzTokenList != null)
  {
   // iterate through the list looking for your custom token
   for (int i=0; i
   for (int i=0; i<authzTokenList.size(); i++)
   {
    TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

    // Looks for the name and version of your custom SingleSignonToken
          // implementation
    if (tokenHolder.getName().equals("myCookieName")
              && tokenHolder.getVersion() == 1)
    {
     // Passes the bytes into your custom SingleSignonToken constructor
            // to deserialize
     customSSOToken = new
      com.ibm.websphere.security.token.CustomSingleSignonTokenImpl
                  (tokenHolder.getBytes());

    }
   }
  }
  else
          // This is not a propagation login. Create a new instance of your
          // SingleSignonToken implementation
  {
       // Gets the principal from the default SingleSignonToken. This principal
       //  must match all tokens.
   defaultAuthToken = (com.ibm.wsspi.security.token.AuthenticationToken)
    sharedState.get(com.ibm.wsspi.security.auth.callback.Constants.WSAUTHTOKEN_KEY);
   String principal = defaultAuthToken.getPrincipal();

   // Adds a new custom single sign-on (SSO) token. This is an initial login.
       //  Pass the principal into the constructor
   customSSOToken = new com.ibm.websphere.security.token.
```

```
          CustomSingleSignonTokenImpl(principal);

  // add any initial attributes
  if (customSSOToken != null)
  {
   customSSOToken.addAttribute("key1", "value1");
   customSSOToken.addAttribute("key1", "value2");
   customSSOToken.addAttribute("key2", "value1");
   customSSOToken.addAttribute("key3", "something different");
  }
 }

    // Note: You can add the token to the Subject during commit in case something
    // happens during the login.
}

public boolean commit() throws LoginException
{
 if (customSSOToken != null)
 {
  // Sets the customSSOToken token into the Subject
  try
  {
   public final SingleSignonToken customSSOTokenPriv = customSSOToken;
         // Do this in a doPrivileged code block so that application code does not
         // need to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
   {
    public Object run()
    {
     try
     {
      // Adds the custom SSO token if it is not null and
                //  not already in the Subject
                              if ((customSSOTokenPriv != null) &&
        (!subject.getPrivateCredentials().
                         contains(customSSOTokenPriv)))
      {
       subject.getPrivateCredentials().
                      add(customSSOTokenPriv);
      }
     }
     catch (Exception e)
     {
      throw new WSLoginFailedException (e.getMessage(), e);
     }

     return null;
    }
   });
  }
  catch (Exception e)
  {
   throw new WSLoginFailedException (e.getMessage(), e);
  }
 }
}

// Defines your login module variables
com.ibm.wsspi.security.token.SingleSignonToken customSSOToken = null;
com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
java.util.Map _sharedState = null;
}
```

Example: An HTTP cookie retrieval:

The following example shows you how to retrieve a cookie from an HTTP request, decode the cookie so that it is back to your original bytes, and create your custom SingleSignonToken object from the bytes. This example shows how to complete these steps from a login module. However, you also can complete these steps using a servlet.

For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
     // Handles the WSTokenHolderCallback to see if this is an
     // initial or propagation login.
  Callback callbacks[] = new Callback[2];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");
  callbacks[1] = new WSServletRequestCallback("HttpServletRequest: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // Handles the exception
  }

  // receive the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();
  javax.servlet.http.HttpServletRequest request =
         ((WSServletRequestCallback) callbacks[1]).getHttpServletRequest();

  if (request != null)
  {

   // Checks if the cookie is present
   javax.servlet.http.Cookie[] cookies = request.getCookies();
   String[] cookieStrings = getCookieValues (cookies, "myCookeName1");

   if (cookieStrings != null)
   {
    String cookieVal = null;
    for (int n=0;n<cookieStrings.length;n++)
    {
     cookieVal = cookieStrings[n];
     if (cookieVal.length()>0)
     {
               // Removes the cookie encoding from the cookie to get
               // your custom bytes
      byte[] cookieBytes =
       com.ibm.websphere.security.WSSecurityHelper.
                     convertCookieStringToBytes(cookieVal);
      customSSOToken =
       new com.ibm.websphere.security.token.
                     CustomSingleSignonTokenImpl(cookieBytes);

               // Now that you have your cookie from the request,
               // you can do something with it here, or add it
               // to the Subject in the commit() method for use later.
     if (debug || tc.isDebugEnabled())
```

```
     {
      System.out.println("*** GOT MY CUSTOM SSO TOKEN FROM
                    THE REQUEST ***");
     }
    }
   }
  }
 }

}

public boolean commit() throws LoginException
{
 if (customSSOToken != null)
 {
  // Sets the customSSOToken token into the Subject
  try
  {
   public final SingleSignonToken customSSOTokenPriv = customSSOToken;
         // Do this in a doPrivileged code block so that application code does not
         // need to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
   {
    public Object run()
    {
     try
     {
                 // Add the custom SSO token if it is not null and not
                 // already in the Subject
                              if ((customSSOTokenPriv != null) &&
        (!subject.getPrivateCredentials().
                        contains(customSSOTokenPriv)))
      {
       subject.getPrivateCredentials().add(customSSOTokenPriv);
      }
     }
     catch (Exception e)
     {
      throw new WSLoginFailedException (e.getMessage(), e);
     }

     return null;
    }
   });
  }
  catch (Exception e)
  {
   throw new WSLoginFailedException (e.getMessage(), e);
  }
 }
}

// Private method to get the specific cookie from the request
private String[] getCookieValues (Cookie[] cookies, String hdrName)
{
 Vector retValues = new Vector();
 int numMatches=0;
 if (cookies != null)
 {
  for (int i = 0; i < cookies.length; ++i)
  {
   if (hdrName.equals(cookies[i].getName()))
   {
    retValues.add(cookies[i].getValue());
    numMatches++;
    System.out.println(cookies[i].getValue());
   }
```

```
   }
  }

  if (retValues.size()>0)
   return (String[]) retValues.toArray(new String[numMatches]);
  else
   return null;
 }

 // Defines your login module variables
 com.ibm.wsspi.security.token.SingleSignonToken customSSOToken = null;
 com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
 java.util.Map _sharedState = null;
}
```

Implementing a custom authentication token

This topic explains how you might create your own authentication token implementation, which is set in the login Subject and propagated downstream.

About this task

With this implementation you can specify an authentication token that can be used by a custom login module or application. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. You must deserialize the bytes at the target and add that information back on the thread. This task also might include encryption and decryption.
- Affect the overall uniqueness of the Subject using the getUniqueID application programming interface (API).

Note: Custom authentication token implementations are not used by the security runtime in WebSphere Application Server to enforce authentication. WebSphere Application Security runtime uses this token in the following situations only:

- Call the getBytes method for serialization
- Call the getForwardable method to determine whether to serialize the authentication token.
- Call the getUniqueId method for uniqueness
- Call the getName and the getVersion methods for adding serialized bytes to the token holder that is sent downstream

All of the other uses are custom implementations.

To implement a custom authentication token, you must complete the following steps:

1. Write a custom implementation of the AuthenticationToken interface. Many different methods are available for implementing the AuthenticationToken interface. However, make sure the methods that are required by the AuthenticationToken interface and the token interface are fully implemented. After you implement this interface, you can place it in the *install_dir*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so the class has the necessary permissions required by the server code.

   Note: All of the token types that are defined by the propagation framework have similar interfaces. The token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the authentication token, might extend the abstract class and then most of the work is complete.

To see an implementation of the AuthenticationToken interface, see “Example: A com.ibm.wsspi.security.token.AuthenticationToken implementation.”

2. Add and receive the custom authentication token during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. However, to deserialize the information you must plug in a custom login module. After the object is instantiated in the login module, you can add the object to the Subject during the commit method.

   If you only want to add information to the Subject to get propagated, see “Propagating a custom Java serializable object” on page 1387. If you want to ensure that the information is propagated, do your own custom serialization, or specify the uniqueness for Subject caching purposes, consider writing your own authentication token implementation.

   The code sample in “Example: A custom authentication token login module” on page 1385, shows how to determine if the login is an initial login or a propagation login. The difference between these login types is whether the WSTokenHolderCallback callback contains propagation data. If the callback does not contain propagation data, initialize a new custom authentication token implementation and set it into the Subject. If the callback contains propagation data, look for your specific custom authentication token TokenHolder instance, convert the byte array back into your custom AuthenticationToken object, and set it back into the Subject. The code sample shows both instances.

   You can make your authentication token read-only in the commit phase of the login module. If you do not make the token read-only, attributes can be added within your applications.

3. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module for receiving serialized versions of your custom authorization token.

   Because this login module relies on information in the shared state that is added by the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module, add this login module after the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module. For information on how to add your custom login module to the existing login configurations, see Developing custom login modules for a system login configuration.

Results

After completing these steps, you have implemented a custom authentication token.

Example: A com.ibm.wsspi.security.token.AuthenticationToken implementation:

The following example illustrates an authentication token implementation. The following sample code does not extend an abstract class, but rather implements the com.ibm.wsspi.security.token.AuthenticationToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if considerable differences exist between how you handle the various token implementations.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomAuthenticationTokenImpl implements com.ibm.wsspi.security.
```

```
   token.AuthenticationToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
  // 2 hours in millis, by default
   private static long expire_period_in_millis = 2*60*60*1000;
 private String oidName = "your_oid_name";
  // This string can really be anything if you do not want to use an OID.

/**
 * Constructor used to create initial AuthenticationToken instance
 */
 public CustomAuthenticationTokenImpl (String principal)
 {
  // Sets the principal in the token
  addAttribute("principal", principal);
  // Sets the token version
  addAttribute("version", "1");
  // Sets the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis()
        + expire_period_in_millis).toString());
 }

/**
 * Constructor used to deserialize the token bytes received during a
 * propagation login.
 */
 public CustomAuthenticationTokenImpl (byte[] token_bytes)
 {
  try
  {
       // The data in token_bytes should be signed and encrypted if the
       // hashtable is acting as an authentication token.
   hashtable = (java.util.Hashtable) custom_decryption_algorithm (token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // If you set the expiration to 0, the token does not expire
  if (expiration != 0)
  {
   // Returns a response that identifies whether this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }
```

```
/**
 * Gets the expiration as a long type.
 * @return long
 */
 public long getExpiration()
 {
  // Gets the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // The expiration is the first element and there should only be one expiration
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
     // You can choose whether your token gets propagated. In some cases
     // you might want it to be local only.
  return true;
 }

/**
 * Gets the principal to which this token belongs. If this is an
 * authorization token, this principal string must match the
 * authentication token principal string or the message is rejected.
 * @return String
 */
 public String getPrincipal()
 {
  // This value might be any combination of attributes
  String[] principal = getAttributes("principal");

  if (principal != null && principal[0] != null)
  {
   return principal[0];
  }

  System.out.println("getExpiration: returning null");
  return null;
 }

/**
 * Returns a unique identifier of the token based upon information the provider
 * considers makes this a unique token. This identifier is used for caching purposes
 * and can be used in combination with other token unique IDs that are part of
 * the same Subject.
 *
 * This method should return null if you want the accessID of the user to represent
 * uniqueness.  This is the typical scenario.
 *
 * @return String
 */
 public String getUniqueID()
 {
     // If you do not want to affect the cache lookup, just return NULL here.
     return null;
```

```
  String cacheKeyForThisToken = "dynamic attributes";

     // If you do want to affect the cache lookup, return a string of
     // attributes that you want factored into the lookup.
  return cacheKeyForThisToken;
 }

/**
 * Gets the bytes to be sent across the wire. The information in the byte[]
 * needs to be enough to recreate the token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
          // Do this if the object is set read-only during login commit
          // because this ensures that new data is not set.
    if (isReadOnly() && tokenBytes == null)
     tokenBytes = custom_encryption_algorithm (hashtable);

    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }

/**
 * Gets the name of the token, which is used to identify the byte[] in the
 * protocol message.
 * @return String
 */
 public String getName()
 {
  return oidName;
 }

/**
 * Gets the version of the token as a short type.  This also is used
 * to identify the byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only.  The implementation
 * needs to ensure that any set methods check that this state has been set.
 */
 public void setReadOnly()
```

```
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is read-only
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name/value pair.  Returns the previous values set for key,
 * or null if not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
 public String[] addAttribute(String key, String value)
 {
  // Gets the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // Copies the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // Allocates a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // Adds the String to the current array list
   array.add(value);

   // Adds the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // Returns the old array
   return old_array;
  }
```

```
  return (String[]) array.toArray(new String[0]);
 }


/**
 * Gets the list of all attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep copying of this token, if necessary.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.wsspi.security.token.AuthenticationToken deep_clone =
   new com.ibm.websphere.security.token.CustomAuthenticationTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }

/**
 * This method returns true if this token is storing a user ID and password
 * instead of a token.
 * @return boolean
 */
 public boolean isBasicAuth()
 {
  return false;
 }
}
```

Example: A custom authentication token login module:

This examples shows how to determine if the login is an initial login or a propagation login.

For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
    Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
```

```
// Handles the WSTokenHolderCallback to see if this is an initial or
   // propagation login.
Callback callbacks[] = new Callback[1];
callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

try
{
 callbackHandler.handle(callbacks);
}
catch (Exception e)
{
 // Handles exception
}

// Receives the ArrayList of TokenHolder objects (the serialized tokens)
List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

if (authzTokenList != null)
{
 // Iterates through the list looking for your custom token
 for (int i=0; i<authzTokenList.size(); i++)
 {
  TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

        // Looks for the name and version of your custom AuthenticationToken
        // implementation
  if (tokenHolder.getName().equals("your_oid_name") && tokenHolder.getVersion() == 1)
  {
          // Passes the bytes into your custom AuthenticationToken constructor
          // to deserialize
   customAuthzToken = new
    com.ibm.websphere.security.token.
             CustomAuthenticationTokenImpl(tokenHolder.getBytes());

  }
 }
}
else
        // This is not a propagation login. Create a new instance of your
        // AuthenticationToken implementation
{
      //  Gets the principal from the default AuthenticationToken. This principal
      //  should match all default tokens.
      //  Note: WebSphere Application Server runtime only enforces this for
      //  default tokens. Thus, you can choose
      //  to do this for custom tokens, but it is not required.
 defaultAuthToken = (com.ibm.wsspi.security.token.AuthenticationToken)
  sharedState.get(com.ibm.wsspi.security.auth.callback.Constants.WSAUTHTOKEN_KEY);
 String principal = defaultAuthToken.getPrincipal();

     // Adds a new custom authentication token. This is an initial login. Pass
     // the principal into the constructor
 customAuthToken = new com.ibm.websphere.security.token.
        CustomAuthenticationTokenImpl(principal);

 // Adds any initial attributes
 if (customAuthToken != null)
 {
  customAuthToken.addAttribute("key1", "value1");
  customAuthToken.addAttribute("key1", "value2");
  customAuthToken.addAttribute("key2", "value1");
  customAuthToken.addAttribute("key3", "something different");
 }
}

   // Note: You can add the token to the Subject during commit in case
   // something happens during the login.
```

```
}

public boolean commit() throws LoginException
{
 if (customAuthToken != null)
 {
  // Sets the customAuthToken token into the Subject
  try
  {
   private final AuthenticationToken customAuthTokenPriv = customAuthToken;
         // Do this in a doPrivileged code block so that application code does
         // not need to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
   {
    public Object run()
    {
     try
     {
                 // Adds the custom Authentication token if it is not
                 // null and not already in the Subject
                              if ((customAuthTokenPriv != null) &&
        (!subject.getPrivateCredentials().
                        contains(customAuthTokenPriv)))
      {
       subject.getPrivateCredentials().add(customAuthTokenPriv);
      }
     }
     catch (Exception e)
     {
      throw new WSLoginFailedException (e.getMessage(), e);
     }

     return null;
    }
   });
  }
  catch (Exception e)
  {
   throw new WSLoginFailedException (e.getMessage(), e);
  }
 }
}

// Defines your login module variables
com.ibm.wsspi.security.token.AuthenticationToken customAuthToken = null;
com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
java.util.Map _sharedState = null;
}
```

Propagating a custom Java serializable object

This document describes how to add an object into the Subject from a login module and describes other infrastructure considerations to make sure that the Java object gets propagated.

Before you begin

Prior to completing this task, verify that security propagation is enabled in the administrative console.

About this task

With security attribute propagation enabled, you can propagate data either horizontally with single sign-on (SSO) enabled or downstream using Common Secure Interoperability Version 2 (CSIv2). When a login occurs, either through an application login configuration or a system login configuration, a custom login module can be plugged in to add Java serialized objects into the Subject during login. This document describes how to add an object into the Subject from a login module and describes other infrastructure

considerations to make sure that the Java object gets propagated.

1. Add your custom Java object into the Subject from a custom login module. A two-phase process exists for each Java Authentication and Authorization Service (JAAS) login module. WebSphere Application Server completes the following processes for each login module present in the configuration:

   login method
   In this step, the login configuration callbacks are analyzed, if necessary, and the new objects or credentials are created.

   commit method
   In this step, the objects or credentials that are created during login are added into the Subject.

   After a custom Java object is added into the Subject, WebSphere Application Server serializes the object on the sending server, deserializes the object on the receiving server, and adds the object back into the Subject downstream. However, some requirements exist for this process to occur successfully. For more information on the JAAS programming model, see the JAAS information provided in Security: Resources for learning.

   Note: Whenever you plug a custom login module into the login infrastructure of WebSphere Application Server, make sure that the code is trusted. When you put the classes together in a Java archive (JAR) file and add the file to the *app_server_root*/lib/ext/ directory, the login module has Java 2 Security AllPermissions permissions . It is recommended that you add your login module and other infrastructure classes into any private directory. However, you must modify the *profile_root*/properties/server.policy file to make sure that your private directory, Java archive (JAR) file, or both have the permissions required to run the application programming interfaces (API) that are called from the login module. Because the login module might be run after the application code on the call stack, you might add doPrivileged code so that you do not need to add additional properties to your applications.

   The following code sample shows how to add doPrivileged code. For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {

 }

 public boolean login() throws LoginException
 {
     // Construct callback for the WSTokenHolderCallback so that you
     // can determine if
     // your custom object has propagated
    Callback callbacks[] = new Callback[1];
    callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

    try
    {
         _callbackHandler.handle(callbacks);
    }
    catch (Exception e)
    {
    throw new LoginException (e.getLocalizedMessage());
    }

    // Checks to see if any information is propagated into this login
    List authzTokenList = ((WSTokenHolderCallback) callbacks[1]).
            getTokenHolderList();

    if (authzTokenList != null)
    {
        for (int i = 0; i< authzTokenList.size(); i++)
```

```
           {
          TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

                //  Look for your custom object. Make sure you use
                //  "startsWith"because there is some data appended
                //  to the end of the name indicating in which Subject
                //  Set it belongs. Example from getName():
                //  "com.acme.CustomObject (1)". The class name is
                //  generated at the sending side by calling the
                //  object.getClass().getName() method. If this object
                //  is deserialized by WebSphere Application Server,
                //  then return it and you do not need to add it here.
                //  Otherwise, you can add it below.
                //  Note: If your class appears in this list and does
                //  not use custom serialization (for example, an
                //  implementation of the Token interface described in
                //  the Propagation Token Framework), then WebSphere
                //  Application  Server automatically deserializes the
                //  Java object for you. You might just return here if
                //  it is found in the list.

          if (tokenHolder.getName().startsWith("com.acme.CustomObject"))
         return true;
      }
   }
      //  If you get to this point, then your custom object has not propagated
       myCustomObject = new com.acme.CustomObject();
       myCustomObject.put("mykey", "mydata");
}

public boolean commit() throws LoginException
{
 try
 {
      // Assigns a reference to a final variable so it can be used in
      // the doPrivileged block
  final com.acme.CustomObject myCustomObjectFinal = myCustomObject;
  // Prevents your applications from needing a JAAS getPrivateCredential
      // permission.
  java.security.AccessController.doPrivileged(new java.security.
          PrivilegedExceptionAction()
  {
   public Object run() throws java.lang.Exception
   {
           // Try not to add a null object to the Subject or an object
           // that already exists.
    if (myCustomObjectFinal != null && !subject.getPrivateCredentials().
               contains(myCustomObjectFinal))
    {
              // This call requires a special Java 2 Security permission,
              // see the JAAS application programming interface (API)
              // documentation.
     subject.getPrivateCredentials().add(myCustomObjectFinal);
    }
    return null;
   }
  });
 }
 catch (java.security.PrivilegedActionException e)
 {
  // Wraps the exception in a WSLoginFailedException
  java.lang.Throwable myException = e.getException();
  throw new WSLoginFailedException (myException.getMessage(), myException);
 }
}
```

```
 // Defines your login module variables
 com.acme.CustomObject myCustomObject = null;
}
```

2. Verify that your custom Java class implements the java.io.Serializable interface. An object that is added to the Subject must be serialized if you want the object to propagate. For example, the object must implement the java.io.Serializable interface. If the object is not serialized, the request does not fail, but the object does not propagate. To make sure an object that is added to the Subject is propagated, implement one of the token interfaces that is defined in Security attribute propagation or add attributes to one of the following existing default token implementations:

   AuthorizationToken
   Add attributes if they are user-specific. For more information, see Using the default authorization token.

   PropagationToken
   Add attributes that are specific to an invocation. For more information, see Using the default propagation token.

   If you are careful adding custom objects and follow all the steps to make sure that WebSphere Application Server can serialize and deserialize the object at each hop, then it is sufficient to use custom Java objects only.

3. Verify that your custom Java class exists on all of the systems that might receive the request.

   When you add a custom object into the Subject and expect WebSphere Application Server to propagate the object, put the class definitions together in a Java archive (JAR) file and add the file to the *app_server_root*/lib/ext/ directory on all of the nodes where serialization or deserialization might occur. Also, verify that the Java class versions are the same.

4. Verify that your custom login module is configured in all of the login configurations used in your environment where you need to add your custom object during a login. Any login configuration that interacts with WebSphere Application Server generates a Subject that might be propagated outbound for an Enterprise JavaBeans (EJB) request. If you want WebSphere Application Server to propagate a custom object in all cases, make sure that the custom login module is added to every login configuration that is used in your environment. For more information, see Developing custom login modules for a system login configuration.

5. Verify that security attribute propagation is enabled on all of the downstream servers that receive the propagated information. When an EJB request is sent to a downstream server and security attribute propagation is disabled on that server, only the authentication token is sent for backwards compatibility. Therefore, you must review the configuration to verify that propagation is enabled in all of the cells that might receive requests. You must check several places in the administrative console to make sure propagation is fully enabled. For more information, see Propagating security attributes among application servers.

6. Add any custom objects to the propagation exclude list that you do not want to propagate. You can configure a property to exclude the propagation of objects that match specific class names, package names, or both. For example, you can have a custom object that is related to a specific process. If the object is propagated, it does not contain valid information. You must tell WebSphere Application Server not to propagate this object. Complete the following steps to specify the object in the propagation exclude list, using the administrative console:

   a. Click **Security > Global security > Custom properties > New**.

   b. Add `com.ibm.ws.security.propagationExcludeList` in the **Name** field.

   c. Add the name of the custom object in the **Value** field. You can add a list of custom objects to the propagation exclude list, separated by a colon (:). For example, you might enter `com.acme.CustomLocalObject:com.acme.private.*`. You can enter a class name such as `com.acme.CustomLocalObject` or a package name such as `com.acme.private.*`. In this example, WebSphere Application Server does not propagate any class that equals `com.acme.CustomLocalObject` or begins with `com.acme.private.`

Although you can add custom objects to the propagation exclude list, you must be aware of a side effect. WebSphere Application Server stores the opaque token, or the serialized Subject contents, in a local cache for the life of the single sign-on (SSO) token. The life of the SSO token, which has a default of two hours, is configured in the SSO properties on the administrative console. The information that is added to the opaque token includes only the objects not in the exclude list.

If your authentication cache does not match your SSO token timeout, configure the authentication cache properties. See Configuring the authentication cache. It is recommended that you make your authentication cache timeout value equal to the SSO token timeout.

Results

As a result of this task, custom Java serializable objects are propagated horizontally or downstream. For more information on the differences between horizontal and downstream propagation, see Security attribute propagation.

Developing a custom interceptor for trust associations

You can define the interceptor class method that you want to use. WebSphere Application Server supports two trust association interceptor interfaces: com.ibm.websphere.security.TrustAssociationInterceptor and com.ibm.wsspi.security.tai.TrustAssociationInterceptor.

Before you begin

If you are using a third party reverse proxy server other than Tivoli® WebSEAL, you must provide an implementation class for the product interceptor interface for your proxy server. This article describes the com.ibm.websphere.security.TrustAssociationInterceptor.java interface that you must implement.

Note: The Trust Association Interceptor (TAI) interface (com.ibm.wsspi.security.tai.TrustAssociationInterceptor) supports several new features and is different from the existing com.ibm.websphere.security.TrustAssociationInterceptor interface.

1. Define the interceptor class method. WebSphere Application Server provides the interceptor Java interface, com.ibm.websphere.security.TrustAssociationInterceptor, which defines the following methods:
   - **`public boolean isTargetInterceptor(HttpServletRequest req)`** creates `WebTrustAssociationException;`.

     The isTargetInterceptor method determines whether the request originated with the proxy server associated with the interceptor. The implementation code must examine the incoming request object and determine if the proxy server forwarding the request is a valid proxy server for this interceptor. The result of this method determines whether the interceptor processes the request or not.
   - **`public void validateEstablishedTrust (HttpServletRequest req)`** creates `WebTrustAssociationException;`.

     The validateEstablishedTrust method determines if the proxy server from which the request originated is trusted or not. This method is called after the isTargetInterceptor method. The implementation code must authenticate the proxy server. The authentication mechanism is proxy-server specific. For example, in the product implementation for the WebSEAL server, this method retrieves the basic authentication information from the HTTP header and validates the information against the user registry used by WebSphere Application Server. If the credentials are invalid, the code creates the WebTrustAssociationException, indicating that the proxy server is not trusted and the request is to be denied.
   - **`public String getAuthenticatedUsername(HttpServletRequest req)`** creates `WebTrustAssociationException;`.

     The getAuthenticatedUsername method is called after trust is established between the proxy server and WebSphere Application Server. The product has accepted the proxy server authentication of the request and must now authorize the request. To authorize the request, the name of the original requestor must be subjected to an authorization policy to determine if the requestor has the

necessary privilege. The implementation code for this method must extract the user name from the HTTP request header and determine if that user is entitled to the requested resource. For example, in the product implementation for the WebSEAL server, the method looks for an `iv-user` attribute in the HTTP request header and extracts the user ID associated with it for authorization.

2. Configuring the interceptor. To make an interceptor configurable, the interceptor must extend com.ibm.websphere.security.WebSphereBaseTrustAssociationInterceptor. Implement the following methods:

   `public int init (java.util.Properties props);`

   The init(Properties) method accepts a `java.util.Properties` object, which contains the set of properties required to initialize the interceptor. All the properties set for an interceptor (by using the **Custom Properties** link for that interceptor or using scripting) is sent to this method. The interceptor then can use these properties to initialize itself. For example, in the product implementation for the WebSEAL server, this method reads the hosts and ports so that a request coming in can be verified to originate from trusted hosts and ports. A return value of **0** implies that the interceptor initialization is successful. Any other value implies that the initialization is not successful and the interceptor is ignored.

   Applicability of the following list

   If a previous implementation of the trust association interceptor returns a different error status you can either change your implementation to match the expectations or make one of the following changes:

   - Add the com.ibm.websphere.security.trustassociation.initStatus property in the trust association interceptor custom properties. Set the property to the value that indicates that the interceptor is successfully initialized. All of the other possible values imply failure. In case of failure, the corresponding trust association interceptor is not used.
   - Add the com.ibm.websphere.security.trustassociation.ignoreInitStatus property in the trust association interceptor custom properties. Set the value of this property to **true**, which tells WebSphere Application Server to ignore the status of this method. If you add this property to the custom properties, WebSphere Application Server does not check the return status, which is similar to previous versions of WebSphere Application Server.

   `public void cleanup ();`

   This method is called when the application server is stopped. It is used to prepare the interceptor for termination.

   `public void setVersion (String s);`

   This method is optional. The method is used to set the version and is for informational purpose only. The default value is `Unspecified`.

   You must configure the following methods implemented by the custom interceptor implementation. **This listing only shows the methods and does not include any implementation.**

```
*********************************************************************
import java.util.*;
import javax.servlet.http.HttpServletRequest;
import com.ibm.websphere.security.*;

public class myTAIImpl extends WebSphereBaseTrustAssociationInterceptor
      implements TrustAssociationInterceptor
{

      public myTAIImpl ()
      {
      }


      public boolean isTargetInterceptor (HttpServletRequest req)
           creates WebTrustAssociationException
      {

           //return true if this is the target interceptor, else return false.
      }
```

```
      public void validateEstablishedTrust (HttpServletRequest req)
           creates WebTrustAssociationFailedException
      {
           //validate if the request is from the trusted proxy server.
           //throw exception if the request is not from the trusted server.

      }

      public String getAuthenticatedUsername (HttpServletRequest req)
           creates WebTrustAssociationUserException
      {
           //Get the user name from the request and if the user is
           //entitled to the requested resource
           //return the user. Otherwise, throw the exception

      }


      public int init (Properties props)
      {
           //initialize the implementation. If successful return 0, else return -1.
      }

      public void cleanup ()
      {
           //Cleanup code.

      }

}
*********************************************************************
```

Note: If the init(Properties) method is implemented as described previously in your custom interceptor, this note does not apply to your implementation, and you can move on to the next step. Previous versions of com.ibm.websphere.security.WebSphereBaseTrustAssociationInterceptor include the public int init (String propsfile) method. This method is no longer required since the interceptor properties are not read from a file. The properties are now entered in the administrative console **Custom Properties** link of the interceptor using the administrative console or scripts. These properties then are made available to your implementation in the init(Properties) method. However, for backward compatibility, the init(String) method still is supported. The init(String) method is called by the default implementation of init(Properties) as shown in the following example.

```
// Default implementation of init(Properties props) method. A Custom
   // implementation should override this.
   public int init (java.util.Properties props)
   {
      String type =
       props.getProperty("com.ibm.websphere.security.trustassociation.types");
      String classfile=
       props.getProperty("com.ibm.websphere.security.trustassociation."
       +type+".config");
      if (classfile != null && classfile.length() > 0 ) {
         return init(classfile);
      } else {
         return -1;
      }
   }
```

Change your implementation to implement the init(Properties) method instead of relying on init(String propsfile) method. As shown in the previous example, this default implementation reads the properties

to load the property file. The com.ibm.websphere.security.trustassociation.types property gets the file containing the properties by concatenating `.config` to its value.

Note: The init(String) method still works if you want to use it instead of implementing the init(Properties) method. The only requirement is that the file name containing the custom trust association properties should now be entered using the **Custom Properties** link of the interceptor in the administrative console or by using scripts. You can enter the property using *either* of the following methods. The first method is used for backward compatibility with previous versions of WebSphere Application Server.

Method 1:

The same property names used in the previous release are used to obtain the file name. The file name is obtained by concatenating the `.config` to the com.ibm.websphere.security.trustassociation.types property value.

If the file name is called `myTAI.properties` and is located in the */properties* directory, set the following properties:

- com.ibm.websphere.security.trustassociation.types = myTAItype
- com.ibm.websphere.security.trustassociation.myTAItype.config = *app_server_root*/myTAI.properties

Method 2:

You can set the com.ibm.websphere.security.trustassociation.initPropsFile property in the trust association custom properties to the location of the file. For example, set the following property:

- `com.ibm.websphere.security.trustassociation.initPropsFile=` *app_server_root*`/myTAI.properties`

Type the previous code as one continuous line.

The location of the properties file is fully qualified (for example,*app_server_root*/`myTAI.properties`). Because the location can be different in a Network Deployment environment, use variables such as `${USER_INSTALL_ROOT}` to refer to the WebSphere Application Server installation directory. For example, if the file name is called `myTAI.properties`, and it is located in the `/properties` directory, then set the following properties:

- com.ibm.websphere.security.trustassociation.types = myTAItype
- com.ibm.websphere.security.trustassociation.myTAItype.config = *app_server_root*/myTAI.properties

3. Compile the implementation once you have implemented it. For example, *app_server_root*`/java/bin/javac -classpath` *install_root*`/plugins/com.ibm.ws.runtime.jar;<install_root>/lib/j2ee.jar myTAIImpl.java`
   a. Copy the class file to a location in the class path (preferably the *app_server_root*`/lib/ext` directory).
   b. Restart all the servers.
4. Delete the default WebSEAL interceptor in the administrative console and click **New** to add your custom interceptor. Verify that the class name is dot separated and appears in the class path.
5. Click the **Custom Properties** link to add additional properties that are required to initialize the custom interceptor. These properties are passed to the init(Properties) method of your implementation when it extends the com.ibm.websphere.security.WebSphereBaseTrustAssociationInterceptor as described in the previous step.
6. Save and synchronize (if applicable) the configuration.
7. Restart the servers for the custom interceptor to take effect.

Example

Refer to the Security: Resources for Learning article for a reference to an example of a custom interceptor.

Trust association interceptor support for Subject creation

The trust association interceptor (TAI) com.ibm.wsspi.security.tai.TrustAssociationInterceptor interface supports several features that are different from the existing com.ibm.websphere.security.TrustAssociationInterceptor interface.

The TAI interface supports a multiphase, negotiated authentication process. For example, some systems require a challenge response protocol back to the client. The two key methods in this interface are:

Key method name

public boolean isTargetInterceptor (HttpServletRequest req)

The isTargetInterceptor method determines whether the request originated with the proxy server that is associated with the interceptor. The implementation code must examine the incoming request object and determine if the proxy server that forwards the request is a valid proxy server for this interceptor. The result of this method determines whether the interceptor processes the request.

Method result

A `true` value tells WebSphere Application Server to have the TAI handle the request.

A `false` value, tells WebSphere Application Server to ignore the TAI.

Key method name

public TAIResult negotiateValidateandEstablishTrust (HttpServletRequest req, HttpServletResponse res)

The negotiateValidateandEstablishTrust method determines whether to trust the proxy server from which the request originated. The implementation code must authenticate the proxy server. The authentication mechanism is proxy-server specific. For example, in the product implementation for the WebSEAL server, this method retrieves the basic authentication information from the HTTP header and validates the information against the user registry that WebSphere Application Serve uses. If the credentials are not valid, the code creates the WebTrustAssociationException exception, which indicates that the proxy server is not trusted and the request is denied. If the credentials are valid, the code returns a TAIResult result, which indicates the status of the request processing with the client identity (Subject and principal name) to use for authorizing the Web resource.

Method result

Returns a TAIResult result, which indicates the status of the request processing. You can query the Request object and modify the Response object can be modified.

The TAIResult class has three static methods for creating a TAIResult result. The TAIResult create methods take an int type as the first parameter. WebSphere Application Server expects the result to be a valid HTTP request return code and is interpreted in one of the following ways:

- If the value is `HttpServletResponse.SC_OK`, this response tells WebSphere Application Server that the TAI completed its negotiation. The response also tells WebSphere Application Server to use the information in the TAIResult result to create a user identity.
- Other values tell WebSphere Application Server to return the TAI output, which is placed into the HttpServletResponse response, to the Web client. Typically, the Web client provides additional information and then places another call to the TAI.

The created TAIResults results have the following meanings:

TAIResult	Explanation
public static TAIResult create(int status);	Indicates a status to WebSphere Application Server. The status cannot be SC_OK because the identity information is provided.

TAIResult	Explanation
public static TAIResult create(int status, String principal);	Indicates a status to WebSphere Application Server and provides the user ID or the unique ID for this user. WebSphere Application Server creates credentials by querying the user registry.
public static TAIResult create(int status, String principal, Subject subject);	Indicates a status to WebSphere Application Server, the user ID or the unique ID for the user, and a custom Subject. If the Subject contains a hashtable, the principal is ignored. The contents of the Subject become part of the eventual user Subject.

All of the following examples are within the negotiateValidateandEstablishTrust method of a TAI.

The following code sample indicates that additional negotiation is required:

```
// Modify the HttpServletResponse object
 //  The response code is meaningful only on the client
 return TAIResult.create(HttpServletResponse.SC_CONTINUE);
```

The following code sample indicates that the TAI determined the user identity. WebSphere Application Server receives the user ID only and queries the user registry for additional information:

```
// modify the HttpServletResponse object
 return TAIResult.create(HttpServletResponse.SC_OK, userid);
```

The following code sample indicates that the TAI determined the user identity. WebSphere Application Server receives the complete user information that is contained in the hashtable. For more information on the hashtable, see Configuring inbound identity mapping. In this code sample, the hashtable is placed in the public credential portion of the Subject:

```
// create Subject and place Hashtable in it
 Subject subject = new Subject;
 subject.getPublicCredentials().add(hashtable);
 //the response code is meaningful only the client
 return TAIResult.create(HttpServletResponse.SC_OK, "ignored", subject);
```

The following code sample indicates that an authentication failure occured. WebSphere Application Server fails the authentication request:

```
//log error message
 // ....
 throw new WebTrustAssociationFailedException("TAI failed for this reason");
```

The following methods are additional methods on the TrustAssociationInterceptor interface. These methods are used for initialization, for shutdown, and for identifying the TAI to WebSphere Application Server. For more information, see the Java documentation.

Method name
public int initialize(Properties props)

Method result
This method is called during TAI initialization and is called only if custom properties are configured for the interceptor.

Method name
public String getVersion()

Method result
This method returns the version of the TAI.

Method name
public String getType()

Method result
This method returns the type of the TAI.

Method name
public void cleanup()

Method result
This method is called when stopping the WebSphere Application Server process. Stopping the WebSphere Application Server process provides an opportunity for the TAI to perform any necessary cleanup. This method is not necessary if cleanup is not required.

Enabling a plugpoint for custom password encryption

Two properties govern the protection of passwords. By configuring these two properties, you can enable a plugpoint for custom password encryption.

Before you begin

To view an example code sample that illustrates the com.ibm.wsspi.security.crypto.CustomPasswordEncryption interface, see "Plug point for custom password encryption" on page 1398.

About this task

The encryption method is called for password processing whenever the custom class is configured and custom encryption is enabled. The decryption method is called whenever the custom class is configured and the password contains the {custom:alias} tag. The custom:alias tag is stripped prior to decryption.

1. To enable custom password encryption, you must configure two properties:
   - **com.ibm.wsspi.security.crypto.customPasswordEncryptionClass** - Defines the custom class that implements the com.ibm.wsspi.security.crypto.CustomPasswordEncryption password encryption interface.
   - **com.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled** - Defines when the custom class is used for default password processing. When the passwordEncryptionEnabled option is not specified or set to false, and the passwordEncryptionClass class is specified, the decryption method is called whenever a {custom:alias} tag still exists in the configuration repository.
2. To configure custom password encryption, configure both of these properties in the security.xml file. The custom encryption class (com.acme.myPasswordEncryptionClass) must be placed in a Java archive (JAR) file in the ${WAS_INSTALL_ROOT}/classes directory in all WebSphere Application Server processes. Every configuration document that contains a password (security.xml and any application bindings that contain RunAs passwords), must be saved before all of the passwords become encrypted with the custom encryption class.
3. If the custom implementation class defaults to the com.ibm.wsspi.security.crypto.CustomPasswordEncryptionImpl interface, and this class is present in the class path, then encryption is enabled by default. This simplifies the enablement process for all nodes. It is not necessary to define any other properties except for those that the custom implementation requires. To disable encryption, but still use this class for decryption, specify the following class.
   - com.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled=false

What to do next

Whenever a custom encryption class encryption operation is called, and it creates a run-time exception or a defined PasswordEncryptException exception, the WebSphere Application Server runtime uses the {xor}

algorithm to encode the password. This encoding prevents the storage of the password in plain text. After the problem with the custom class has been resolved, it automatically encrypts the password the next time the configuration document is saved.

When a RunAs role is assigned a user ID and password, it currently is encoded using the WebSphere Application Server encoding function. Therefore, after the custom plug point is configured to encrypt the passwords, it encrypts the passwords for the RunAs bindings as well. If the deployed application is moved to a cell that does not have the same encryption keys, or the custom encryption is not yet enabled, a login failure results because the password is not readable.

One of the responsibilities of the custom password encryption implementation is to manage the encryption keys. This class must decrypt any password that it encrypted. Any failure to decrypt a password renders that password to be unusable, and the password must be changed in the configuration. All encryption keys must be available for decryption there and no passwords are left using those keys. The master secret must be maintained by the custom password encryption class to protect the encryption keys.

You can manage the master secret by using a stash file for the keystore, or by using a password locator that enables the custom encryption class to locate the password so that it can be locked down.

Plug point for custom password encryption

A plug point for custom password encryption can be created to encrypt and decrypt all passwords in WebSphere Application Server that are currently encoded or decoded using Base64-encoding.

The implementation class of this plug point has the responsibility for managing keys, determining the encryption algorithm to use, and for protecting the master secret. The WebSphere Application Server runtime stores the encrypted passwords in their existing locations, preceded with {custom:alias} tags instead of {xor} tags. The custom part of the tag indicates that it is a custom algorithm. The alias part of the tag is specified by the custom implementation, which helps to indicate how the password is encrypted. The implementation can include the key alias, encryption algorithm, encryption mode, or encryption padding.

A custom provider of this plug point must implement an interface that is designed to encrypt and decrypt passwords. The interface is called by the WebSphere Application Server runtime whenever the custom plug point is enabled. The custom algorithm becomes one of the supported algorithms when the plug point is enabled. Other supported algorithms include {xor} (standard base64 encoding) and {os400} which is used on the iSeries platform.

The following example illustrates the com.ibm.wsspi.security.crypto.CustomPasswordEncryption interface:

```
package com.ibm.wsspi.security.crypto;
public interface CustomPasswordEncryption
{

    /**
     * The encrypt operation takes a UTF-8 encoded String in the form of a byte[].
     * The byte[] is generated from String.getBytes("UTF-8").
     * An encrypted byte[] is returned from the implementation in the EncryptedInfo
     * object.  Additionally, a logical key alias is returned in the EncryptedInfo
     * objectwhich is passed back into the decrypt method to determine which key was
     * used to encrypt this password. The WebSphere Application Server runtime has
     * no knowledge of the algorithm or the key used to encrypt the data.
     *
     * @param byte[]
     * @return com.ibm.wsspi.security.crypto.EncryptedInfo
     * @throws com.ibm.wsspi.security.crypto.PasswordEncryptException
     **/
    public EncryptedInfo encrypt (byte[] decrypted_bytes) throws PasswordEncryptException;

    /**
     * The decrypt operation takes the EncryptedInfo object containing a byte[]
```

```
     * and the logical key alias and converts it to the decrypted byte[].  The
     * WebSphere Application Server runtime converts the byte[] to a String
     * using new String (byte[], "UTF-8");
     *
     * @param com.ibm.wsspi.security.crypto.EncryptedInfo
     * @return byte[]
     * @throws com.ibm.wsspi.security.crypto.PasswordDecryptException
     **/
    public byte[] decrypt (EncryptedInfo info) throws PasswordDecryptException;

    /**
     * The following is reserved for future use and is currently not
     * called by the WebSphere Application Server runtime.
     *
     * @param java.util.HashMap
     **/
    public void initialize (java.util.HashMap initialization_data);
}
```

The com.ibm.wsspi.security.crypto.EncryptedInfo class contains the encrypted bytes with the user-defined alias that is associated with the encrypted bytes. This information is passed back into the encryption method to help determine how the password was originally encrypted.

```
package com.ibm.wsspi.security.crypto;
public class EncryptedInfo
{
    private byte[] bytes;
    private String alias;

/**
 * This constructor takes the encrypted bytes and a keyAlias as parameters.
 * This constructor is used to pass to or from the WebSphere Application Server
 * runtime to enable the runtime to associate the bytes with a specific key that
 * is used to encrypt the bytes.
 */

    public EncryptedInfo (byte[] encryptedBytes, String keyAlias)
    {
        bytes = encryptedBytes;
        alias = keyAlias;
    }

/**
 * This command returns the encrypted bytes.
 *
 * @return byte[]
 */
    public byte[] getEncryptedBytes()
    {
        return bytes;
    }

/**
 * This command returns the key alias.  The key alias is a logical string that is
 * associated with the encrypted password in the model. The format is
 * {custom:keyAlias}encrypted_password. Typically, just the key alias is placed
 * here, but algorithm information can also be returned.
 *
 * @return String
 */
    public String getKeyAlias()
    {
        return alias;
    }

}
```

The encryption method is called for password processing whenever the custom class is configured and custom encryption is enabled. The decryption method is called whenever the custom class is configured and the password contains the {custom:alias} tag . The custom:alias tag is stripped prior to decryption. For more information, see Enabling custom password encryption.

Chapter 12. Naming and directory

Using naming

Naming is used by clients of WebSphere Application Server applications most commonly to obtain references to objects related to those applications, such as Enterprise JavaBeans (EJB) homes.

About this task

The Naming service is based on the Java Naming and Directory Interface (JNDI) Specification and the Object Management Group (OMG) Interoperable Naming (CosNaming) specifications Naming Service Specification, Interoperable Naming Service revised chapters and Common Object Request Broker: Architecture and Specification (CORBA).

1. Develop your application using either JNDI or CORBA CosNaming interfaces. Use these interfaces to look up server application objects that are bound into the namespace and obtain references to them. Most Java developers use the JNDI interface. However, the CORBA CosNaming interface is also available for performing Naming operations on WebSphere Application Server name servers or other CosNaming name servers.
2. Assemble your application using an assembly tool. Application assembly is a packaging and configuration step that is a prerequisite to application deployment. If the application you are assembling is a client to an application running in another process, you should qualify the jndiName values in the deployment descriptors for the objects related to the other application. Otherwise, you might need to override the names with qualified names during application deployment. If the objects have fixed qualified names configured for them, you should use them so that the jndiName values do not depend on the other application's location within the topology of the cell.
3. Optional: Verify that your application is assigned the appropriate security role if administrative security is enabled. For more information on the security roles, see "Naming roles" on page 1411.
4. Deploy your application. Put your assembled application onto the application server. If the application you are assembling is a client to an application running in another server process, be sure to qualify the jndiName values for the other application's server objects if they are not already qualified. For more information on qualified names, refer to "Lookup names support in deployment descriptors and thin clients" on page 1405.
5. Optional: If your application must access applications in other cells, configure foreign cell bindings for the other cells.
6. Configure namespace bindings. This step is necessary in these cases:
   - Your deployed application is to be accessed by legacy client applications running on previous versions of the product. In this case, you must configure additional name bindings for application objects relative to the default initial context for legacy clients. (Version 5 clients have a different initial context from legacy clients.)
   - The application requires qualified name bindings for such reasons as:
     – It will be accessed by Java Platform, Enterprise Edition (Java EE) client applications or server applications running in another server process.
     – It will be accessed by thin client applications.

     In this case, you can configure name bindings as additional bindings for application objects. The qualified names for the configured bindings are *fixed*, meaning they do not contain elements of the cell topology that can change if the application is moved to another server. Objects as bound into the namespace by the system can always be qualified with a topology-based name. You must explicitly configure a name binding to use as a fixed qualified name.

   For more information on qualified names, refer to "Lookup names support in deployment descriptors and thin clients" on page 1405. For more information on configured name bindings, refer to "Configured name bindings" on page 1408.

7. Troubleshoot any problems that develop. If a Naming operation is failing and you need to verify whether certain name bindings exist, use the dumpNameSpace tool to generate a dump of the namespace.

Naming

Naming is used by clients of WebSphere Application Server applications to obtain references to objects related to those applications, such as enterprise bean (EJB) homes.

These objects are bound into a mostly hierarchical structure, referred to as a *namespace*. In this structure, all non-leaf objects are called *contexts*. Leaf objects can be contexts and other types of objects. Naming operations, such as lookups and binds, are performed on contexts. All naming operations begin with obtaining an *initial context*. You can view the initial context as a starting point in the namespace.

The namespace structure consists of a set of *name bindings*, each consisting of a name relative to a specific context and the object bound with that name. For example, the name `myApp/myEJB` consists of one non-leaf binding with the name `myApp`, which is a context. The name also includes one leaf binding with the name `myEJB`, relative to `myApp`. The object bound with the name `myEJB` in this example happens to be an EJB home reference. The whole name `myApp/myEJB` is relative to the initial context, which you can view as a starting place when performing naming operations.

You can access and manipulate the namespace through a *name server*. Users of a name server are referred to as *naming clients*. Naming clients typically use the Java Naming and Directory Interface (JNDI) to perform naming operations. Naming clients can also use the Common Object Request Broker Architecture (CORBA) CosNaming interface.

You can use security to control access to the namespace. For more information, see *Naming roles*.

Typically, objects bound to the namespace are resources and objects associated with installed applications. These objects are bound by the system, and client applications perform lookup operations to obtain references to them. Occasionally, server and client applications bind objects to the namespace. An application can bind objects to transient or persistent partitions, depending on requirements.

In Java Platform, Enterprise Edition (Java EE) or Java Platform, Standard Edition (Java SE) environments, some JNDI operations are performed with `java:` URL names. Names bound under these names are bound to a completely different namespace which is local to the calling process. However, some lookups on the `java:` namespace may trigger indirect lookups to the name server.

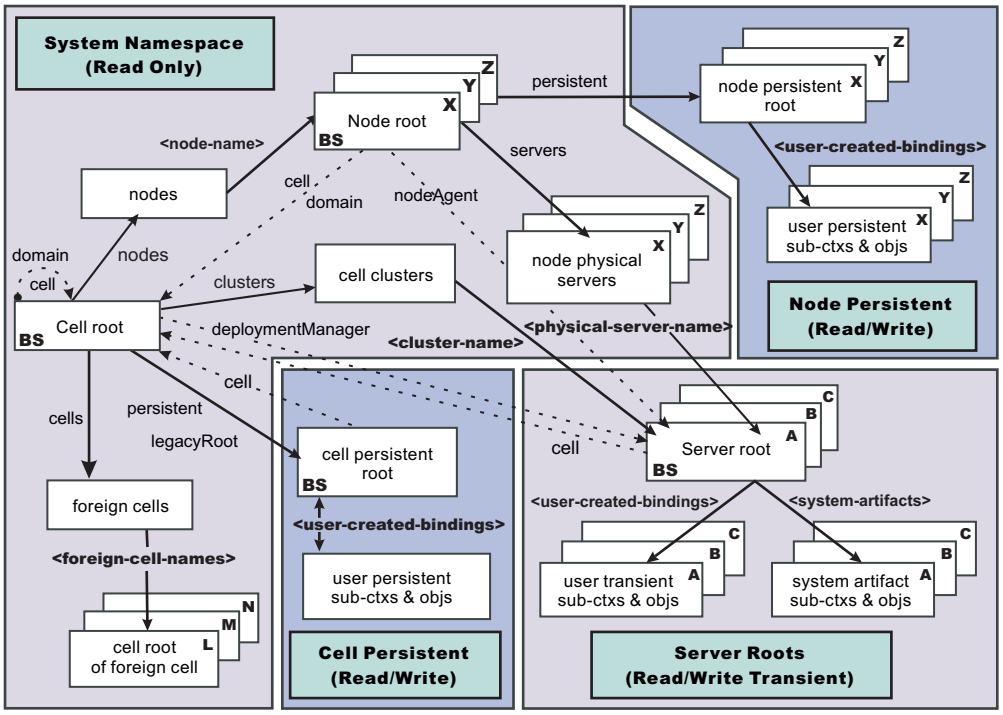
Namespace logical view

The namespace for the entire cell is federated among all servers in the cell. Every server process contains a name server. All name servers provide the same logical view of the cell namespace.

The various server roots and persistent partitions of the namespace are interconnected by a system namespace. You can use the system namespace structure to traverse to any context in a cell's namespace.

A logical view of the namespace in a multiple-server installation is shown in the following diagram.

Logical View of a Cell's Namespace



The bindings in the preceding diagram appear with solid arrows, labeled in bold, and dashed arrows, labeled in gray. Solid arrows represent *primary bindings*. A primary binding is formed when the associated subcontext is created. Dashed arrows show *linked bindings*. A linked binding is formed when an existing context is bound under an additional name. Linked bindings are added for convenience or interoperability with previous WebSphere Application Server versions.

A cell namespace is composed of contexts which reside in servers throughout the cell. All name servers in the cell provide the same logical view of the cell namespace. A name server constructs this view at startup by reading configuration information. Each name server has its own local in-memory copy of the namespace and does not require another running server to function. There are, however, a few exceptions. Server roots for other servers are not replicated among all the servers. The respective server for a server root must be running to access that server root context.

In WebSphere Application Server Network Deployment cells, the cell and node persistent areas can be read even if the deployment manager and respective node agent are not running. However, the deployment manager must be running to update the cell persistent segment, and a node agent must be running to update its respective node persistent segment.

Namespace partitions

There are four major partitions in a cell namespace:
- System namespace partition
- Server roots partition
- Cell persistent partition
- Node persistent partition

System namespace partition

The system namespace contains a structure of contexts based on the cell topology. The system structure supports traversal to all parts of a cell namespace and to the cell root of other cells, which are configured as foreign cells. The root of this structure is the cell root. In addition to the

cell root, the system structure contains a node root for each node in the cell. You can access other contexts of interest specific to a node from the node root, such as the node persistent root and server roots for servers configured in that node.

All contexts in the system namespace are read-only. You cannot add, update, or remove any bindings.

Server roots partition

Each server in a cell has a server root context. A server root is specific to a particular server. You can view the server roots for all servers in a cell as being in a transient read/write partition of the cell namespace. System artifacts, such as enterprise bean (EJB) homes for server applications and resources, are bound under the server root context of the associated server. A server application can also add bindings under its server root. These bindings are transient. Therefore, the server application creates all required bindings at application startup, so they exist anytime the application is running.

A server cluster is composed of many servers that are logically equivalent. Each member of the cluster has its own server root. These server roots are not replicated across the cluster. In other words, adding a binding to the server root of one member does not propagate it to the server roots of the other cluster members. To maintain the same view across the cluster, you should create all user bindings under the server root by the server application at application startup so that the bindings are present under the server root of each cluster member. Because of Workload Management (WLM) behavior, a JNDI client outside a cluster has no control over which cluster member's server root context becomes the target of the JNDI operation. Therefore, you should execute bind operations to the server root of a cluster member from within that cluster member process only.

Server-scoped configured name bindings are relative to a server's server root.

The name of a cluster member must be unique within a cell and must be different from the cell name.

Cell persistent partition

The root context of the cell persistent partition is the cell persistent root. A binding created under the cell persistent root is saved as part of the cell configuration and continues to exist until it is explicitly removed. Applications that need to create additional persistent bindings of objects generally associated with the cell can bind these objects under the cell persistent root.

It is important to note that the cell persistent area is not designed for transient, rapidly changing bindings. The bindings are more static in nature, such as part of an application setup or configuration, and are not created at run time.

The cell persistent area can be read even if the deployment manager is not running. However, the deployment manager must be running to update the cell persistent segment. Because every server contains its own copy of the cell persistent partition, any server can look up locally objects bound in the cell persistent partition.

Cell-scoped configured name bindings are relative to a cell's cell persistent root.

Node persistent partition

The node persistent partition is similar to the cell partition except that each node has its own node persistent root. A binding created under a node persistent root is saved as part of that node configuration and continues to exist until it is explicitly removed.

Applications that need to create additional persistent bindings of objects associated with a specific node can bind those objects under that particular node's node persistent root. As with the cell persistent area, it is important to note that the node persistent area is not designed for transient, rapidly changing bindings. These bindings are more static in nature, such as part of an application setup or configuration, and are not created at run time.

The node persistent area for a node can be read from any server in the node even if the respective node agent is not running. However, the node agent must be running to update the

node persistent area, or for any server outside the node to read from that node persistent partition. Because every server in a node contains its own copy of the node persistent partition for its node, any server in the node can look up locally objects bound in that node persistent partition.

Node-scoped configured name bindings are relative to a node's node persistent root.

Initial context support

All naming operations begin with obtaining an initial context. You can view the initial context as a starting point in the namespace. Use the initial context to perform naming operations, such as looking up and binding objects in the namespace.

Initial contexts registered with the ORB as initial references

The server root, cell persistent root, cell root, and node root are registered with the name server's ORB and can be used as an initial context. An initial context is used by CORBA and enterprise bean applications as a starting point for namespace lookups. The keys for these roots as recognized by the ORB are shown in the following table:

Root Context	Initial Reference Key
Server Root	NameServiceServerRoot
Cell Persistent Root	NameServiceCellPersistentRoot
Cell Root	NameServiceCellRoot, NameService
Node Root	NameServiceNodeRoot

A server root initial context is the server root context for the specific server you are accessing. Similarly, a node root initial context is the node root for the server being accessed.

You can use the previously mentioned keys in CORBA INS object URLs (corbaloc and corbaname) and as an argument to an ORB `resolve_initial_references` call. For examples, see CORBA and JNDI programming examples, which show how to get an initial context.

Default initial contexts

The default initial context depends on the type of client. Different categories of clients and the corresponding default initial context follow.

- **WebSphere Application Server V5 and later JNDI interface implementation**

  The JNDI interface is used by EJB applications to perform namespace lookups. WebSphere Application Server clients by default use the WebSphere Application Server CosNaming JNDI plug-in implementation. The default initial context for clients of this type is the server root of the server specified by the provider URL. For more details, refer to the JNDI programming examples on getting initial contexts.
- **Other JNDI implementation**

  Some applications can perform namespace lookups with a non-product CosNaming JNDI plug-in implementation. Assuming the key **`NameService`** is used to obtain the initial context, the default initial context for clients of this type is the cell root.
- **CORBA**

  The standard CORBA client obtains an initial org.omg.CosNaming.NamingContext reference with the key **`NameService`**. The initial context in this case is the cell root.

Lookup names support in deployment descriptors and thin clients

Server application objects, such as enterprise bean (EJB) homes, are bound relative to the server root context for the server in which the application is installed. Other objects, such as resources, can also be

bound to a specific server root. The names used to look up these objects must be qualified so as to select the correct server root. This topic discusses what relative and qualified names are, when they can be used, and how you can construct them.

Relative names

All names are relative to a context. Therefore, a name that can be resolved from one context in the namespace cannot necessarily be resolved from another context in the namespace. This point is significant because the system binds objects with names relative to the server root context of the server in which the application is installed. Each server has its own server root context. The initial Java Naming and Directory Interface (JNDI) context is by default the server root context for the server identified by the provider URL used to obtain the initial context. (Typically, the URL consists of a host and port.) For applications running in a server process, the default initial JNDI context is the server root for that server. A relative name will resolve successfully when the initial context is obtained from the server which contains the target object, but it will not resolve successfully from an initial context obtained from another server.

If all clients of a server application run in the same server process as the application, all objects associated with that application are bound to the same initial context as the clients' initial context. In this case, only names relative to the server's server root context are required to access these server objects. Frequently, however, a server application has clients that run outside the application's server process. The initial context for these clients can be different from the server application's initial context, and lookups on the relative names for server objects may fail. These clients need to use the qualified name for the server objects. This point must be considered when setting up the jndiName values in a Java Platform, Enterprise Edition (Java EE) client application deployment descriptors and when constructing lookup names in thin clients. Qualified names resolve successfully from any initial context in the cell.

Qualified names

All names are relative to a context. Here, the term *qualified name* refers to names that can be resolved from any initial context in a cell. This action is accomplished by using names that navigate to the same context, the cell root. The rest of the qualified name is then relative to the cell root and uniquely identifies an object throughout the cell. All initial contexts in a server (that is, all naming contexts in a server registered with the ORB as an initial reference) contain a binding with the name **cell**, which links back to the cell root context. All qualified names begin with the string **cell/** to navigate from the current initial context back to the cell root context.

A qualified name for an object is the same throughout the cell. The name can be topology-based, or some fixed name bound under the cell persistent root. Topology-based names, described in more detail below, navigate through the system namespace to reach the target object. A fixed name bound under the cell persistent root has the same qualified name throughout the cell and is independent of the topology. Creating a fixed name under the cell persistent root for a server application object requires an extra step when the server application is installed, but this step eliminates impacts to clients when the application is moved to a different location in the cell topology. The process for creating a fixed name is described later in this section.

Generally, you **must** use qualified names for EJB jndiName values in a Java EE client application deployment descriptors and for EJB lookup names in thin clients. The only exception is when the initial context is obtained from the server in which the target object resides. For example, a session bean which is a client to an entity bean can use a relative name if the two beans run in the same server. If the session bean and entity beans run in different servers, the jndiName for the entity bean must be qualified in the session bean's deployment descriptors. The same requirement may be true for resources as well, depending on the scope of the resource.

- **Topology-based names**

The system namespace partition in a cell's namespace reflects the cell's topology. This structure can be navigated to reach any object bound into the cell's namespace. Topology-based qualified names include elements from the topology which reflect the object's location within the cell.

For a system-bound object, such as an EJB home, the form for a topology-based qualified name depends on whether the object is bound to a single server or cluster. Both forms are described below.

Single server

An object bound in a single server has a topology-based qualified name of the following form:

`cell/nodes/`*`nodeName`*`/servers/`*`serverName`*`/`*`relativeJndiName`*

where *nodeName* and *serverName* are the node name and server name for the server where the object is bound, and *relativeJndiName* is the unqualified name of the object; that is, the object's name relative to its server's server root context.

Server cluster

An object bound in a server cluster has a topology-based qualified name of the following form:

`cell/clusters/`*`clusterName`*`/`*`relativeJndiName`*

where *clusterName* is the name of the server cluster where the object is bound, and *relativeJndiName* is the unqualified name of the object; that is, the object's name relative to a cluster member's server root context.

- **Fixed names**

  It is possible to create a fixed name for a server object so that the qualified name is independent of the cell topology. This quality is desirable when clients of the application run in other server processes or as pure clients. Fixed names have the advantage of not changing if the object is moved to another server. The jndiName values in deployment descriptors for a Java EE client application can reference the qualified fixed name for a server object regardless of the cell topology on which the client or server application is being installed.

  Defining a cell-wide fixed name for a server application object requires an extra step after the server application is installed. That is, a binding for the object must be created under the cell persistent root. A fixed name bound under the cell persistent root can be any name, but all names under the cell persistent root must be unique within the cell because the cell persistent root is global to the entire cell.

  A qualified fixed name has the form:

  `cell/persistent/`*`fixedName`*

  where *fixedName* is an arbitrary fixed name.

  The binding can be created programmatically (for example, using JNDI). However, it is probably more convenient to configure a cell-scoped binding for the server object.

  You must keep the programmatic or configured binding up-to-date. Configured EJB bindings are based on the location of the enterprise bean within the cell topology, and moving the EJB application to another server, for example, requires the configured binding to be updated. Similar changes affect an EJB home reference programmatically bound so that the fixed name would need to be rebound with a current reference. However, for Java EE clients, the jndiName value for the object, and for thin clients, the lookup name for the object, remains the same. In other words, clients that access objects by fixed names are not affected by changes to the configuration of server applications they access.

Using lookup names in deployment descriptor bindings

Java EE applications can contain deployment descriptors, such as ejb-ref, resource-ref, and resource-env-ref, that are used to declare various types of references. These reference declarations define java:comp/env lookup names that are available to corresponding Java EE components. Each java:comp/env lookup name must be mapped to a lookup name in the global name space, relative to the server root context, which is the default initial JNDI context.

If a reference maps to an object that is bound under the server root context for the same server as the component that is executing the lookup, you can use a relative lookup name. If a reference maps to an

object that is bound under the server root context of another server, you must qualify the lookup name. For example, you must qualify a lookup name if a servlet, that is running on one server, declares an ejb-ref for an EJB that is running on another server. Similarly, if the reference maps to an object that is bound into a persistent partition of the name space, or to an object that is bound through a cell-scoped or node-scoped configured name space binding, you must use a qualified name.

You can specify deployment descriptor reference binding values when you install the application, and edit them after the application is installed. If you need to change the JNDI lookup name a reference maps to, in the administrative console, click **Applications > Enterprise Applications >** *application_name*. In the References section, there are links that correspond to the various reference types, such as EJB references and Resource environment entry references, that are declared by this application. Click on the link for the reference type that you need to change, and then specify a new value in the **Target Resource JNDI Name** field.

JNDI support in WebSphere Application Server

The product includes a name server to provide shared access to Java components, and an implementation of the `javax.naming` JNDI package which supports user access to the name server through the Java Naming and Directory Interface (JNDI) naming interface.

WebSphere Application Server does ***not*** provide implementations for:
- `javax.naming.directory` or
- `javax.naming.ldap` packages

Also, WebSphere Application Server does ***not*** support interfaces defined in the `javax.naming.event` package.

However, to provide access to LDAP servers, the development kit shipped with WebSphere Application Server supports the Sun Microsystems implementation of:
- `javax.naming.ldap` and
- `com.sun.jndi.ldap.LdapCtxFactory`

The WebSphere Application Server JNDI implementation is based on the JNDI interface, and was tested with the Sun Microsystems JNDI Service Provider Interface (SPI).

The default behavior of this JNDI implementation is adequate for most users. However, users with specific requirements can control certain aspects of JNDI behavior.

Configured name bindings

Administrators can configure bindings into the namespace. A configured binding is different from a programmatic binding in that the system creates the binding every time a server is started, even if the target context is in a transient partition.

Administrators can add name bindings to the namespace through the configuration. Name servers add these configured bindings to the namespace view, by reading the configuration data for the bindings. Configuring bindings is an alternative to creating the bindings from a program. Configured bindings have the advantage of being created each time a server starts, even when the binding is created in a transient partition of the namespace. Cell-scoped configured bindings provide a fixed qualified name for server application objects.

Scope

You can configure a binding at one of the following four scopes: cell, node, server, or cluster. Cell-scoped bindings are created under the cell persistent root context. Node-scoped bindings are created under the

node persistent root context for the specified node. Server-scoped bindings are created under the server root context for the selected server. Cluster-scoped bindings are created under the server root context in each member of the selected cluster.

The scope you select for new bindings depends on how the binding is to be used. For example, if the binding is not specific to any particular node, cluster, or server, or if you do not want the binding to be associated with any specific node, cluster, or server, a cell-scoped binding is a suitable scope. Defining fixed names for enterprise beans to create fixed qualified names is just such an application. If a binding is to be used only by clients of an application running on a particular server (or cluster), or if you want to configure a binding with the same name on different servers (or clusters) which resolve to different objects, a server-scoped (or cluster-scoped) binding would be appropriate. Note that two servers or clusters can have configured bindings with the same name but resolve to different objects. At the cell scope, only one binding with a given name can exist.

Intermediate contexts

Intermediate contexts created with configured bindings are read-only. For example, if an EJB home binding is configured with the name `some/compound/name/ejbHome`, the intermediate contexts `some`, `some/compound`, and `some/compound/name` will be created as read-only contexts. You cannot add, update, or remove any read-only bindings.

The configured binding name cannot conflict with existing bindings. However, configured bindings can use the same intermediate context names. Therefore, a configured binding with the name `some/compound/name2/ejbHome2` does not conflict with the previous example name.

Configured binding types

Types of objects that you can bind follow:

EJB: EJB home installed in some server in the cell

The following data is required to configure an EJB home binding:

- JNDI name of the EJB server or server cluster where the enterprise bean is deployed
- Target root for the configured binding (scope)
- The name of the configured binding, relative to the target root.

A cell-scoped EJB binding is useful for creating a fixed lookup name for an enterprise bean so that the qualified name is not dependent on the topology.

Note: In standalone servers, an EJB binding resolving to another server cannot be configured because the name server does not read configuration data for other servers. That data is required to construct the binding.

CORBA: CORBA object available from some CosNaming name server

You can identify any CORBA object bound into some INS compliant CosNaming server with a corbaname URL. The referenced object does not have to be available until the binding is actually referenced by some application.

The following data is required in order to configure a CORBA object binding:

- The corbaname URL of the CORBA object
- An indicator if the bound object is a context or leaf node object (to set the correct CORBA binding type of context or object)
- Target root for the configured binding
- The name of the configured binding, relative to the target root

Indirect: Any object bound in WebSphere Application Server namespace accessible with JNDI

Besides CORBA objects, this includes javax.naming.Referenceable, javax.naming.Reference, and java.io.Serializable objects. The target object itself is not bound to the namespace. Only the information required to look up the object is bound. Therefore, the referenced name server does

not have to be running until the binding is actually referenced by some application. The following data is required in order to configure an indirect JNDI lookup binding:
- JNDI provider URL of name server where object resides
- JNDI lookup name of object
- Target root for the configured binding (scope)
- The name of the configured binding, relative to the target root.

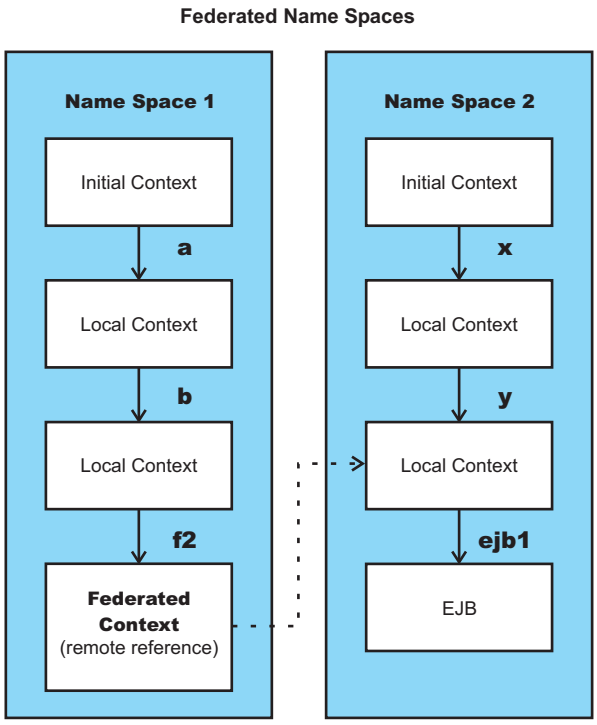
A cell-scoped indirect binding is useful when creating a fixed lookup name for a resource so that the qualified name is not dependent on the topology. You can also achieve this topology by widening the scope of the resource definition.

String: String constant
You can configure a binding of a string constant. The following data is required to configure a string constant binding:
- String constant value
- Target root for the configured binding (scope)
- The name of the configured binding, relative to the target root

Namespace federation

Federating namespaces involves binding contexts from one namespace into another namespace.

For example, assume that a namespace, Namespace 1, contains a context under the name `a/b`. Also assume that a second namespace, Namespace 2, contains a context under the name `x/y`. (See the following illustration.) If context `x/y` in Namespace 2 is bound into context `a/b` in Namespace 1 under the name `f2`, the two namespaces are federated. Binding `f2` is a federated binding because the context associated with that binding comes from another namespace. From Namespace 1, a lookup of the name `a/b/f2` returns the context bound under the name `x/y` in Namespace 2. Furthermore, if context `x/y` contains an enterprise bean (EJB) home bound under the name `ejb1`, the EJB home can be looked up from Namespace 1 with the lookup name `a/b/f2/ejb1`. Notice that the name crosses namespaces. This fact is transparent to the naming client.



In a product namespace, you can create federated bindings with the following restrictions:

- Federation is limited to CosNaming name servers. A product name server is a Common Object Request Broker Architecture (CORBA) CosNaming implementation. You can create federated bindings to other CosNaming contexts. You cannot, for example, bind contexts from an LDAP name server implementation.
- If you use JNDI to federate the namespace, you must use a WebSphere Application Server initial context factory to obtain the reference to the federated context. If you use some other initial context factory implementation, you might not be able to create the binding or the level of transparency might be reduced.
- A federated binding to a non-product naming context has the following functional limitations:
  - JNDI operations are restricted to the use of CORBA objects. For example, you can look up EJB homes, but you cannot look up non-CORBA objects such as data sources.
  - JNDI caching is not supported for non-product namespaces. This restriction affects the performance of lookup operations only.
  - If security is enabled, the producr does not support federated bindings to non-product namespaces.
- Do not federate two product standalone server namespaces. Incorrect behavior might result. If you want to federate product namespaces, use servers running under the Network Deployment package of WebSphere Application Server.
- When federating the namespaces of two cells running a Network Deployment package of WebSphere Application Server, the names of the cells must be different. Otherwise, incorrect behavior can result.

Naming roles

The Java 2 Platform, Enterprise Edition (J2EE) role-based authorization concept is extended to protect the CosNaming service.

CosNaming security offers increased granularity of security control over CosNaming functions. CosNaming functions are available on CosNaming servers such as the WebSphere Application Server. They affect the content of the name space. Generally two ways are acceptable in which client programs result in CosNaming calls. The first is through the Java Naming and Directory Interface (JNDI) methods. The second is CORBA clients invoking CosNaming methods directly.

The following security roles exist. However, the roles have an authority level from low to high as shown in the following list. The list also provides the security-related interface methods for each role. The interface methods that are not listed are either not supported or not relevant to security.

- **CosNamingRead**. Users who are assigned the CosNamingRead role can do queries of the name space, such as through the JNDI lookup method. The Everyone special-subject is the default policy for this role.

Table 34. CosNamingRead role packages and interface methods

Package	Interface methods
javax.naming	• Context.list • Context.listBindings • Context.lookup • NamingEnumeration.hasMore • NamingEnumeration.next
org.omg.CosNaming	• NamingContext.list • NamingContext.resolve • BindingIterator.next_one • BindingIterator.next_n • BindingIterator.destroy

- **CosNamingWrite**. Users who are assigned the CosNamingWrite role can do write operations (such as JNDI bind, rebind, or unbind) plus CosNamingRead operations. As a default policy, Subjects are not assigned this role.

Table 35. CosNamingWrite role packages and interface methods

Package	Interface methods
javax.naming	• Context.bind • Context.rebind • Context.rename • Context.unbind
org.omg.CosNaming	• NamingContext.bind • NamingContext.bind_context • NamingContext.rebind • NamingContext.rebind_context • NamingContext.unbind

- **CosNamingCreate**. Users who are assigned the CosNamingCreate role are allowed to create new objects in the name space through JNDI createSubcontext operations plus CosNamingWrite operations. As a default policy, Subjects are not assigned this role.

Table 36. CosNamingCreate role packages, interface methods

Package	Interface methods
javax.naming	Context.createSubcontext
org.omg.CosNaming	NamingContext.bind_new_context

- **CosNamingDelete**. Users who are assigned the CosNamingDelete role can destroy objects in the name space, for example byusing the JNDI destroySubcontext method and CosNamingCreate operations. As a default policy, Subjects are not assigned this role.

Table 37. CosNamingDelete role packages and interface methods

Package	Interface methods
javax.naming	Context.destroySubcontext
org.omg.CosNaming	NamingContext.destroy

Note: The `javax.naming` package applies to the CosNaming JNDI service provider only. All of the variants of a JNDI interface method have the same role mapping.

If the caller is not authorized, the packages listed in the previous tables exhibit the following behavior:

javax.naming
This package creates the javax.naming.NoPermissionException exception, which maps NO_PERMISSION from the CosNaming method invocation to NoPermissionException.

org.omg.CosNaming
This package creates the org.omg.CORBA.NO_PERMISSION exception.

Users, groups, or the AllAuthenticated and Everyone special subjects can be added or removed to or from the naming roles from the WebSphere Application Server administrative console at any time. However, you must restart the server for the changes to take effect. A best practice is to map groups or one of the special-subjects, rather than specific users, to Naming roles because it is more flexible and easier to administer in the long run. By mapping a group to a naming role, adding or removing users to or from the group occurs outside of WebSphere Application Server and does not require a server restart for the change to take effect.

If a user is assigned a particular naming role and that user is a member of a group that is assigned a different naming role, the user is granted the most permissive access between the role that is assigned and the role the group is assigned. For example, assume that the MyUser user is assigned the CosNamingRead role. Also, assume that the MyGroup group is assigned the CosNamingCreate role. If the

MyUser user is a member of the MyGroup group, the MyUser user is assigned the CosNamingCreate role because the user is a member of the MyGroup group. If the MyUser user is not a member of the MyGroup group, is assigned the CosNamingRead role.

The CosNaming authorization policy is only enforced when administrative security is enabled. When administrative security is enabled, attempts to do CosNaming operations without the proper role assignment result in a org.omg.CORBA.NO_PERMISSION exception from the CosNaming server.

In WebSphere Application Server, each CosNaming function is assigned to one role only. Therefore, users who are assigned the CosNamingCreate role cannot query the name space unless they also are assigned the CosNamingRead role. In most cases, a creator needs three roles assigned: CosNamingRead, CosNamingWrite, and CosNamingCreate. The CosNamingRead and CosNamingWrite roles assignment for the creator example in above have been included in CosNamingCreate role. In most cases, WebSphere Application Server administrators do not have to change the roles assignment for every user or group when they move to this release from a previous one.

Although the ability exists to greatly restrict access to the name space by changing the default policy, doing so might result in unexpected org.omg.CORBA.NO_PERMISSION exceptions at runtime. Typically, J2EE applications access the name space and the identity is that of the user that authenticated to WebSphere Application Server when he J2EE application is accessed. Unless the J2EE application provider clearly communicates the expected naming roles, fully consider changing the default naming authorization policy.

Foreign cell bindings

If you have applications in a cell that access other applications in another cell, you can configure a foreign cell name binding for the other cell. A *foreign cell name binding* is a context binding that resolves to the Cell Root context of the other cell. All applications in the local cell can look up objects in the foreign cell through the foreign cell binding.

Foreign cell bindings limit bootstrap address information for a foreign cell to a single location, instead of placing in the local cell's application deployment data the bootstrap address information contained in every foreign cell reference. If the bootstrap address for the foreign cell changes, you only need to update the foreign cell binding. You do not need to update the deployment data for any application in the local cell that looks up application objects in the foreign cell through the foreign cell binding.

For example, assume the foreign cell CellB has a cell-scoped EJB namespace binding configured with a name in the namespace of `ejb/AccountHome`. Applications running in CellB would look up the home with a JNDI name of `cell/persistent/ejb/AccountHome`. (J2EE applications would actually use a `java:comp/env` name that maps to that JNDI name through deployment descriptor data.) If you configure a foreign cell binding to CellB in the local cell, applications running in the local cell can look up AccountHome with a JNDI name of `cell/cells/CellB/persistent/ejb/AccountHome`. In both lookup names, the suffix `persistent/ejb/AccountHome` is relative to the Cell Root context in CellB.

The foreign cell and the local cell must have different names.

Naming and directories: Resources for learning

Additional information and guidance on naming and directories is available on various Internet sites.

Use the following links to find relevant supplemental information about naming and directories. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

The naming service provided with WebSphere Application Server Versions 6.x and 7.0 is the same as that provided for Version 5, thus information on the Version 5.0 naming and directories applies to Versions 6.x and 7.0.

The following links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming instructions and examples

- Naming in WebSphere Application Server V5: Impact on Migration and Interoperability, http://www.ibm.com/developerworks/websphere/library/techarticles/0305_weiner/weiner.html
- *WebSphere Application Server V6.1:* System Management Configuration Handbook, SG24-7304-00, http://www.redbooks.ibm.com/abstracts/SG247304.html?Open
- *IBM WebSphere Developer Technical Journal:* Co-hosting multiple versions of J2EE applications, http://www.ibm.com/developerworks/websphere/techjournal/0405_poddar/0405_poddar.html

Programming specifications

- Specifications and API documentation

Developing applications that use JNDI

References to enterprise bean (EJB) homes and other artifacts such as data sources are bound to the WebSphere Application Server name space. These objects can be obtained through Java Naming and Directory Interface (JNDI). Before you can perform any JNDI operations, you need to get an initial context. You can use the initial context to look up objects bound to the name space.

About this task

The following examples describe how to get an initial context and how to perform lookup operations.
- Getting the default initial context
- Getting an initial context by setting the provider URL property
- Setting the provider URL property to select a different root context as the initial context
- Looking up an EJB home with JNDI

In these examples, the default behavior of features specific to the WebSphere Application Server JNDI Context implementation is used.

The WebSphere Application Server JNDI context implementation includes special features. JNDI caching enhances performance of repeated lookup operations on the same objects. Name syntax options offer a choice of a name syntaxes, one optimized for typical JNDI clients, and one optimized for interoperability with CosNaming applications. Most of the time, the default behavior of these features is the preferred behavior. However, sometimes you should modify the behavior for specific situations.

JNDI caching and name syntax options are associated with a javax.naming.InitialContext instance. To select options for these features, set properties that are recognized by the WebSphere Application Server initial context factory. To set JNDI caching or name syntax properties which will be visible to the initial context factory, do the following:

1. Optional: Configure JNDI caches

   JNDI caching can greatly increase performance of JNDI lookup operations. By default, JNDI caching is enabled. In most situations, this default is the desired behavior. However, in specific situations, use the other JNDI cache options.

   Objects are cached locally as they are looked up. Subsequent lookups on cached objects are resolved locally. However, cache contents can become stale. This situation is not usually a problem, since most objects you look up do not change frequently. If you need to look up objects which change relatively frequently, change your JNDI cache options.

   JNDI clients can use several properties to control cache behavior.

   You can set properties:

- From the command line by entering the actual string value. For example:

  ```
  java -Dcom.ibm.websphere.naming.jndicache.maxentrylife=1440
  ```

- In a jndi.properties file by creating a file named jndi.properties as a text file with the desired properties settings. For example:

  ```
  ...
  com.ibm.websphere.naming.jndicache.cacheobject=none
  ...
  ```

  Include the file as the beginning of the classpath, so that the class loader loads your copy of jndi.properties before any other copies.

- Within a Java program by using the PROPS.JNDI_CACHE* Java constants, defined in the com.ibm.websphere.naming.PROPS file. The constant definitions follow:

  ```
  public static final String JNDI_CACHE_OBJECT =
   "com.ibm.websphere.naming.jndicache.cacheobject";
  public static final String JNDI_CACHE_OBJECT_NONE      = "none";
  public static final String JNDI_CACHE_OBJECT_POPULATED = "populated";
  public static final String JNDI_CACHE_OBJECT_CLEARED   = "cleared";
  public static final String JNDI_CACHE_OBJECT_DEFAULT   =
   JNDI_CACHE_OBJECT_POPULATED;

  public static final String JNDI_CACHE_NAME =
   "com.ibm.websphere.naming.jndicache.cachename";
  public static final String JNDI_CACHE_NAME_DEFAULT = "providerURL";

  public static final String JNDI_CACHE_MAX_LIFE =
   "com.ibm.websphere.naming.jndicache.maxcachelife";
  public static final int    JNDI_CACHE_MAX_LIFE_DEFAULT = 0;

  public static final String JNDI_CACHE_MAX_ENTRY_LIFE =
   "com.ibm.websphere.naming.jndicache.maxentrylife";
  public static final int    JNDI_CACHE_MAX_ENTRY_LIFE_DEFAULT = 0;
  ```

  To use the previous properties in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor as follows:

  ```
  java.util.Hashtable env = new java.util.Hashtable();
  ...

  // Disable caching
  env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE); ...
  javax.naming.Context initialContext = new javax.naming.InitialContext(env);
  ```

Example: Controlling JNDI cache behavior from a program

Following are examples that illustrate how you can use JNDI cache properties to achieve the desired cache behavior. Cache properties take effect when an InitialContext object is constructed.

```
import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.naming.Context;
import com.ibm.websphere.naming.PROPS;

/*****
 Caching discussed in this section pertains to the WebSphere Application
 Server initial context factory. Assume the property,
 java.naming.factory.initial, is set to
 "com.ibm.websphere.naming.WsnInitialContextFactory" as a
 java.lang.System property.
*****/

Hashtable env;
Context ctx;

// To clear a cache:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_CLEARED);
```

```
ctx = new InitialContext(env);

// To set a cache's maximum cache lifetime to 60 minutes:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_MAX_LIFE, "60");
ctx = new InitialContext(env);

// To turn caching off:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
ctx = new InitialContext(env);

// To use caching and no caching:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_POPULATED);
ctx = new InitialContext(env);
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
Context noCacheCtx = new InitialContext(env);

Object o;

// Use caching to look up home, since the home should rarely change.
o = ctx.lookup("com/mycom/MyEJBHome");
// Narrow, etc. ...

// Do not use cache if data is volatile.
o = noCacheCtx.lookup("com/mycom/VolatileObject");
// ...
```

Example: Looking up a JavaMail session with JNDI

The following example shows a lookup of a JavaMail resource:

```
// Get the initial context as shown above
...
Session session =
     (Session) initialContext.lookup("java:comp/env/mail/MailSession");
```

2. Optional: Specify the name syntax

   INS syntax is designed for JNDI clients that need to interoperate with CORBA applications. This syntax allows a JNDI client to make the proper mapping to and from a CORBA name. INS syntax is very similar to the JNDI syntax with the additional special character, dot (.). Dots are used to delimit the id and kind fields in a name component. A dot is interpreted literally when it is escaped. Only one unescaped dot is allowed in a name component. A name component with a non-empty id field and empty kind field is represented with only the id field value and must not end with an unescaped dot. An empty name component (empty id and empty kind field) is represented with a single unescaped dot. An empty string is not a valid name component representation.

   JNDI name syntax is the default syntax and is suitable for typical JNDI clients. This syntax includes the following special characters: forward slash (/) and backslash (\). Components in a name are delimited by a forward slash. The backslash is used as the escape character. A forward slash is interpreted literally if it is escaped, that is, preceded by a backslash. Similarly, a backslash is interpreted literally if it is escaped.

   Most WebSphere applications use JNDI to look up EJB objects and do not need to look up objects bound by CORBA applications. Therefore, the default name syntax used for JNDI names is the most convenient. If your application needs to look up objects bound by CORBA applications, you may need to change your name syntax so that all CORBA CosNaming names can be represented.

   JNDI clients can set the name syntax by setting a property. The property setting is applied by the initial context factory when you instantiate a new java.naming.InitialContext object. Names specified in JNDI operations on the initial context are parsed according to the specified name syntax.

   You can set the property:

- From a command line, enter the actual string value. For example:

  ```
  java -Dcom.ibm.websphere.naming.name.syntax=ins
  ```
- Create a file named jndi.properties as a text file with the desired properties settings. For example:

  ```
  ...
  com.ibm.websphere.naming.name.syntax=ins
  ...
  ```

  Include the file at the beginning of the classpath, so that the class loader loads your copy of jndi.properties before any other copies.
- Use the PROPS.NAME_SYNTAX* Java constants, defined in the com.ibm.websphere.naming.PROPS file, in a Java program. The constant definitions follow:

  ```
  public static final String NAME_SYNTAX =
      "com.ibm.websphere.naming.name.syntax";
  public static final String NAME_SYNTAX_JNDI = "jndi";
  public static final String NAME_SYNTAX_INS  = "ins";
  ```

  To use the previous properties in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor as follows:

  ```
  java.util.Hashtable env = new java.util.Hashtable();
  ...
  env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS); // Set name syntax to INS
  ...
  javax.naming.Context initialContext = new javax.naming.InitialContext(env);
  ```

Example: Setting the syntax used to parse name strings

The name syntax property can be passed to the InitialContext constructor through its parameter, in the System properties, or in a jndi.properties file. The initial context and any contexts looked up from that initial context parse name strings based on the specified syntax.

The following example shows how to set the name syntax to make the initial context parse name strings according to INS syntax.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS; // WebSphere naming constants
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, ...);
env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS);
Context initialContext = new InitialContext(env);
// The following name maps to a CORBA name component as follows:
//    id = "a.name", kind = "in.INS.format"
// The unescaped dot is used as the delimiter.
// Escaped dots are interpreted literally.
java.lang.Object o = initialContext.lookup("a\\.name.in\\.INS\\.format");
...
```

INS name syntax requires that embedded periods (.) in a name such as `in.INS.format` be escaped using a backslash character (\). In a Java String literal, a backslash character (\) must be escaped with another backslash character (\\).

Example: Getting the default initial context

There are various ways for a program to get the default initial context.

The following example gets the default initial context. Note that no provider URL is passed to the javax.naming.InitialContext constructor.

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
...
Context initialContext = new InitialContext();
...
```

The default initial context returned depends the runtime environment of the Java Naming and Directory Interface (JNDI) client. Following are the initial contexts returned in the various environments:

Thin client
The initial context is the server root context of the server running on the local host at port 2809.

Pure client
The initial context is the context specified by the java.naming.provider.url property passed to launchClient command with the -CCD command line parameter. The context usually is the server root context of the server at the address specified in the URL, although it is possible to construct a corbaname or corbaloc URL which resolves to some other context.

If no provider URL is specified, it is the server root context of the server running on the host and port specified by the `-CCproviderURL`, or `-CCBootstrapHost` and `-CCBootstrapPort` command line parameters. The default host is the local host, and the default port is 2809.

Server process
The initial context is the server root context for that process.

Even though no provider URL is explicitly specified in the above example, the InitialContext constructor might find a provider URL defined in other places that it searches for property settings.

Users of properties which affect ORB initialization should read the rest of this section for a deeper understanding of exactly how initial contexts are obtained.

Determining which server is used to obtain the initial context

WebSphere Application Server name servers are CORBA CosNaming name servers, and the product provides a CosNaming JNDI plug-in implementation for JNDI clients to perform naming operations on product namespaces. The CosNaming plug-in implementation is selected through a JNDI property that is passed to the InitialContext constructor. This property is `java.naming.factory.initial`, and it specifies the initial context factory implementation to use to obtain an initial context. The factory returns a javax.naming.Context instance, which is part of its implementation.

The initial context factory, `com.ibm.websphere.naming.WsnInitialContextFactory`, is typically used by applications to perform JNDI operations. The WebSphere Application Server runtime environment is set up to use this initial context factory if one is not specified explicitly by the JNDI client. When the initial context factory is invoked, an *initial context* is obtained. The following paragraphs explain how the initial context factory obtains the initial context in client and server environments.

- **Registration of initial references in server processes**

  Every WebSphere Application Server has an ORB used to receive and dispatch invocations on objects running in that server. Services running in the server process can register initial references with the ORB. Each initial reference is registered under a key, which is a string value. An initial reference can be any CORBA object. WebSphere Application Server name servers register several initial contexts as initial references under predefined keys. Each name server initial reference is an instance of the interface org.omg.CosNaming.NamingContext.
- **Obtaining initial references in pure client processes**

  Pure JNDI clients, that is, JNDI clients which are not running in a WebSphere Application Server process, also have an ORB instance. This client ORB instance can be passed to the InitialContext constructor, but typically the initial context factory creates and initializes the client ORB instance transparently. A client ORB can be initialized with initial references, but the initial references most likely

resolve to objects running in some server. The initial context factory does not define any default initial references when it initializes an ORB. If the `resolve_initial_references` method is invoked on the client ORB when no initial references have been configured, the method invocation fails. This condition is typical for pure client processes. To obtain an initial NamingContext reference, the initial context factory must invoke `string_to_object` with an IIOP type CORBA object URL, such as `corbaloc:iiop:myhost:2809`. The URL specifies the address of the server from which to obtain the initial context. The host and port information is extracted from the provider URL passed to the InitialContext constructor.

If no provider URL is defined, the WebSphere Application Server initial context factory uses the default provider URL of `corbaloc:iiop:localhost:2809`.

The `string_to_object` ORB method resolves the URL and communicates with the target server ORB to obtain the initial reference.

- **Obtaining initial references in server processes**

  If the JNDI client is running in a WebSphere Application Server process, the initial context factory obtains a reference to the server ORB instance if the JNDI client does not provide an ORB instance. Typically, JNDI clients running in server processes use the server ORB instance; that is, they do not pass an ORB instance to the InitialContext constructor. The name server which is running in the server process sets a provider URL as a java.lang.System property to serve as the default provider URL for all JNDI clients in the process. This default provider URL is `corbaloc:rir:/NameServiceServerRoot`. This URL resolves to the server root context for that server. (The URL is equivalent to invoking `resolve_initial_references` on the ORB with a key of NameServiceServerRoot. The name server registers the server root context as an initial reference under that key.)

- **Understanding the legacy ORB protocol**

  Releases previous to WebSphere Application Server Version 5 used a different ORB implementation, which used a legacy protocol in contrast with the Interoperable Name Service (INS) protocol now used. This change has affected the implementation of the initial context factory. **Certain types of pure clients can experience different behavior when getting initial JNDI contexts as compared to previous releases of WebSphere Application Server**. This behavior is discussed in more detail below.

  The following ORB properties are used with the legacy ORB protocol for ORB initialization and are now deprecated:
  – com.ibm.CORBA.BootstrapHost
  – com.ibm.CORBA.BootstrapPort

  The new INS ORB is different in a major respect, in that it exhibits no default behavior if no initial references are defined.

  In the legacy ORB, the bootstrap host and port values defaulted to `localhost` and `900`.

  All initial references were obtained from the server running on the bootstrap host and port. So, if the ORB user provided no bootstrap host and port, all initial references are resolved from the server running on the local host at port 900. The INS ORB has no concept of bootstrap host or bootstrap port. All initial references are defined independently. That is, different initial references could resolve to different servers. If `ORB.resolve_initial_references` is invoked with a key such that the ORB is not initialized with an initial reference having that key, the call fails.

  In releases previous to Version 5, the initial context factory invoked `resolve_initial_references` on the ORB in the absence of any provider URL. This action succeeded if a name server at the default bootstrap host and port was running. In the current release, with the INS ORB, this would fail. (Actually, the ORB would fall back to the legacy protocol during the deprecation period, but when the legacy protocol is no longer supported, the operation would fail.)

  The initial context factory now uses a default provider URL of `corbaloc:iiop:localhost:2809`, and invokes `string_to_object` with the provider URL.

  This operation preserves the behavior that pure clients in previous releases experienced when they set no ORB bootstrap properties or provider URL. **However, this different initial context factory implementation changes the behavior experienced by certain legacy pure clients, which do not specify a provider URL**:
  – Clients which set the ORB bootstrap properties listed above when getting an initial context.

– Clients which supply their own ORB instance to the InitialContext constructor.

There are two ways to circumvent this change of behavior:

– Always specify an IIOP type provider URL. This approach does not depend on the bootstrap host and port properties and continues to work when support for the bootstrap host and port properties is removed. For example, you can express bootstrap host and port property values of `myHost` and `2809`, respectively, as `corbaloc:iiop:myHost:2809`.

– Use an rir type provider URL:
  - Specify `corbaloc:rir:/NameServiceServerRoot` if the ORB is initialized to use a Version 5 server as the bootstrap server.
  - Specify `corbaname:rir:/NameService#domain/legacyRoot` if the ORB is initialized to use a Version 4.0.x server as the bootstrap server.
  - Specify `corbaloc:rir:/NameService` if the ORB is initialized to use a server other than a Version 5 or 4.0.x server as the bootstrap server.

  URLs of this type are equivalent to invoking `resolve_initial_references` on the ORB with the specified key. If the bootstrap host and port properties are being used to initialize the ORB, this approach will not work when the bootstrap and host properties are no longer supported.

- **The InitialContext constructor search order for JNDI properties**

  If the code snippet shown at the beginning of this section is executed by an application, the bootstrap server depends on the value of the property, java.naming.provider.url.

  If the property is not set (in server processes the default value is set as a system property), the default host of `localhost` and default port of `2809` are used as the address of the server from which to obtain the initial context.

  The JNDI specification describes where the InitialContext constructor looks for java.naming.provider.url property settings, but briefly, the property is picked up from the following places in the order shown:

  InitialContext constructor
  This does not apply to the above example because the example uses the empty InitalContext constructor.

  System environment
  You can add JNDI properties to the system environment as an option on the Java command invocation and by program code. The recommended way to set the provider URL in the system environment is as an option supplied to the Java command invocation. Setting the provider URL in this manner is not temporal, so that getting a default initial context will always yield the same result. It is generally recommended that program code not set the provider URL property in the system environment because as a side-effect, this could adversely affect other, possibly unrelated, code running elsewhere in the same process.

  jndi.properties file
  There may be many jndi.properties files that are within the scope of the class loader in effect. All jndi.properties files are used for setting JNDI properties, but the provider URL setting is determined by the first jndi.properties file returned by the class loader.

Example: Getting an initial context by setting the provider URL property

In general, Java Naming and Directory Interface (JNDI) clients should assume the correct environment is already configured so there is no need to explicitly set property values and pass them to the `InitialContext` constructor. However, a JNDI client might need to access a namespace other than the one identified in its environment. In this case, it is necessary to explicitly set the java.naming.provider.url (provider URL) property used by the `InitialContext` constructor. A provider URL contains bootstrap server information that the initial context factory can use to obtain an initial context. Any property values passed in directly to the `InitialContext` constructor take precedence over settings of those same properties found elsewhere in the environment.

You can use two different provider URL forms with theWebSphere Application Server initial context factory:

- A CORBA object URL
- An IIOP URL

CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. CORBA object URLs are part of the OMG CosNaming Interoperable Naming Specification. A corbaname URL, for example, can include initial context and lookup name information and can be used as a lookup name without the need to explicitly obtain another initial context. The IIOP URLs are the legacy JNDI format, but are still supported by the WebSphere Application Server initial context factory.

The following examples illustrate the use of these URLs.

- "Using a CORBA object URL"
- "Using a CORBA object URL with multiple name server addresses"
- "Using a CORBA object URL from a non-WebSphere Application Server JNDI implementation" on page 1422
- "Using an IIOP URL" on page 1422

Using a CORBA object URL

This example shows a CORBA object URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Using a CORBA object URL with multiple name server addresses

CORBA object URLs can contain more than one bootstrap address. You can use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap addresses for all servers in the cluster in the URL. The operation succeeds if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address list will be processed. For example, the second bootstrap address may be used to obtain the initial context even though the server at the first bootstrap address in the list is available.

Multiple-address provider URLs resolving to servers on non-z/OS systems cannot contain bootstrap addresses for node agent processes. The URLs should only contain the bootstrap addresses of members of the same cluster. Otherwise, incorrect behavior might occur. When resolving to servers running on the z/OS operating system, the URL can contain bootstrap addresses for node agent processes.

An example of a corbaloc URL with multiple addresses follows.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
// All of the servers in the provider URL below are members of
// the same cluster.
env.put(Context.PROVIDER_URL,
     "corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810");
Context initialContext = new InitialContext(env);
...
```

Using a CORBA object URL from a non-WebSphere Application Server JNDI implementation

Initial context factories for CosNaming JNDI plug-in implementations other than the WebSphere Application Server initial context factory most likely obtain an initial context using the object key, `NameService`. When you use such a context factory to obtain an initial context from a WebSphere Application Server name server, the initial context is the cell root context. Since system artifacts such as EJB homes associated with a server are bound under the server's server root context, names used in JNDI operations must be qualified. If you want to use relative names, ensure your initial context is the server root context under which the target object is bound. In order to make the server root context the initial context, specify a corbaloc provider URL with an object key of `NameServiceServerRoot`.

This example shows a CORBA object type URL from a non-WebSphere Application Server JNDI implementation. This example assumes full CORBA object URL support by the non-WebSphere Application Server JNDI implementation. The object key of `NameServiceServerRoot` is specified so that the initial context will be the specified server's server root context.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.somecompany.naming.TheirInitialContextFactory");
env.put(Context.PROVIDER_URL,
      "corbaname:iiop:myhost.mycompany.com:9810/NameServiceServerRoot");
Context initialContext = new InitialContext(env);
...
```

If qualified names are used, you can use the default key of `NameService`.

Using an IIOP URL

The IIOP type of URL is a legacy format which is not as flexible as CORBA object URLs. However, URLs of this type are still supported. The following example shows an IIOP type URL as the provider URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "iiop://myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Example: Setting the provider URL property to select a different root context as the initial context

Each server contains its own server root context, and, when bootstrapping to a server, the server root is the default initial JNDI context. Most of the time, this default is the desired initial context, since system artifacts such as EJB homes are bound there. However, other root contexts exist, which can contain bindings of interest. It is possible to specify a provider URL to select other root contexts.

Examples for selecting other root contexts follow:

- Initial root contexts with a CORBA object URL
- Initial root contexts with the namespace root property

Selecting the initial root context with a CORBA object URL

There are several object keys registered with the bootstrap server that you can use to select the root context for the initial context. To select a particular root context with a CORBA object URL object key, set the object key to the corresponding value. The default object key is NameService. Using JNDI yields the server root context. A table that lists the different root contexts and their corresponding object key follows:

Root Context	CORBA Object URL Object Key
Server Root	NameServiceServerRoot
Cell Persistent Root	NameServiceCellPersistentRoot
Cell Root	NameServiceCellRoot
Node Root	NameServiceNodeRoot

The following example shows the use of a corbaloc URL with the object key set to select the cell persistent root context as the initial context.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL,
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceCellPersistentRoot");
Context initialContext = new InitialContext(env);
...
```

Selecting the initial root context with the namespace root property

You can also select the initial root context by passing a namespace root property setting to the InitialContext constructor. Generally, the object key setting described above is sufficient. Sometimes a property setting is preferable. For example, you can set the root context property on the Java invocation to make which server root is being used as the initial context transparent to the application. The default server root property setting is `defaultroot`, which yields the server root context.

Root Context	Namespace Root Property Value
Server Root	bootstrapserverroot
Cell Persistent Root	cellpersistentroot
Cell Root	cellroot
Node Root	bootstrapnoderoot

The initial context factory ignores the namespace root property if the provider URL contains an object key other than `NameService`.

The following example shows use of the namespace root property to select the cell persistent root context as the initial context. Note that available constants are used instead of hard-coding the property name and value.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
```

```
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
env.put(PROPS.NAME_SPACE_ROOT, PROPS.NAME_SPACE_ROOT_CELL_PERSISTENT);
Context initialContext = new InitialContext(env);
...
```

Example: Looking up an EJB home or business interface with JNDI

Most applications that use Java Naming and Directory Interface (JNDI) run in a container. Some do not. The name used to look up an object depends on whether or not the application is running in a container. Sometimes it is more convenient for an application to use a corbaname URL as the lookup name. Container-based JNDI clients and thin Java clients can use a corbaname URL.

The following examples show how to perform JNDI lookups from different types of applications.

- "JNDI lookup from an application running in a container"
- "JNDI lookup from an application that does not run in a container" on page 1425
- "JNDI lookup with a corbaname URL" on page 1427

JNDI lookup from an application running in a container

Applications that run in a container can use `java:` lookup names. Lookup names of this form provide a level of indirection such that the lookup name used to look up an object is not dependent on the object's name as it is bound in the name server's namespace. The deployment descriptors for the application provide the mapping from the `java:` name and the name server lookup name. The container sets up the `java:` namespace based on the deployment descriptor information so that the `java:` name is correctly mapped to the corresponding object.

The following example shows a lookup of an EJB 3.0 remote business interface. The actual home lookup name is determined by the interface's ibm-ejb-jar-bnd.xml binding file, if present, or by the default name assigned by the EJB container if no binding file is present. For more information, see Default bindings for business interfaces and homes and User-defined bindings for EJB business interfaces and homes.

```
// Get the initial context as shown in a previous example.
...
// Look up the business interface using the JNDI name.
try {
   java.lang.Object ejbBusIntf =
      initialContext.lookup(
         "java:comp/env/com/mycompany/accounting/Account");
   accountIntf =
      (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
   catch (NamingException e) { // Error getting the business interface
   ...
}
```

The following example shows a lookup of an EJB 1.x or 2.x EJB home. The actual home lookup name is determined by the application's deployment descriptors. The enterprise bean (EJB) resides in an EJB container, which provides an interface between the bean and the application server on which it resides.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome =
      initialContext.lookup(
         "java:comp/env/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
   catch (NamingException e) { // Error getting the home interface
   ...
}
```

JNDI lookup from an application that does not run in a container

Applications that do not run in a container cannot use `java:` lookup names because it is the container which sets the `java:` namespace up for the application. Instead, an application of this type must look the object up directly from the name server. Each application server contains a name server. System artifacts such as EJB homes are bound relative to the server root context in that name server. The various name servers are federated by means of a system namespace structure. The recommended way to look up objects on different servers is to qualify the name so that the name resolves from any initial context in the cell. If a relative name is used, the initial context must be the same server root context as the one under which the object is bound. The form of the qualified name depends on whether the qualified name is a topology-based name or a fixed name. Examples of each form of qualified name follow.

- **Topology-based qualified names**

  Topology-based qualified names traverse through the system namespace to the server root context under which the target object is bound. A topology-based qualified name resolves from any initial context in the cell.

  The topology-based qualified name depends on whether the object resides on a single server or server cluster. Examples of each lookup follow.

  Single server

  The following example shows a lookup of an EJB business interface that is running in the single server, MyServer, configured in the node, Node1.

```
// Get the initial context as shown in a previous example.
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
// Look up the business interface using the JNDI name
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
      "cell/nodes/Node1/servers/MyServer/com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
catch (NamingException e) { // Error getting the business interface
   ...
}
```

  The following example shows a lookup of an EJB home that is running in the single server, MyServer, configured in the node, Node1.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "cell/nodes/Node1/servers/MyServer/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

  Server cluster

  The example below shows a lookup of an EJB business interface which is running in the cluster, MyCluster. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in a previous example.
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
   // Look up the business interface using the JNDI name
```

```
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
      "cell/clusters/MyCluster/com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
catch (NamingException e) { // Error getting the business interface
   ...
}
```

The example below shows a lookup of an EJB home which is running in the cluster, MyCluster. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
   // Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "cell/clusters/MyCluster/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

- **Fixed qualified names**

  If the target object has a cell-scoped fixed name defined for it, you can use its qualified form instead of the topology-based qualified name. Even though the topology-based name works, the fixed name does not change with the specific cell topology or with the movement of the target object to a different server.

  An example lookup of an EJB business interface with a qualified fixed name follows.

```
// Get the initial context as shown in a previous example.
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
// Look up the business interface using the JNDI name
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
     "cell/persistent/com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
   }
catch (NamingException e) { // Error getting the business interface
...

}
```

  An example lookup with a qualified fixed name is shown below.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
     "cell/persistent/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
   }
catch (NamingException e) { // Error getting the home interface
...
}
```

JNDI lookup with a corbaname URL

A corbaname can be useful at times as a lookup name. If, for example, the target object is not a member of the federated namespace and cannot be located with a qualifiied name, a corbaname can be a convenient way to look up the object.

A lookup of an EJB business interface with a corbaname URL follows.

```
// Get the initial context as shown in a previous example.
...
// Look up the business interface using a corbaname URL.
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
      "corbaname:iiop:someHost:2809#com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
catch (NamingException e) { // Error getting the business interface
   ...
}
```

A lookup with a corbaname URL follows.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using a corbaname URL
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "corbaname:iiop:someHost:2809#com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

JNDI interoperability considerations

You must take extra steps to enable your programs to interoperate with non-product JNDI clients and to bind resources from MQSeries to a namespace.

EJB clients running in an environment other than WebSphere Application Server accessing EJB applications running on WebSphere Application Server servers

When an enterprise bean (EJB) application running in WebSphere Application Server is accessed by a non-product EJB client, the JNDI initial context factory is presumed to be a non-product implementation. In this case, the default initial context is the cell root. If the JNDI service provider being used supports CORBA object URLs, the corbaname format can be used to look up the EJB home.

The construction of the stringified name depends on whether the object is installed on a single server or cluster.

Single server

Following is a URL that has the bootstrap host **myHost**, the port **2809**, and the enterprise bean installed in the server **server1** in node **node1** and bound in that server under the name **myEJB**:

```
initialContext.lookup(
  "corbaname:iiop:myHost:2809#cell/nodes/node1/servers/server1/myEJB");
```

Server cluster

Following is a URL that has the bootstrap host **myHost**, the port **2809**, and the enterprise bean installed in a server cluster named **myCluster** and bound in that cluster under the name **myEJB**:

```
initialContext.lookup(
    "corbaname:iiop:myHost:2809#cell/clusters/myCluster/myEJB");
```

The lookup works with any name server bootstrap host and port configured in the same cell.

The lookup also works if the bootstrap host and port belong to a member of the cluster itself. To avoid a single point of failure, the bootstrap server host and port for each cluster member can be listed in the URL as follows:

```
initialContext.lookup(
    "corbaname:iiop:host1:9810,:host2:9810#cell/clusters/myCluster/myEJB");
```

The name prefix **cell/clusters/myCluster/** is not necessary if boostrapping to the cluster itself, but it will work. The prefix is needed, however, when looking up enterprise beans in other clusters. Name bindings under the **clusters** context are implemented on the name server to resolve to the server root of a running cluster member during a lookup; thus avoiding a single point of failure.

Without CORBA object URL support

If the JNDI initial context factory being used does not support CORBA object URLs, the initial context can be obtained from the server, and the lookup can be performed on the initial context as follows:

```
Hashtable env = new Hashtable();
env.put(CONTEXT.PROVIDER_URL, "iiop://myHost:2809");
Context ic = new InitialContext(env);
Object o = ic.lookup("cell/clusters/myCluster/myEJB");
```

Binding resources from MQSeries 5.2

In releases previous to WebSphere Application Server Version 5.0, the MQSeries jmsadmin tool could be used to bind resources to the namespace. When used with a WebSphere Application Server namespace, the resource is bound within a transient partition in the namespace and does not persist past the life of the server process. Instead of binding the MQSeries resources with the jmsadmin tool, bind them from the administrative console, under **Resources** in the console navigation tree.

JNDI caching

To increase the performance of Java Naming and Directory Interface (JNDI) operations, the product JNDI implementation employs caching to reduce the number of remote calls to the name server for lookup operations. For most cases, use the default cache setting.

When an InitialContext object is instantiated, an association is established between the InitialContext instance and a cache. The initial context and any contexts returned directly or indirectly from a lookup on the initial context are all associated with that same cache instance. By default, the association is based on the provider URL, in particular, the host name and port. The caller can specify the cache name to override this default behavior. A cache instance of a given name is shared by all instances of InitialContext configured to use a cache of that name which were created with the same context class loader in effect. Two enterprise bean (EJB) applications running in the same server will use their own cache instances, if they are using different context class loaders, even if the cache names are the same.

After an association between an InitialContext instance and cache is established, the association does not change. A `javax.naming.Context` object returned from a lookup operation inherits the cache association of the Context object on which the lookup was performed. Changing cache property values with the `Context.addToEnvironment()` or `Context.removeFromEnvironment()` method does not affect cache behavior. You can change properties affecting a given cache instance with each InitialContext instantiation.

A cache is restricted to a process and does not persist past the life of that process. A cached object is returned from lookup operations until either the maximum cache life for the cache is reached, or the maximum entry life for the object's cache entry is reached. After this time, a lookup on the object causes the cache entry for the object to be refreshed. By default, caches and cache entries have unlimited lifetimes.

Usually, cached objects are relatively static entities, and objects becoming stale is not a problem. However, you can set timeout values on cache entries or on a cache so that cache contents are periodically refreshed.

If a bind or rebind operation is executed on an object, the change is not reflected in any caches other than the one associated with the context from which the bind or rebind was issued. This scenario is most likely to happen when multiple processes are involved, since different processes do not share the same cache, and context objects in all threads in a process typically share the same cache instance for a given name service provider.

JNDI cache settings

Various Java Naming and Directory Interface (JNDI) cache property settings follow. Ensure that all property values are string values.

com.ibm.websphere.naming.jndicache.cachename

The name of the cache to associate with an initial context instance can be specified with this property.

It is possible to create multiple InitialContext instances, each operating on the namespace of a different name server. By default, objects from each bootstrap address are cached separately, since they each involve independent namespaces and name collisions could occur if they used the same cache. The provider URL specified when the initial context is created by default serves as the basis for the cache name. With this property, a JNDI client can specify a cache name. Valid options for cache names follow:

Valid options	Resulting cache behavior
providerURL (default)	Use the value for java.naming.provider.url property as the basis for the cache name. Cache names are based on the bootstrap host and port specified in the URL. The boostrap host is normalized to a fully qualfied name, if possible. For example, ″corbaname:iiop:server1:2809#some/starting/context″ and ″corbaloc:iiop://server1″ are normalized to the same cache name. If no provider URL is specified, a default cache name is used.
Any string	Use the specified string as the cache name. You can use any arbitrary string with a value other than ″providerURL″ as a cache name.

com.ibm.websphere.naming.jndicache.cacheobject

Turn caching on or off and clear an existing cache with this property.

By default, when an InitialContext is instantiated, it is associated with an existing cache or, if one does not exist, a new one is created. An existing cache is used with its existing contents. In some circumstances, this behavior is not desirable. For example, when objects that are looked up change frequently, they can become stale in the cache. Other options are available. The following table lists these other options along with the corresponding property value.

Valid values	Resulting cache behavior
populated (default)	Use a cache with the specified name. If the cache already exists, leave existing cache entries in the cache; otherwise, create a new cache.
cleared	Use a cache with the specified name. If the cache already exists, clear all cache entries from the cache; otherwise, create a new cache.
none	Do not cache. If this option is specified, the cache name is irrelevant. Therefore, this option will not disable a cache that is already associated with other InitialContext instances. The InitialContext that is instantiated is not associated with any cache.

com.ibm.websphere.naming.jndicache.maxcachelife

Impose a limit to the age of a cache with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to `cleared`. This property enables a JNDI client to set the maximum life of a cache. This property differs from the maxentrylife property (below) in that the entire cache is cleared when the cache lifetime is reached. The table below lists the various maxcachelife values and their affect on cache behavior:

Valid options	Resulting cache behavior
0 (default)	Make the cache lifetime unlimited.
Positive integer	Set the maximum lifetime of the entire cache, in minutes, to the specified value. When the maximum lifetime for the cache is reached, the next attempt to read any entry from the cache causes the cache to be cleared

com.ibm.websphere.naming.jndicache.maxentrylife

Impose a limit to the age of individual cache entries with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to `cleared`. This property enables a JNDI client to set the maximum lifetime of individual cache entries. This property differs from the maxcachelife property in that individual entries are refreshed individually as their maximum lifetime reached. This might avoid any noticeable change in performance that might occur if the whole cache is cleared at once. The table below lists the various maxentrylife values and their effect on cache behavior:

Valid options	Resulting cache behavior
0 (default)	Lifetime of cache entries is unlimited.
Positive integer	Set the maximum lifetime of individual cache entries, in minutes, to the specified value. When the maximum lifetime for an entry is reached, the next attempt to read the entry from the cache causes the individual cache entry to refresh.

JNDI to CORBA name mapping considerations

WebSphere Application Server name servers are an implementation of the CORBA CosNaming interface. The product provides a Java Naming and Directory Interface (JNDI) implementation which you can use to access CosNaming name servers through the JNDI interface. Issues can exist when mapping JNDI name strings to and from CORBA names.

Each component in a CORBA name consists of an `id` and `kind` field, but a JNDI name component consists of no such fields. Each component in a JNDI name is atomic. Typical JNDI clients do not need to make a distinction between the `id` and `kind` fields of a name component, or know how JNDI name strings map to CORBA names. JNDI clients of this sort can use the JNDI syntax described below. When a name is parsed according to JNDI syntax, each name component is mapped to the `id` field of the corresponding CORBA name component. The `kind` field always has an empty value. This basic syntax is the least obtrusive to the JNDI client in that it has the fewest special characters. However, you cannot represent with this syntax a CORBA name with a non-empty `kind` field. This restriction can prevent EJB applications from interoperating with CORBA applications.

Some clients, however must interoperate with CORBA applications which use CORBA names with non-empty `kind` fields. These JNDI clients must make a distinction between `id` and `kind` so that JNDI names are correctly mapped to CORBA names, particularly when the CORBA names contain components with non-empty `kind` fields. Such JNDI clients can use the INS name syntax. With its additional special character, you can use INS to represent any CORBA name. Use of this syntax is not recommended unless it is necessary, because this syntax is more restrictive from the JNDI client's perspective in that the JNDI client must be aware that name components with multiple unescaped dots are syntactically invalid. INS name syntax is part of the OMG CosNaming Interoperable Naming Specification.

Developing applications that use CosNaming (CORBA Naming interface)

CORBA clients can perform naming operations on WebSphere Application Server name servers through the CosNaming interface.

About this task

The following examples show how to obtain an ORB instance and an initial context as well as how to look up an EJB home.

1. Get an initial context.
2. Perform desired CosNaming operations.

Example: Getting an initial context with CosNaming

In WebSphere Application Server, an initial context is obtained from a bootstrap server. The address for the bootstrap server consists of a host and port. To get an initial context, you must know the host and port for the server that is used as the bootstrap server.

Obtaining an initial context consists of two basic steps:
1. Obtain an ORB reference.
2. Use an ORB reference to get an initial context. Alternatively, use an existing ORB and invoke string_to_object with a CORBA object URL with multiple name server addresses to get an initial context.

Obtaining an ORB reference

Pure CosNaming clients, that is clients that are not running in a server process, must create and initialize an ORB instance with which to obtain the initial context. CosNaming clients which run in server processes can obtain a reference to the server ORB with a JNDI lookup. The following examples illustrate how to create and initialize a client ORB and how to obtain a server ORB reference.

Creating a client ORB instance

To create an ORB instance, invoke the static method, org.omg.CORBA.ORB.init. The init method requires a property set to the name of the ORB class you want to instantiate. An ORB implementation with the class name com.ibm.CORBA.iiop.ORB is included with the product. The WebSphere Application Server ORB recognizes additional properties with which you can specify initial references.

The basic steps for creating an ORB are as follows:
1. Create a Properties object.
2. Set the ORB class property to the product's ORB class.
3. Set the initial reference properties.
4. Invoke ORB.init, passing in the Properties object.

```
...
import java.util.Properties;
import org.omg.CORBA.ORB;
...
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass","com.ibm.ws390.orb.ORB");
props.put("com.ibm.CORBA.ORBInitRef.NameService",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameService");
props.put("com.ibm.CORBA.ORBInitRef.NameServiceServerRoot",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceServerRoot");
ORB _orb = ORB.init((String[])null, props);
...
```

Obtaining a reference to the server ORB

CosNaming clients which run in a server process can obtain a reference to the server ORB with a JNDI lookup on a java: name, shown as follows:

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
import org.omg.CORBA.ORB;
...
Context initialContext = new InitialContext();
ORB orb = (ORB) initialContext.lookup("java:comp/ORB");
...
```

Using an ORB reference to get an initial naming reference

There are two basic ways to get an initial CosNaming context. Both ways involve an ORB method invocation. The first way is to invoke the resolve_initial_references method on the ORB with an initial reference key. For this call to work, the ORB must be initialized with an initial reference for that key. The other way is to invoke the string_to_object method on the ORB, passing in a CORBA object URL with the host and port of the bootstrap server. The following examples illustrate both approaches.

Invoking resolve_initial_references

Once an ORB reference is obtained, invoke the resolve_initial_references method on the ORB to obtain a reference to the initial context. The following code example invokes resolve_initial_reference on an ORB reference.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Obtain ORB reference as shown in examples earlier in this section
...
org.omg.CORBA.Object obj = _orb.resolve_initial_references("NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Note that the key NameService is passed to the resolve_initial_references method. Other initial context keys are registered in product servers. For example, NameServiceServerRoot can be used to obtain a reference to the server root context in the bootstrap name server. For more information on the initial contexts registered in server ORBs, refer to “Initial context support” on page 1405.

Invoking string_to_object with a CORBA object URL

You can use an INS-compliant ORB to obtain an initial context even if the ORB is not initialized with any initial references or bootstrap properties, or if those property settings are for a different server than the name server from which you want to obtain the initial context. To obtain an initial context by explicitly specifying the bootstrap name server, invoke the string_to_object method on the ORB, passing in a CORBA object URL which contains the bootstrap server host and port.

The code in the example below invokes the string_to_object method on an existing ORB reference, passing in a CORBA object URL which identifies the desired initial context.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Obtain ORB reference as shown in examples earlier in this section
...
```

```
org.omg.CORBA.Object obj =
 orb.string_to_object("corbaloc:iiop:myhost.mycompany.com:2809/NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Note that the key NameService is used in the corbaloc URL. Other initial context keys are registered in product servers. For example, you can use NameServiceServerRoot to obtain a reference to the server root context in the bootstrap name server.

Using an existing ORB and invoking string_to_object with a CORBA object URL with multiple name server addresses to get an initial context

CORBA object URLs can contain more than one bootstrap server address. Use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap server addresses for all servers in the cluster in the URL. The operation will succeed if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address list will be processed. For example, the second bootstrap server address may be used to obtain the initial context even though the first bootstrap server in the list is available. An example of a corbaloc URL with multiple addresses follows.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Assume orb is an existing ORB instance
org.omg.CORBA.Object obj = orb.string_to_object(
   "corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810/NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Example: Looking up an EJB home with CosNaming

You can look up an EJB home or other CORBA object from a WebSphere Application Server name server through the CORBA CosNaming interface.

You can invoke `resolve` or `resolve_str` on the initial context, or you can invoke `string_to_object` on the ORB. You can use a qualified name so that the name resolves regardless of which name server the lookup is executed on, or use an unqualified name that only resolves from the server root context on the name server that actually contains the object binding. (The qualified name traverses the federated system namespace to the specified server root context.)

Qualified and unqualified names

Each application server contains a name server. System artifacts such as EJB homes are bound in that name server. The various name servers are federated by means of a system namespace structure. The recommended way to look up objects on different servers is to use a qualified name.

A qualified name can be a topology-based name, based on the name of the cluster or single server and node that contains the object.

You can define fixed qualified names for objects. With qualified names, you can look up objects residing on different servers from the same initial context by traversing the system namespace structure. Alternatively, you can use an unqualified name, but an unqualified name will only resolve using the name server associated with the object's application server.

CosNaming.resolve (and resolve_str) vs. ORB.string_to_object

If you have an initial context from any name server in a WebSphere Application Server cell, you can look up any CORBA object with a qualified name. You do not need additional host and port information for the target object's name server.

Alternatively, you can look up an object by invoking `string_to_object` on the ORB, passing in a corbaname URL. Typically, an IIOP type URL is specified, so the bootstrap address information required for an initial context must be contained in the URL. You can use a qualified or unqualified stringified name, but an unqualifed name resolves only if the initial context is from the name server in which the object is bound.

The following examples show CosNaming resolve operations using qualified topology-based lookup names and an unqualified lookup name.

CosNaming resolve operation using a qualified name

The topology-based qualified name for an object depends on whether the object is bound in a single server or a server cluster. Examples of each follow.

Single server

The following example shows the lookup of an EJB home that is running in a single server. The enterprise bean that is being looked up is running in the server, `MyServer`, on the node, `Node1`.

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
  "cell/nodes/Node1/servers/MyServer/mycompany/accounting/AccountEJB");
accountHome =
  (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Server cluster

The following example shows a lookup of an EJB home that is running in a cluster. The enterprise bean being that is looked up is running in the cluster, Cluster1. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
   "cell/clusters/Cluster1/mycompany/accounting/AccountEJB");
accountHome =
   (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

ORB string_to_object operation using an unqualified stringified name

If the resolve operation is being performed on the name server that contains the object, the system namespace does not need to be traversed, and you can use an unqualified lookup name. Note that this name does not resolve on other name servers. If an unqualified name is provided, the object key must be NameServiceServerRoot so that the correct initial context is selected. If a qualified name is provided, you can use the default key of NameService.

The following example shows a lookup of an EJB home. The enterprise bean that is being looked up is bound on the name server running on the host `myHost` on port `2809`. Note the object key of `NameServiceServerRoot`.

```
// Assume orb is an existing ORB instance
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = orb.string_to_object(
    "corbaname:iiop:myHost:2809/NameServiceServerRoot#mycompany/accounting");
accountHome =
    (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Chapter 13. Object Request Broker

Managing Object Request Brokers

An Object Request Broker (ORB) manages the interaction between clients and servers using the Internet InterORB Protocol (IIOP). There are several ways to manage an ORB. For example, you can use ORB custom property settings, or system property settings to configure an ORB, or you can provide objects during ORB initialization.

About this task

Default ORB property values are set when the product starts and the ORB service initializes. These properties control the run-time behavior of the ORB and can also affect the behavior of product components that are tightly integrated with the ORB, such as security. You might have to modify some ORB settings to fit your system requirements.

After an ORB instance is established in a process, changes to ORB properties do not affect the behavior of a running ORB instance. You must stop the process and restart it before the modified settings take effect.

A list of possible tasks for managing an ORB follows.

- Adjust timeout settings to improve handling of network failures.
- Tune the ORB. For example, if most of your initial method invocations are very small, you might want to set the com.ibm.CORBA.enableLocateRequest custom property to false.
- Adjust the size of General Inter-ORB Protocol (GIOP) fragments that the ORB uses. You might want to make this adjustment if your applications frequently send large requests.
- Change the port that the ORB listens on.
- Specify an alternative to the default RAS manager of the ORB.
- Change the maximum number of connection requests that can remain unhandled by the product ORB before the application server starts to reject new incoming connection requests.

Object Request Brokers

An Object Request Broker (ORB) manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.

The ORB provides a framework for clients to locate objects in the network and to call operations on those objects as if the remote objects are located in the same running process as the client, providing location transparency. The client calls an operation on a local object, known as a *stub*. The stub forwards the request to the remote object, where the operation runs and the results are returned to the client.

The client-side ORB is responsible for creating an IIOP request that contains the operation and required parameters, and for sending the request on the network. The server-side ORB receives the IIOP request, locates the target object, invokes the requested operation, and returns the results to the client. The client-side ORB demarshals the returned results and passes the result to the stub, which, in turn, returns to the client application, as if the operation had been run locally.

This product uses an ORB to manage communication between client applications and server applications as well as communication among product components. During product installation, default property values are set when the ORB is initialized. These properties control the run-time behavior of the ORB and can also affect the behavior of product components that are tightly integrated with the ORB, such as security. This product does not support the use of multiple ORB instances.

ORB service transport channels are used for ORB I/O operations within an application server environment. These transport chains are part of the channel framework function that provides a common networking service for all components.

Object Request Broker service settings

Use this page to configure the Java Object Request Broker (ORB) service.

To view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* **> Container services > ORB service**.

Several settings are available for controlling internal Object Request Broker (ORB) processing. You can use these settings to improve application performance in the case of applications that contain enterprise beans. You can make changes to these settings for the default server or any application server that is configured in the administrative domain.

Request timeout

Specifies the number of seconds to wait before timing out on a request message.

If you use command-line scripting, the full name of this system property is com.ibm.CORBA.RequestTimeout.

Data type	int
Units	Seconds
Default	180
Range	0 - largest integer recognized by Java

ORB tracing

Enables the tracing of ORB General Inter-ORB Protocol (GIOP) messages.

This setting affects two system properties: com.ibm.CORBA.Debug and com.ibm.CORBA.CommTrace. If you set these properties through command-line scripting, you must set both properties to `true` to enable the tracing of GIOP messages.

Data type	Boolean
Default	Not enabled (false)

Pass by reference

Specifies how the ORB passes parameters. If enabled, the ORB passes parameters by reference instead of by value, to avoid making an object copy. If you do not enable the pass by reference option, a copy of the parameter passes rather than the parameter object itself. This can be expensive because the ORB must first make a copy of each parameter object.

You can use this option only when the Enterprise JavaBeans (EJB) client and the EJB are on the same classloader. This requirement means that the EJB client and the EJB must be deployed in the same EAR file.

If the Enterprise JavaBeans (EJB) client and server are installed in the same instance or the product, and the client and server use remote interfaces, enabling the pass by reference option can improve performance up to 50%. The pass by reference option helps performance only where non-primitive object types are passed as parameters. Therefore, int and floats are always copied, regardless of the call model.

Note: Enable this property with caution because unexpected behavior can occur. If an object reference is modified by the callee, the caller's object is modified as well, since they are the same object.

If you use command-line scripting, the full name of this system property is com.ibm.CORBA.iiop.noLocalCopies.

Data type	Boolean
Default	Not enabled (false)

The use of this option for enterprise beans with remote interfaces violates Enterprise JavaBeans (EJB) Specification, Version 2.0 (see section 5.4). Object references passed to Enterprise JavaBeans (EJB) methods or to EJB home methods are not copied and can be subject to corruption.

Consider the following example:

```
Iterator iterator = collection.iterator();
MyPrimaryKey pk = new MyPrimaryKey();
while (iterator.hasNext()) {
   pk.id = (String) iterator.next();
   MyEJB myEJB = myEJBHome.findByPrimaryKey(pk);
}
```

In this example, a reference to the same MyPrimaryKey object passes into the product with a different ID value each time. Running this code with pass by reference enabled causes a problem within the application server because multiple enterprise beans are referencing the same MyPrimaryKey object. To avoid this problem, set the com.ibm.websphere.ejbcontainer.allowPrimaryKeyMutation system property to `true` when the pass by reference option is enabled. Setting the pass by reference option to `true` causes the EJB container to make a local copy of the PrimaryKey object. As a result, however, a small portion of the performance advantage of setting the pass by reference option is lost.

As a general rule, any application code that passes an object reference as a parameter to an enterprise bean method or to an EJB home method must be scrutinized to determine if passing that object reference results in loss of data integrity or in other problems.

After examining your code, you can enable the pass by reference option by setting the com.ibm.CORBA.iiop.noLocalCopies system property to `true`. You can also enable the pass by reference option in the administrative console. Click **Servers > Server Types > Application servers >** *server_name* **> Container services > ORB Service** and select **Pass by reference**.

Object Request Broker custom properties

There are several ways to configure an Object Request Broker (ORB). For example, you can use ORB custom property settings, or system property settings to configure an ORB, or you can provide objects during ORB initialization. If you use the following ORB custom properties to configure an ORB, remember that two types of default values exist for some of these properties: the Java SE Development Kit (JDK) default values and the WebSphere Application Server default values.

The JDK default is the value that the ORB uses for a property if the property is not specified in any way. The WebSphere Application Server default is the value that the WebSphere Application Server product sets for a property in one of the following files:

- The orb.properties file when an application server is installed.
- The server.xml file when an application server is configured.

Because WebSphere Application Server explicitly sets its default value, if both a WebSphere Application Server and a JDK default value are defined for a property, the WebSphere Application Server default takes precedence over the JDK default.

For more information about the different ways to specify ORB properties and the precedence order, read the JDK Diagnostic Guide for the version of the JDK that you are using.

The `orb.properties` file, that is located in the *was_home*/java/jre/lib directory, contains ORB custom properties that are initially set to the WebSphere Application Server default values during the product installation process.

You can use the administrative console to specify new values for these ORB custom properties. Any value that you specify takes precedence over any JDK or WebSphere Application Server default values for these properties. The ORB custom properties settings that you specify in the administrative console are stored in the `server.xml` system file and are passed to an ORB in a properties object whenever an ORB is initialized.

To use the administrative console to set ORB custom properties, click **Servers > Server Types > Application servers >** *server_name* **> Container services > ORB service > Custom properties**. You can then change the setting of one of the listed custom properties or click **New** to add a new property to the list. Then, click **Apply** to save your change. When you finish making changes, click **OK** and then click **Save** to save your changes.

To use the java command on a command line, use the `-D` option; for example:

```
java -Dcom.ibm.CORBA.propname1=value1 -Dcom.ibm.CORBA.propname2=value2 ... application name
```

To use the launchclient command on a command line, prefix the property with `-CC`; for example:

```
launchclient yourapp.ear -CCDcom.ibm.CORBA.propname1=value1 -CCDcom.ibm.CORBA.propname2=value2
   ... optional application arguments
```

The Custom properties page might already include Secure Sockets Layer (SSL) properties that were added during product installation. A list of the additional properties that are associated with the ORB service follows. Unless otherwise indicated, the default values that are provided in the descriptions of these properties are the JDK default values.

com.ibm.CORBA.BootstrapHost

Specifies the domain name service (DNS) host name or IP address of the machine on which initial server contact for this client resides.

Note: This setting is deprecated.

For a command-line or programmatic alternative, read the topic *Client-side programming tips for the Object Request Broker service*.

com.ibm.CORBA.BootstrapPort

Specifies the port that the ORB uses to bootstrap to the machine on which the initial server contact for this client listens.

Note: This setting is deprecated.

For a command line or programmatic alternative, read the topic *Client-side programming tips for the Object Request Broker service*.

Default 2809

com.ibm.CORBA.ConnectTimeout

The com.ibm.CORBA.ConnectTimeout property specifies the maximum time, in second, that the client ORB waits before timing out when attempting to establish an IIOP connection with a remote server ORB. Typically, client applications use this property. By default, this property is not used by the application server. However, if necessary, you can specify the property for each individual application server through the administrative console.

Client applications can specify the com.ibm.CORBA.ConnectTimeout property in one of two ways:
- By including it in the orb.properties file
- By using the -CCD option to set the property with the launchclient script. The following example specifies a maximum timeout value of ten seconds:

```
launchclient clientapp.ear -CCDcom.ibm.com.CORBA.ConnectTimeout=10...
```

Begin by setting your timeout value to 20-30 seconds, but consider factors such as network congestion and application server load and capacity. Lower values can provide better failover performance, but can result in exceptions if the remote server does not have enough time to complete the connection.

Valid Range	0-300
Default	0, which means that the client ORB waits indefinitely

com.ibm.CORBA.ConnectionInterceptorName

Specifies the connection interceptor class that is used to determine the type of outbound IIOP connection to use for a request, and if secure, the quality of protection characteristics associated with the request.

WebSphere Application Server default	com.ibm.ISecurityLocalObjectBaseL13Impl.SecurityConnectionInterceptor
JDK default	None

com.ibm.CORBA.enableLocateRequest

Specifies whether the ORB uses the locate request mechanism to find objects in a WebSphere Application Server cell. Use this property for performance tuning.

When this property is set to `true`, the ORB first sends a short message to the server to find the object that it needs to access. This first contact is called the *locate request*. If most of your initial method invocations are small, setting this property to `false` might improve performance because this setting change can reduce the GIOP traffic by as much as one-half. If most of your initial method invocations are large, you should set this property to `true`. When the property is set to `true`, the small locate request message is sent instead of the large locate request message. The large message is then sent to the target after the desired object is found.

WebSphere Application Server default	true
JDK default	false

com.ibm.CORBA.ListenerPort

Specifies the port on which this server listens for incoming requests. This setting only applies for client-side ORBs.

Default	Next available system-assigned port number
Range	0 to 2147483647

com.ibm.CORBA.LocalHost

Specifies the host name or IP address of the system on which the server ORB is running. If you do not specify a value for this property, during ORB initialization, the product sets this property to the host name or IP address specified for the BOOTSTRAP_ADDRESS end point in the serverindex.xml file. For client applications, if no value is specified for this property, the ORB obtains a value at run time by calling InetAddress.getLocalHost().getHostAddress() method.

com.ibm.CORBA.RasManager

Specifies an alternative to the default RAS manager of the ORB. This property must be set to com.ibm.websphere.ras.WsOrbRasManager before the ORB can be integrated with the rest of the RAS processing for the product.

WebSphere Application Server default	com.ibm.websphere.ras.WsOrbRasManager
JDK default	None

com.ibm.CORBA.ServerSocketQueueDepth

Specifies the maximum number of connection requests that can be waiting to be handled by the Server ORB before the product starts to reject new incoming connection requests. This property corresponds to the `backlog` argument to a ServerSocket constructor and is handled directly by TCP/IP.

If you see a ″connection refused″ message in a trace log, typically, either the port on the target machine is not open, or the server is overloaded with queued-up connection requests. Increasing the value specified for this property can help alleviate this problem if there does not appear to be any other problem in the system.

Default	50
Range	From 50 to the largest value of the Java int type

com.ibm.CORBA.ShortExceptionDetails

Specifies that the exception detail message that is returned whenever the server ORB encounters a CORBA system exception contains a short description of the exception as returned by the toString method of java.lang.Throwable class. Otherwise, the message contains the complete stack trace as returned by the printStackTrace method of java.lang.Throwable class.

com.ibm.CORBA.WSSSLClientSocketFactoryName

Specifies the class that the ORB uses to create SSL sockets for secure outbound IIOP connections.

WebSphere Application Server default	com.ibm.ws.security.orbssl.WSSSLClientSocketFactoryImpl
JDK default	None

com.ibm.CORBA.WSSSLServerSocketFactoryName

Specifies the class that the ORB uses to create SSL sockets for inbound IIOP connections.

WebSphere Application Server default	com.ibm.ws.security.orbssl.WSSSLServerSocketFactoryImpl
JDK default	None

com.ibm.websphere.ObjectIDVersionCompatibility

This property applies when you have a mixed release cluster for which you are performing an incremental cell upgrade, and at least one of the releases is earlier than Version 5.1.1.

In an environment that includes mixed release cells, the migration program automatically sets this property to `1`.

After you upgrade all of the cluster members to the same release, you can remove this property from the list of ORB custom properties, or you can change the value that is specified for the property to `2`. Either action improves performance.

When this property is set to `1`, the ORB runs using version 1 object identities, which are required for mixed cells that contain application servers with releases prior to V5.1.1. If you do not specify a value for this property or if you set this property to `2`, the ORB runs using version 2 object identities, which cannot be used with pre-V5.1.1 application servers.

The *Migrating, coexisting, and interoperating* PDF includes instructions on how to perform an incremental cell upgrade.

com.ibm.ws.orb.services.redirector.MaxOpenSocketsPerEndpoint

Specifies the maximum number of connections that the IIOP Tunnel Servlet maintains in its connection cache for each target host and port. If the number of concurrent client requests to a single host and port exceeds the setting for this property, the IIOP Tunnel Servlet opens a temporary connection to the target server for each extra client request, and then closes the connection after it receives the reply. Connections that are opened, but not used within five minutes, are removed from the cache for the IIOP Tunnel Servlet.

WebSphere Application Server default	3
JDK default	Not applicable
Range	0 - largest integer recognized by Java

com.ibm.ws.orb.services.redirector.RequestTimeout

Specifies the number of seconds that the IIOP Tunnel Servlet waits for a reply from the target server on behalf of a client before timing out. If a value is not specified for this property, or is incorrectly specified, the com.ibm.CORBA.RequestTimeout property setting for the application server, on which the IIOP Tunnel Servlet is installed, is used as the setting for the com.ibm.ws.orb.services.redirector.RequestTimeout property.

The value you specify for this property must be at least as high as the highest client setting for the com.ibm.CORBA.RequestTimeout property; otherwise the IIOP Tunnel Servlet might timeout more quickly than the client typically times out while waiting for a reply. If this property is set to zero, the IIOP Tunnel Servlet does not time out.

WebSphere Application Server default	com.ibm.CORBA.RequestTimeout property setting for the application server on which the IIOP Tunnel Servlet is installed.
JDK default	Not applicable
Range	0 - largest integer recognized by Java

com.ibm.ws.orb.transport.useMultiHome

Specifies whether the server ORB binds to all network interfaces in the system. If you specify `true`, the ORB binds to all network interfaces that are available to it. If you specify `false`, the ORB only binds to the network interface that is specified for the com.ibm.CORBA.LocalHost system property.

WebSphere Application Server default	true
JDK default	true

javax.rmi.CORBA.UtilClass

Specifies the name of the Java class that the product uses to implement the javax.rmi.CORBA.UtilDelegate interface.

This property supports delegation for method implementations in the javax.rmi.CORBA.Util class. The javax.rmi.CORBA.Util class provides utility methods that can be used by stubs and ties to perform common operations. The delegate is a singleton instance of a class that implements this interface and provides a replacement implementation for all of the methods of javax.rmi.CORBA.Util. To enable a delegate, provide the class name of the delegate as the value of the javax.rmi.CORBA.UtilClass system property. The default value provides support for the com.ibm.CORBA.iiop.noLocalCopies property.

WebSphere Application Server default	com.ibm.ws.orb.WSUtilDelegateImpl
JDK default	None

Client-side programming tips for the Object Request Broker service

Every Internet InterORB Protocol (IIOP) request and response exchange consists of a client-side ORB and a server-side ORB. It is important that any application that uses IIOP is properly programmed to communicate with the client-side Object Request Broker (ORB).

The following tips should help you ensure that an application that uses IIOP to handle request and response exchanges is properly programmed to communicate with the client-side Object Request Broker (ORB).

Resolution of initial references to services

Client applications can use the ORBInitRef and ORBDefaultInitRef properties to configure the network location that the ORB service uses to find a service such as naming. When set, these properties are included in the parameters that are used to initialize the ORB, as illustrated in the following example:

```
org.omg.CORBA.ORB.init(java.lang.String[] args,
                       java.util.Properties props)
```

You can set these properties in client code or by command-line argument. It is possible to specify more than one service location by using multiple ORBInitRef property settings (one for each service), but only a single ORBDefaultInitRef value can be specified.

For setting in client code, these properties are com.ibm.CORBA.ORBInitRef.*service_name* and com.ibm.CORBA.ORBDefaultInitRef, respectively. For example, to specify that the naming service (NameService) is located in sample.server.com at port 2809, set the com.ibm.CORBA.ORBInitRef.NameService property to corbaloc::sample.server.com:2809/NameService.

For setting by command-line argument, these properties are -ORBInitRef and -ORBDefaultInitRef, respectively. To locate the same naming service specified previously, use the following Java command:

After these properties are set for services that the ORB supports, Java Platform, Enterprise Edition (Java EE) applications can call the resolve_initial_references function on the ORB, as defined in the CORBA/IIOP specification, to obtain the initial reference to a given service.

Preferred API for obtaining an ORB instance

For Java EE applications, you can use either of the following approaches. However, it is strongly recommended that you use the Java Naming and Directory Interface (JNDI) approach to ensure that the same ORB instance is used throughout the client application; you avoid the unintended inconsistencies that might occur when different ORB instances are used.

JNDI approach: For Java EE applications (including enterprise beans, Java EE clients and servlets), you can obtain an ORB instance by creating a JNDI InitialContext object and looking up the ORB under the java:comp/ORB name, as illustrated in the following example:

```
javax.naming.Context ctx = new javax.naming.InitialContext();
org.omg.CORBA.ORB orb =
   (org.omg.CORBA.ORB)javax.rmi.PortableRemoteObject.narrow(ctx.lookup("java:comp/ORB"),
                                                            org.omg.CORBA.ORB.class);
```

The ORB instance obtained using JNDI is a singleton object, shared by all the Java EE components that are running in the same Java virtual machine process.

CORBA approach: Because thin-client applications do not run in a Java EE container, they cannot use JNDI interfaces to look up the ORB. In this case, you can obtain an ORB instance by using CORBA programming interfaces, as follows:

```
java.util.Properties props = new java.util.Properties();
java.lang.String[] args = new java.lang.String[0];
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
```

In contrast to the JNDI approach, the CORBA specification requires that a new ORB instance be created each time the ORB.init method is called. If necessary to change the ORB default settings, you can add ORB property settings to the Properties object that is passed in the ORB.init method call.

The use of the com.ibm.ejs.oa.EJSORB.getORBinstance method, supported in previous releases of this product is deprecated.

API restrictions with sharing an ORB instance among Java EE application components

For performance reasons, it often makes sense to share a single ORB instance among components in a Java EE application. As required by the Java EE Specification, Version 1.3, all Web and EJB containers provide an ORB instance in the JNDI namespace as java:comp/ORB. Each container can share this instance among application components but is not required to. For proper isolation between application components, application code must comply with the following restrictions:

- Do not call the ORB shutdown or destroy methods
- Do not call org.omg.CORBA_2_3.ORB methods register_value_factory, or unregister_value_factory

In addition, do not share an ORB instance among application components in different Java EE applications.

Required use of rmic and idlj that ship with the IBM Developer Kit

The Java Runtime Environment (JRE) used by this product includes the **rmic** and **idlj** tools. You use the tools to generate Java language bindings for the CORBA/IIOP protocol.

During product installation, the tools are installed in the *app_server_root*/java/ibm_bin directory. Versions of these tools included with Java development kits in the $JAVA_HOME/bin directory other than the IBM Developer Kit installed with this product are incompatible with this product.

When you install this product, the *app_server_root*/java/ibm_bin directory is included in the $PATH search order to enable use of the rmic and idlj scripts provided by IBM. Because the scripts are in the *app_server_root*/java/ibm_bin directory instead of the JRE standard *app_server_root*/java/bin directory, it is unlikely that you can overwrite them when applying maintenance to a JRE not provided by IBM.

In addition to the rmic and idlj tools, the JRE also includes Interface Definition Language (IDL) files. The files are based on those defined by the Object Management Group (OMG) and can be used by applications that need an IDL definition of selected ORB interfaces. The files are placed in the *app_server_root*/java/ibm_lib directory.

Before using either the rmic or idlj tool, ensure that the *app_server_root*/java/ibm_bin directory is included in the proper PATH variable search order in the environment. If your application uses IDL files in the *app_server_root*/java/ibm_lib directory, also ensure that the directory is included in the PATH variable.

Character code set conversion support for the Java Object Request Broker service

The CORBA/IIOP specification defines a framework for negotiation and conversion of character code sets used by the Java Object Request Broker (ORB) service.

This product supports the framework and provides the following system properties for modifying the default settings:

com.ibm.CORBA.ORBCharEncoding

Specifies the name of the native code set that the ORB uses for character data (referred to as *NCS-C* in the CORBA/IIOP specification). By default, the ORB uses UTF8. Valid code set values for this property are shown in the table that follows this list; values that are valid only for ORBWCharDefault are indicated.

com.ibm.CORBA.ORBWCharDefault

Specifies the default code set that the ORB uses for transmission of wide character data when no code set for wide character data is found in the tagged component in the Interoperable Object Reference (IOR) or in the GIOP service context. If no code set for wide character data is found and this property is not set, the ORB raises an exception, as specified in the CORBA specification. No default value is set for this property. The only valid code set values for this property are UCS2 or UTF16.

Note: If you are using a distributed application server with a z/OS application server, you must set this property on the distributed client to UCS2 or you might experience an exception.

The CORBA code set negotiation and conversion framework specifies the use of code set registry IDs as defined in the Open Software Foundation (OSF) code set registry. The ORB translates the Java file.encoding names shown in the following table to the corresponding OSF registry IDs. These IDs are then used by the ORB in the IOR Code set tagged component and GIOP code set service context as specified in the CORBA and IIOP specification.

Java name	OSF registry ID	Comments
ASCII	0x00010020	
ISO8859_1	0x00010001	
ISO8859_2	0x00010002	
ISO8859_3	0x00010003	
ISO8859_4	0x00010004	
ISO8859_5	0x00010005	
ISO8859_6	0x00010006	
ISO8859_7	0x00010007	
ISO8859_8	0x00010008	
ISO8859_9	0x00010009	
ISO8859_15_FDIS	0x0001000F	
Cp1250	0x100204E2	
Cp1251	0x100204E3	
Cp1252	0x100204E4	
Cp1253	0x100204E5	
Cp1254	0x100204E6	
Cp1255	0x100204E7	
Cp1256	0x100204E8	
Cp1257	0x100204E9	
Cp943C	0x100203AF	
Cp943	0x100203AF	
Cp949C	0x100203B5	
Cp949	0x100203B5	
Cp1363C	0x10020553	
Cp1363	0x10020553	
Cp950	0x100203B6	
Cp1381	0x10020565	
Cp1386	0x1002056A	

Java name	OSF registry ID	Comments
EUC_JP	0x00030010	
EUC_KR	0x0004000A	
EUC_TW	0x00050010	
Cp964	0x100203C4	
Cp970	0x100203CA	
Cp1383	0x10020567	
Cp33722C	0x100283BA	
Cp33722	0x100283BA	
Cp930	0x100203A2	
Cp1047	0x10020417	
UCS2	0x00010100	Valid only for the ORBWCharDefault
UTF8	0x05010001	
UTF16	0x00010109	Valid only for the ORBWCharDefault

Object Request Brokers: Resources for learning

Use the following links to find relevant supplemental information about Object Request Brokers (ORBs). The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- “Planning, business scenarios, and IT architecture”
- “Administration”
- “Programming specifications”

Planning, business scenarios, and IT architecture

- CORBA FAQ

  Getting started with Object Request Brokers and CORBA.
- WebSphere Application Server CORBA Interoperability

  This document describes WebSphere CORBA interoperability for WebSphere Application Server products.
- CORBA Interoperability Samples

  These samples demonstrate the general principles by which WebSphere Application Server applications can interoperate with CORBA applications.

Administration

- IANA Character Set Registry

  This document contains a list of all valid character encoding schemes.
- developerWorks WebSphere

Programming specifications

- Catalog Of OMG CORBA/IIOP Specifications

  This document provides a catalog of OMG CORBA/IIOP specifications.

ORB services advanced settings on the z/OS platform

Use this topic to support Object Request Broker (ORB) service advanced settings. This support includes ORB listener keep alive, ORB Secure Sockets Layer (SSL) listener keep alive, control threads, and workload profile.

To view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* **> Container services > ORB service > z/OS additional settings**.

ORB listener keep alive

Defines the value in seconds provided to TCP/IP on the SOCK_TCP_KEEPALIVE option for the Internet Inter-ORB Protocol (IIOP) listener.

This option verifies that idle sessions are still valid by polling the client TCP/IP stack. If the client goes away without notifying the server, the session is still active on the server side. Use this property to clean up these unnecessary sessions. If the client does not respond, the session closes. The default is `0` (zero). If the property is not set, the TCP/IP option is not set. Setting the SOCK_TCP_KEEPALIVE option generates network traffic on idle sessions, which can cause problems.

Data type	Integer
Range	0 - 2147040

ORB SSL listener keep alive

This property defines the value in seconds provided to TCP/IP on the SOCK_TCP_KEEPALIVE option for the SSL IIOP listener.

This option verifies if idle sessions are still valid by polling the client TCP/IP stack. If the client goes away without notifying the server, the session is still active on the server side. Use this option to clean up these unnecessary sessions. If the client does not respond, the session closes. The default is `0` (zero). If the property is not set, the TCP/IP option is not set. Setting the SOCK_TCP_KEEPALIVE option generates network traffic on idle sessions, which can be undesirable.

Data type	Integer
Range	0 - 2147040

Workload manager timeout

Specifies the maximum time in seconds that IIOP requests are queued and dispatched to a servant process.

Data type	Integer
Range	0 - 2147040
Default	300
Disable workload manager queue timeout	0

Workload profile

Specifies the server workload profile, which can be ISOLATE, IOBOUND, CPUBOUND, LONGWAIT, or CUSTOM.

The workload profile controls workload-pertinent decisions that are made by the WebSphere Application Server for z/OS run time, such as the number of threads used in the servant. The default value is IOBOUND, which is the appropriate value for most applications. Use one of the other values when your application requires more threads.

Workload profile	Number of Threads	Description

ISOLATE	1	Specifies that the servants are restricted to a single application thread. Use ISOLATE to ensure that concurrently dispatched applications do not run in the same servant. Two requests processed in the same servant can cause one request to corrupt another.
IOBOUND	MIN(30, MAX(5,(Number of CPUs*3)))	Specifies more threads in applications that perform I/O-intensive processing on the z/OS operating system. The calculation of the thread number is based on the number of CPUs. IOBOUND is used by most applications that have a balance of CPU intensive and remote operation calls. A gateway or protocol converter are two examples of applications that use the IOBOUND profile.
CPUBOUND	MAX((Number of CPUs-1),3)	Specifies that the application performs processor-intensive operations on the z/OS operating system, and therefore would not benefit from more threads than the number of CPUs. The calculation of the thread number is based on the number of CPUs. Use the CPUBOUND profile setting in CPU intensive applications, like XML parsing and XML document construction, where the vast majority of the application response time is spent using the CPU.
LONGWAIT	40	Specifies more threads than IOBOUND for application processing. LONGWAIT spends most of its time waiting for network or remote operations to complete. Use this setting when the application makes frequent calls to another application system, like Customer Information Control System (CICS) screen scraper applications, but does not do much of its own processing.
CUSTOM	User defined	Specifies that the number of servant application threads is determined by the value that is specified for the servant_region_custom_thread_count server custom property. The minimum number of application threads that can be defined for this custom property is 1; the maximum number of application threads that can be specified is 100.

Note: Number of CPUs is the number of CPUs online when the controller comes up.

You can look at message BBOO0234I in the controller job log to check the number of worker threads.

Object request broker component troubleshooting tips

The following topics might help you diagnose problems with the Object request broker (ORB) component.

Enabling tracing for the Object Request Broker component

The object request broker (ORB) service is one of the product run time services. Tracing messages that are sent and received by the ORB is a useful starting point for troubleshooting the ORB service. You can selectively enable or disable tracing of ORB messages for each server in a product installation, and for each application client.

For instructions on how to set trace controls so that tracing occurs for the ORB subcomponent, see Setting trace controls for IBM service.

Log files and messages associated with Object Request Broker

For a summary of how messages get routed in the product, see Message routing.

Java packages containing the Object Request Broker service

The ORB service resides in the following Java packages:

com.ibm.CORBA.*
: This package provides the mapping of the IBM CORBA APIs to the Java programming language, including the class ORB.

com.ibm.rmi.*
: This package provides the IBM Remote Method Invocation (RMI) APIs that are used to establish remote communication between programs written in the Java programming language.

com.ibm.ws.orb.*
: This package provides the APIs that are used to specify configuration settings for the ORB.

com.ibm.ws.orbimpl.*
: This package provides the IBM implementation classes for the ORB.

com.ibm.ws390.orb.*
: This package provides z/OS-only classes.

com.ibm.ws390.channel.ziop.*
: This package contains portability APIs for z/OS IIOP transport channels.

com.ibm.ws390.ziop.*
: This package contains portability APIs for the z/OS IIOP runtime.

org.omg.CORBA.*
: This package provides the mapping of the OMG CORBA APIs to the Java programming language, including the class ORB.

javax.rmi.CORBA.*
: This package contains portability APIs for the RMI-IIOP runtime.

JAR files that contain the previously mentioned packages include:

- *app_server_root*/java/lib/ibmorb.jar, which contains the base ORB classes com.ibm.CORBA.*, com.ibm.rmi.*, javax.rmi.CORBA.*, and org.omg.CORBA.*
- *app_server_root*/plugins/com.ibm.ws.runtime.jar, which contains the extension classes com.ibm.ws.orb.*, com.ibm.ws.orbimpl.*, and com.ibm.CORBA.services.*.
- *app_server_root*/plugins/com.ibm.ws.runtime.ws390.jar, which contains the com.ibm.ws390.* classes.
- *app_server_root*/lib/bootstrap.jar and *app_server_root*/lib/bootstrapws390.jar, which include some ORB related classes.

Tools used with Object Request Broker

The tools used to compile Java remote interfaces to generate language bindings used by the ORB at runtime reside in the following APIs:
- com.ibm.tools.rmic.*
- com.ibm.idl.*

The JAR file that contains these APIs is *app_server_root*/java/lib/tools.jar.

Object Request Broker properties

The ORB service requires a number of ORB properties for correct operation. It is not necessary for most users to modify these properties, and only the system administrator should modify them when required. Consult IBM Support personnel for assistance. The properties reside in the properties file, located at *app_server_root*/properties/orb.properties.

If none of these steps fixes your problem, check the Support page to see if the problem has been identified and documented. The Support page contains hints and tips, technotes, and descriptions of available fixes.

Before opening a problem report, collect the information, described in Troubleshooting help from IBM, that Support needs to resolve problems. The Support page includes documents and tools that can help you gather this information.

Enabling HTTP tunneling

HTTP tunneling enables clients, that reside outside of a firewall, to bundle all of the information, that the client-side Object Request Broker (ORB) needs to send to the server-side ORB, into a normal HTTP request. This request can then be sent to the server on port 80, just like any other HTTP request.

Before you begin

Make sure that the client-side ORB is an IBM ORB. Tunneling does not work if you are using a non-IBM ORB on the client.

Also, if Secure Sockets Layer (SSL) security is required for the tunneling, make sure that the required certificates and key files are configured.

About this task

Sometimes clients residing outside of a firewall need to communicate with modules, such as EJB modules, that reside on a server inside of the firewall. The client-side and server-side ORBs manage this interaction between the client and the server. However, firewalls normally block the ports that a client, uses to talk to the server-side ORB. Therefore if your installation uses a firewall that blocks the ports a client uses to talk to the server-side ORB, you should set up HTTP tunneling.

The IIOPTunnelServlet, which is shipped with the product as class file com.ibm.CORBA.services.IIOPTunnelServlet.class, allows an HTTP client, such as a Java client, that is embedded with RMI-IIOP, to communicate with a server that resides inside of a firewall. This class file, along with the following three class files, are bundled within the WAS_HOME/plugins/com.ibm.ws.runtime_6.1.0.jar file. These additional class files enhance the servlet's capabilities.

com.ibm.CORBA.services.redirector.ConnectionStream.class
com.ibm.CORBA.services.redirector.Redirector.class
com.ibm.CORBA.services.redirector.RedirectorController.class

When tunneling is enabled, the IIOPTunnelServlet servlet on the server receives the HTTP request and unpacks all of the ORB information. The servlet then calls the server-side ORB on the client's behalf. The server-side ORB treats the request as it would treat any normal ORB request and responds to the servlet. The servlet packs the ORB response into an HTTP response and sends the response back to the client-side ORB, through the firewall. The client-side ORB unpacks the HTTP response and pulls out the response.

Tunneling can operate over HTTPS as well as over HTTP. Therefore, you can use Secure Sockets Layer (SSL) security to secure your tunneling clients if your security procedures require that all communication to your servers is SSL secured.

1. Create an installable IIOPTunnel.ear file that includes the IIOPTunnelServlet servlet.

   Before you can run the IIOPTunnelServlet servlet on the server, you must make it part of an application that you can install on the server. You can use an application assembly tool to create an installable IIOPTunnel.ear file that includes this servlet. For example, if you use the assembly tool that is shipped with the product:

   a. Start the tool.
   b. Open the WEB perspective.
   c. In the Project Explorer view, right click in an empty pane and select **New > Dynamic Web Project**.
   d. In the Create Dynamic Web Project wizard, change the project Name to IIOPTunnel, or another name that is meaningful to you. By default, the **Add Module to an EAR project** option is selected, the EAR project name is set to IIOPTunnelEAR, and the Context Root is set to IIOPTunnel.
   e. Keep these default settings and click **Finish**.
   f. Add the com.ibm.ws.runtime_7.0.0.jar file to the Web Project Build Path.

      Before you can register the new servlet in the Web Deployment Descriptor, you must add the IIOPTunnelServlet servlet, that resides in the WAS_HOME/lib/plugins/com.ibm.ws.runtime_7.0.0.jar file, to your build path.

      1) Right click the IIOPTunnel Web Project, and select **Properties > Java Build Path**.
      2) Select the Libraries tab and press the Add external JARs button.
      3) Add the com.ibm.ws.runtime_7.0.0.jar file, and then click **OK**.

2. Export your EAR file.

   a. Right click on the IIOPTunnelEAR project.
   b. Click **Export > EAR File**, browse to your selected destination directory and specify the EAR file name as IIOPTunnel.ear, or the file name that you specified in Step 1d.
   c. Click **Finish**.

      You get your IIOPTunnel.ear file, which is ready for you to deploy.

3. Install the IIOPTunnel.ear file on your target application server. You can accept all default values during installation.

   Remember to adjust the tunnelAgentURL in the client to reflect the actual location of the IIOPTunnelServlet on your server.

   Note:

   `http(s)://host_name:port/context_root/Servlet_URLmapping`

   The*host_name:port* are the host name and port that is assigned to teh server on which the IIOPTunnelServlet resides. The port can be either an HTTP or an HTTPS port, depending on your security requirements.

   The *context_root* and *Servlet_URLmapping* values must match the values that are defined for the context-root and servlet-URLmapping elements in the servlet web.xml file.

   For example, if the servlet is installed on the default server, and context-root=iioptunnel, and Servlet-URLmapping=tunnel, the following URL must be specified for tunnelAgentURL in the client:

   `http://localhost:9080/IIOPTunnel/IIOPTunnelServlet`

To verify that the servlet is deployed and running successfully, you can open a browser and point to http:// hostname:9080/iioptunnel/tunnel. If the servlet is working, the browser tries to download the servlet as if it were just a normal file. You can then cancel the download.

4. Verify that the servlet is deployed and running successfully

   To verify that the servlet is deployed and running successfully, you can open a browser and point to http:// hostname:9080/IIOPTunnel/IIOPTunnelServlet. If the servlet is working, the browser tries o download the servlet as if it were just a normal file. Simply cancel the download.

   Specify the following parameters if you encounter a problem deploying and running the servlet.

   ```
   -Dcom.ibm.CORBA.TunnelAgentURL=https://localhost:9080/IIOPTunnel/IIOPTunnelServlet?debug=true
   ```

5. Configure the ORB Service for the client-side ORB to enable tunneling

   The client determines whether standard IIOP and HTTP tunneling should be used for communication with the server-side ORB. Therefore you must set the following ORB properties on the client.

   ```
   com.ibm.CORBA.ForceTunnel=ALWAYS
   com.ibm.CORBA.TunnelAgentURL=http://host_name:9080/IIOPTunnel/IIOPTunnelServlet com.ibm.CORBA.FragmentSize=0
   ```

   To enabled tunneling on the client ORB, the com.ibm.CORBA.ForceTunnel property must be set to **ALWAYS**. This setting indicates that this client is always going to tunnel. Other values that can be specified for the com.ibm.CORBA.ForceTunnel property are:

   NEVER, which indicates that you want to disable HTTP tunneling. If a TCP connection fails, a CORBA system exception (COMM_FAILURE) occurs.

   WHENREQUIRED, which indicates that you want to use HTTP tunneling if TCP connections fail.

   The second property specifies the fully qualified URL at which the tunneling servlet is reached. The port 9080 is the WC_defaulthost port for the server. The port number you specify must match the port number that is specified in the configuration file, serverindex.xml, for the server on which the IIOPTunnelServlet servlet resides.

   The third property turns off ORB fragmenting. Normally, the ORB breaks up communications into fragments, to improve performance, but tunneling will not work if the ORB is fragmenting.

   You can also set these properties by adding them as parameters to the JVM command line:

   ```
   -Dcom.ibm.CORBA.ForceTunnel=always
   -Dcom.ibm.CORBA.TunnelAgentURL=http://host_name:9080/iioptunnel/tunnel
   -Dcom.ibm.CORBA.FragmentSize=0
   ```

   Optionally, you can also set the following property if you want to specify client-side security settings:

   ```
   -Dcom.ibm.CORBA.ConfigURL=file:PROFILE_ROOT/properties/sas.client.props
   ```

6. Turn off fragmenting on the server-side ORB. The only property that you must configure for the server-side ORB to enable tunneling is the com.ibm.CORBA.FragmentSize property. This property must be set to `0` to turn off fragmenting.
   a. In the administrative console, click **Servers > Server Types > WebSphere application servers**, and click the server where the tunneling servlet is installed.
   b. Click **ORB Service**, then click **Custom properties**.
   c. Click **New** and then specify **com.ibm.CORBA.FragmentSize** in the Name field and **0** in the Value field.
   d. Click **OK**, and then save the changes.
7. Stop and then restart the application server.

What to do next

The client can start to sent requests through the firewall to the server that is configured for HTTP tunneling.

Chapter 14. Transactions

Using the transaction service

WebSphere Application Server applications can use transactions to coordinate multiple updates to resources as atomic units (as indivisible units of work) such that all or none of the updates are made permanent.

About this task

In WebSphere Application Server, transactions are managed by three main components:

- A transaction manager. The transaction manager supports the enlistment of recoverable XAResources and ensures that each resource of this type is driven to a consistent outcome either at the end of a transaction or after a failure and restart of the application server.

  Also, WebSphere Application Server for z/OS supports the coordination of resource managers through RRS (z/OS resource recovery services).
- A container in which the enterprise application runs. The container manages the enlistment of XAResources on behalf of the application when the application performs updates to transactional resource managers (for example, databases). Optionally, the container can control the demarcation of transactions for enterprise beans configured for container-managed transactions.
- An application programming interface, UserTransaction, that is available to bean-managed enterprise beans and servlets. These application components can use the UserTransaction interface to control the demarcation of their own transactions.

  For details about the methods available with the UserTransaction interface, see the Additional Application Programming Interfaces (APIs) or the Java Transaction API (JTA) 1.1 Specification.

  Also, Java Transaction API (JTA) support includes additional application programming interfaces so that application frameworks can manipulate the unit of work (UOW) context of a thread, and components can register with a JTA transaction (for example, a persistence manager can be notified of transaction completion).

Use the following tasks to work with transactions in WebSphere Application Server applications:

- “Developing components to use transactions” on page 1482
- Configuring transaction properties for an application server
- Configuring transaction properties for peer recovery
- Managing manual peer recovery of the transaction service
- Managing active and prepared transactions
- “Managing active and prepared transactions using scripting” on page 1486
- Interoperating transactionally between application servers
- Using WS-Transaction policy to coordinate transactions or business activities for Web services
- Troubleshooting transactions
- “Using one-phase and two-phase commit resources in the same transaction” on page 1489
- “Using the ActivitySession service” on page 1495

Transaction support in WebSphere Application Server

This topic provides conceptual information about the support for transactions provided by the Transaction Service of WebSphere Application Server.

A transaction is unit of activity, within which multiple updates to resources can be made atomic (as an indivisible unit of work) such that all or none of the updates are made permanent. For example, during the processing of an SQL COMMIT statement, the database manager atomically commits multiple SQL statements to a relational database. In this case, the transaction is contained entirely within the database

manager and can be thought of as a *resource manager local transaction (RMLT)*. In some contexts, a transaction is referred to as a *logical unit of work* (LUW). If a transaction involves multiple resource managers, for example multiple database managers, an external transaction manager is required to coordinate the individual resource managers. A transaction that spans multiple resource managers is referred to as a *global transaction*. WebSphere Application Server is a transaction manager that can coordinate global transactions, can be a participant in a received global transaction, and can also provide an environment in which resource manager local transactions can run.

The way that applications use transactions depends on the type of application component, as follows:
- A session bean can use either container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (component-managed transactions where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) and application client components use component-managed transactions.

WebSphere Application Server is a transaction manager that supports the coordination of resource managers through their XAResource interface, and participates in distributed global transactions with transaction managers that support the CORBA Object Transaction Service (OTS) protocol or Web Service Atomic Transaction (WS-AtomicTransaction) protocol. WebSphere Application Server also participates in transactions imported through Java EE Connector 1.5 resource adapters. You can also configure WebSphere applications to interact with databases, JMS queues, and JCA connectors through their *local transaction* support, when you do not require distributed transaction coordination.

In addition to supporting the coordination of XAResource-based resource managers, WebSphere Application Server for z/OS supports the coordination of resource managers through RRS (z/OS resource recovery services). RRS-compliant resource managers include DB2, WebSphere MQ, IMS, and CICS. IBM WebSphere Application Server for z/OS can coordinate a mix of RRSTransactional resource managers and XA capable resource managers under the same global transaction.

Resource managers that offer transaction support can be categorized into those that support two-phase coordination (by offering an XAResource interface or by supporting RRS) and those that support only one-phase coordination (for example through a LocalTransaction interface). The WebSphere Application Server transaction support provides coordination, within a transaction, for any number of two-phase capable resource managers. It also enables a single one-phase capable resource manager to be used within a transaction in the absence of any other resource managers, although a WebSphere transaction is not necessary in this case.

Under normal circumstances, you cannot mix one-phase commit capable resources and two-phase commit capable resources in the same global transaction, because one-phase commit resources cannot support the prepare phase of two-phase commit. There are some special circumstances where it is possible to include mixed-capability resources in the same global transaction:
- In scenarios where there is only a single one-phase commit resource provider that participates in the transaction and where all the two-phase commit resource-providers that participate in the transaction are used in a read-only fashion. In this case, the two-phase commit resources all vote read-only during the prepare phase of two-phase commit. Because the one-phase commit resource provider is the only provider to actually perform any updates, the one-phase commit resource does not need to be prepared.
- In scenarios where there is only a single one-phase commit resource provider that participates in the transaction with one or more two-phase commit resource providers and where *last participant support* is enabled. Last participant support enables the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction. For more information about last participant support, see Using one-phase and two-phase commit resources in the same transaction.

The ActivitySession service provides an alternative unit-of-work (UOW) scope to that provided by global transaction contexts. It is a distributed context that can be used to coordinate multiple one-phase resource managers. The WebSphere EJB container and deployment tooling support ActivitySessions as an extension to the Java EE programming model. Enterprise beans can be deployed with lifecycles that are influenced by ActivitySession context, as an alternative to transaction context. An application can then interact with a resource manager for the period of a client-scoped ActivitySession, rather than only the duration of an EJB method, and have the resource manager's local transaction outcome directed by the ActivitySession. For more information about ActivitySessions, see Using the ActivitySession service.

You can use transaction classes to classify client workload for workload management. The workload is different WebSphere transactions targeted to separate servant regions, each with goals defined by appropriate service classes. Each transaction is dispatched in its own WLM enclave in a servant region process, and is managed according to the goals of its service class. The server controller, which workload management views as a queue manager, uses the enclave associated with a client request to manage the priority of the work. If the work has a high priority, workload management can direct the work to a high-priority servant in the server. If the work has a low priority, workload management can direct the work to a low-priority servant. The effect is to partition the work according to priority within the same server.

Resource manager local transaction (RMLT)

A resource manager local transaction (RMLT) is a resource manager's view of a local transaction; that is, it represents a unit of recovery on a single connection that is managed by the resource manager.

Resource managers include:

- Enterprise Information Systems that are accessed through a resource adapter, as described in the Java EE Connector Architecture.
- Relational databases that are accessed through a JDBC datasource.
- JMS queue and topic destinations.

Resource managers offer specific interfaces to enable control of their RMLTs. Resource adapter components of the Java EE connector architecture that include support for local transactions provide a LocalTransaction interface. The LocalTransaction interface enables applications to request that the resource adapter commits or rolls back RMLTs. JDBC datasources provide a Connection interface for the same purpose.

The boundary at which all RMLTs must be complete is defined in WebSphere Application Server by a local transaction containment (LTC).

Global transactions

If an application uses two or more resources, an external transaction manager is needed to coordinate the updates to all the resource managers in a global transaction.

Global transaction support is available to Web and enterprise bean components and, with some limitations, to application client components. Enterprise bean components can be subdivided into two categories: beans that use container-managed transactions (CMT) and beans that use bean-managed transactions (BMT).

BMT enterprise beans, application client components, and Web components can use the Java Transaction API (JTA) UserTransaction interface to define the demarcation of a global transaction. To obtain the UserTransaction interface, use a Java Naming and Directory Interface (JNDI) lookup of java:comp/UserTransaction, or use the getUserTransaction method from the SessionContext object.

The UserTransaction interface is not available to CMT enterprise beans. If CMT enterprise beans attempt to obtain this interface, an exception is thrown, in accordance with the Enterprise JavaBeans (EJB) specification.

Ensure that programs that perform a JNDI lookup of the UserTransaction interface use an InitialContext that resolves to a local implementation of the interface. Also ensure that such programs use a JNDI location that is appropriate for the EJB version.

WebSphere Application Server Version 4 and later releases bind the UserTransaction interface at the JNDI location that is specified in the EJB Version 1.1 specification. This location is java:comp/UserTransaction.

A Web component or enterprise bean (CMT or BMT) can use additional interfaces that provide JTA support. These interfaces provide the transaction identity and a mechanism to receive notification of transaction completion. The interfaces include the TransactionSynchronizationRegistry interface, the ExtendedJTATransaction interface, and the UOWSynchronizationRegistry interface.

Local transaction containment (LTC)

A *local transaction containment (LTC)* is used to define the application server behavior in an unspecified transaction context.

Unspecified transaction context is defined in the Enterprise JavaBeans specification, Version 2.0 and later. For example, see Enterprise JavaBeans Specification.

An LTC is a bounded unit-of-work scope, within which zero, one, or more resource manager local transactions (RMLT) can be accessed. The LTC defines the boundary at which all RMLTs must be complete; any incomplete RMLTs are resolved, according to policy, by the container. By default, an LTC is local to a bean instance; it is not shared across beans, even if those beans are managed by the same container. LTCs are started by the container before dispatching a method on an enterprise application component (such as an enterprise bean or servlet) whenever the dispatch occurs in the absence of a global transaction context. LTCs are completed by the container depending on the application-configured LTC boundary, for example, at the end of the method dispatch. There is no programmatic interface to the LTC support; LTCs are managed exclusively by the container. The application deployer configures LTCs on individual application components (Web application or EJB) by using transaction attributes in the application deployment descriptor.

Note: In this release, a local transaction containment (LTC) is shareable, that is, a single LTC can span multiple application components, including Web application components and enterprise beans that use container-managed transactions, so that those components can share connections without using a global transaction. Sharing a single resource manager between application components improves performance, increases scalability, and reduces lock contention for resources.

LTCs can be shared across multiple components, including Web application components and enterprise beans that use container-managed transactions. This is useful in situations such as when there is frequent use of Web module include calls, where a thread can have several connections blocked by LTCs in different Web modules. In this situation, the application might encounter code deadlocks under load, when threads start to wait for themselves to free connections. To overcome this issue without using a global transaction, specify that application components can share LTCs by setting the Shareable attribute in the deployment descriptor of each component. You must use a deployment descriptor; you cannot specify this attribute if annotation has been used.

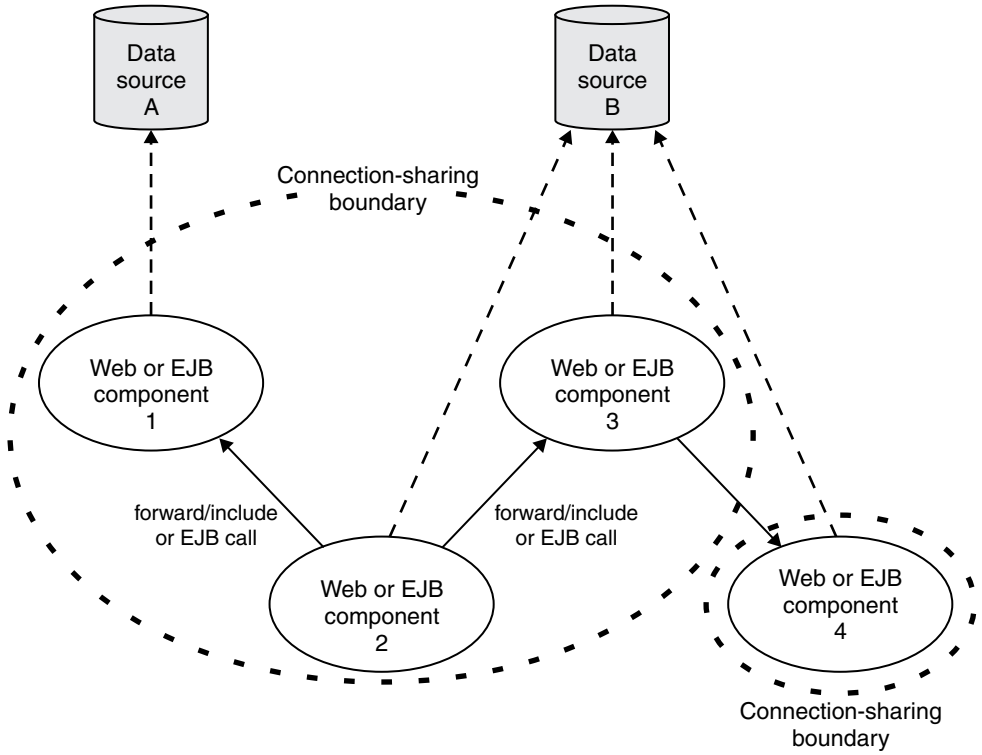
When you set the Shareable attribute, the extended deployment descriptor XML file includes the following line:

```
<local-transaction boundary="BEAN_METHOD" resolver="CONTAINER_AT_BOUNDARY"
unresolved-action="COMMIT" shareable="true"/>
```

To obtain the full benefits of a shared LTC, also ensure that the resource reference for each component defaults to shareable connections.

In the following diagram, components 1, 2 and 3 are deployed with the Shareable attribute and component 4 is not. If components 2 and 3 both obtain connections to data source B, and their resource references

for data source B default to shareable connections, they share the connection, but component 4 does not.



Applications that use shareable LTCs cannot explicitly commit or roll back resource manager connections that are used in a shareable LTC (although they can use connections that have an autoCommit capability). This ensures proper encapsulation of connection usage by each component and protects one component from having to make any assumptions about the behavior of other components that share the connection.

If an application starts any non-autocommit work in an LTC for which the Resolver attribute is set to Application and the Shareable attribute is set to true, an exception is thrown at run time. For example, on a JDBC Connection, non-autocommit work is work that the application performs after using the setAutoCommit(false) method to switch off the autocommit option on the connection. Enterprise beans that use bean managed transactions (BMT) cannot be assembled with the Shareable attribute set on the LTC configuration.

A local transaction containment cannot exist concurrently with a global transaction. If application component dispatch occurs in the absence of a global transaction, the container always establishes an LTC. The only exceptions to this behavior is when an application component dispatch occurs without container interposition; for example, for a stateless session bean `create`.

A local transaction containment can be scoped to an ActivitySession context that exists longer than the enterprise bean method in which it is started, as described in ActivitySessions and transaction contexts.

Local and global transaction considerations

Applications use resources, such as Java Database Connectivity (JDBC) data sources or connection factories, that are configured through the Resources view of the administrative console. How these resources participate in a global transaction depends on the underlying transaction support of the resource provider.

For example, most JDBC providers can provide either XA or non-XA versions of a data source. A non-XA data source can support only resource manager local transactions (RMLT), but an XA data source can support two-phase commit coordination, as well as local transactions.

Additionally, some JDBC Providers support the use of z/OS Resource Recovery Service (RRS) to coordinate transaction processing. This type of JDBC Provider is RRSTransactional. When RRS is used, both local and global transactions are supported.

If an application uses two or more resource providers that support only RMLTs, atomicity cannot be assured because of the one-phase nature of these resources. To ensure atomic behavior, the application should use resources that support XA coordination or RRS coordination and should access them within a global transaction.

If an application uses only one RMLT, atomic behavior can be guaranteed by the resource manager, which can be accessed in a local transaction containment (LTC) context.

An application can also access a single resource manager in a global transaction context, even if that resource manager does not support the XA coordination. An application can do this because the application server performs an “only resource optimization” and interacts with the resource manager in a RMLT. In a global transaction context, any attempt to use more than one resource provider that supports only RMLTs causes the global transaction to be rolled back.

At any moment, an instance of an enterprise bean can have work outstanding in either a global transaction context or a local transaction containment context, but not both. An instance of an enterprise bean can change from running in one type of context to the other (in either direction), if all outstanding work in the original context is complete. Any violation of this principle causes an exception to be thrown when the enterprise bean tries to start the new context.

Client support for transactions

This topic describes the support of application clients for the use of transactions.

Application clients running in an enterprise application client container can explicitly demarcate transaction boundaries, as described in Using component-managed transactions. Application clients cannot perform, directly in the client container, transactional work in the context of any global transaction that they start, because the client container is not a recoverable process.

Application clients can make requests to remote objects, such as enterprise beans, in the context of a client-initiated transaction. Any transactional work performed in a remote, recoverable, server process is coordinated as part of the client-initiated transaction. The transaction coordinator is created on the first server process to which the client-initiated transaction is propagated.

A client can begin a transaction, then, for example, access a JDBC data source directly in the client process. In such cases, any work performed through the JDBC provider is not coordinated as part of the global transaction. Instead, the work runs under a resource manager local transaction. The client container process is non-recoverable and contains no transaction coordinator with which a resource manager can be enlisted.

A client can begin a transaction, then call a remote application component such as an enterprise bean. In such cases, the client-initiated transaction context is implicitly propagated to the remote application server, where a transaction coordinator is created. Any resource managers accessed on the recoverable application server (or any other application server hosting application components invoked by the client) are enlisted in the global transaction.

Client application components need to be aware that locally-accessed resource managers are not coordinated by client-initiated transactions. Client applications acknowledge this through a deployment option that enables access to the UserTransaction interface in the client container. By default, access to

the UserTransaction interface in the client container is not enabled. To enable UserTransaction demarcation for an application client component, set the **Allow JTA Demarcation** extension property in the client deployment descriptor. For information about editing the client deployment descriptor, refer to the Rational Application Developer information.

Commit priority for transactional resources

You can specify the order in which transactional resources are processed during two-phase commit processing.

Note: This release introduces resource commit ordering, where you control the order in which transactional resources are processed during two-phase commit processing. Resource commit ordering can increase the occurrence of one-phase commits and therefore improve performance, and reduce problems caused by transaction isolation.

If you control the order in which transactional resources are processed during two-phase commit processing, there are two main benefits:

- One-phase commit optimization occurs more often.
- Potential problems caused by transaction isolation are resolved.

To control the order in which transactional resources are processed during two-phase commit processing, you specify the commit priority of a resource by setting the commit priority attribute on a resource reference. The larger the commit priority, the earlier the resource is processed. For example, if a resource has a commit priority of 10, it is processed before a resource with a commit priority of 1. The commit priority value is of type int and can be between -2147483648 and 2147483647.

If you do not specify a commit priority value, a default value of zero is assigned to the resource and is used when ordering resources at run time. If two or more resources are configured with the same priority, including the default priority, they are processed in an unspecified order with respect to each other.

You can specify the commit priority attribute on a resource reference by using Rational Application Developer tools. For detailed information, see the Rational Application Developer information center. The application component must have a deployment descriptor; you cannot specify this attribute if annotation has been used.

One-phase commit optimization

In a transaction with a two-phase commit, if every resource except the last one enlisted in the transaction votes read-only, indicating that those resources are not interested in the outcome of the transaction, a one-phase commit can occur. This means that the transaction service does not need to store resource and transaction information that it would need to roll back a two-phase commit, and therefore performance is improved.

You can control the order in which transactional resources are processed during two-phase commit, so you can process the resources that are most likely to vote read-only first. Therefore, you increase the chance that a one-phase commit might occur.

Typically, for a given transactional resource, you know the work that is performed at run time, so if you can control the order in which the resources in a transaction are processed, you can increase the likelihood of a one-phase commit optimization occurring.

Transaction isolation

When resources are involved in a global transaction, updates that are made as part of a transaction are not visible outside the transaction until the transaction commits, that is, those resources are isolated. This isolation can cause problems with other application components that act on the updates after they are committed. For example, further processing can fail, or can fail intermittently, because updates are order

and time dependent. This problem does not occur with service integration bus messaging work in WebSphere Application Server, but can be a problem for other messaging providers, for example WebSphere MQ.

If you specify the order in which transactional resources are committed, problems caused by isolation are resolved for all transactional systems, not just messaging providers and service integration bus in particular.

The following example describes how problems might occur when you cannot specify the order in which transactional resources are committed. An application updates a row in a database table, then sends a JMS message that triggers additional processing of the row. Both of these actions are performed in the same global transaction, so they are isolated until their respective resources are committed. If the update to the row is committed before the message is sent, the processing that is triggered by the message can access the updated row and process it. If the action to send the message is committed first, this action might trigger the additional processing of the row before the database has committed the update to the row. In this situation, the updated row is still isolated and is not visible, so the additional processing of the row fails.

This problem can be more complicated because it is ordering and timing dependent. If the database is committed first, the problem does not occur. If the action to send the message is committed first, the problem might occur, but it depends whether the database work is committed before the message triggers the further processing of the row. Therefore, the problem can be intermittent, so it is harder to identify its cause.

Considerations with earlier versions of WebSphere Application Server

If you specify the commit priority of a resource, that is, specify any value other than the default value 0, the commit priority is added to the partner log in a recoverable unit section. If you use WebSphere Application Server with versions earlier than WebSphere Application Server Version 7.0, you need to install the appropriate fix pack to the earlier version to ensure that the new section in the log file is recognized. However, commit priority has no effect on the transaction service in versions of WebSphere Application Server that are earlier than Version 7.0.

For general information about different versions of the product, see the topic “Overview of migration, coexistence, and interoperability”.

Transactional high availability

The high availability of the transaction service enables any server in a cluster to recover the transactional work for any other server in the same cluster. This facility forms part of the overall WebSphere Application Server high availability (HA) strategy.

This feature is in addition to the support for Peer restart and recovery, which enables you to restart on a peer system in the sysplex.

As a vital part of providing recovery for transactions, the transaction service logs information about active transactional work in the *transaction recovery log*. The transaction recovery log stores the information in a persistent form, which means that any transactional work in progress at the time of a server failure can be resolved when the server is restarted. This activity is known as *transaction recovery processing*. In addition to completing outstanding transactions, this processing also ensures that any locks held in the associated resource managers are released.

Peer recovery processing

The standard recovery process that is performed when an application server restarts is for the server to retrieve and process the logged transaction information, recover transactional work and complete in-doubt transactions. Completion of the transactional work (and hence the release of any database locks held by

the transactions) takes place after the server successfully restarts and processes its transaction logs. If the server is slow to recover or requires manual intervention, the transactional work cannot be completed and access to associated databases is disrupted.

To minimize such disruption to transactional work and the associated databases, WebSphere Application Server provides a high availability strategy known as *transaction peer recovery*.

Peer recovery is provided within a server cluster. A peer server (another cluster member) can process the recovery logs of a failed server while the peer continues to manage its own transactional workload. You do not have to wait for the failed server to restart, or start a new application server specifically to recover the failed server.

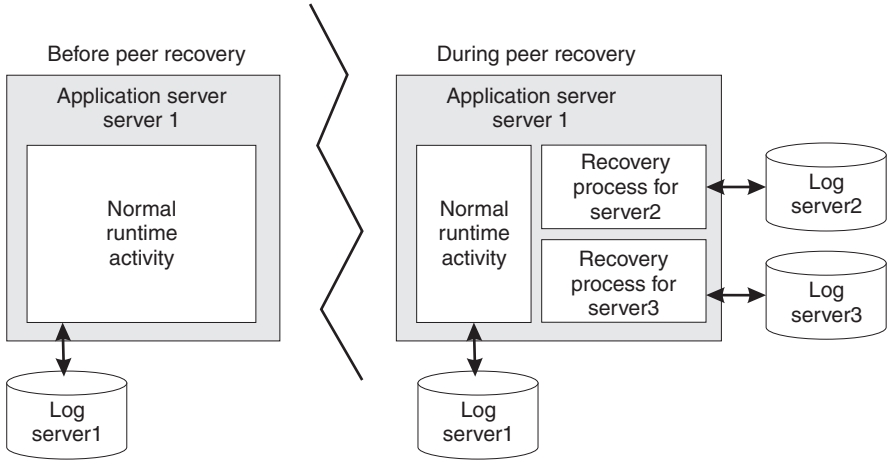


Figure 10. Peer recovery

The peer recovery process is the logical equivalent to restarting the failed server, but does not constitute a complete restart of the failed server within the peer server. The peer recovery process provides an opportunity to complete outstanding work; it cannot start new work beyond recovery processing. No forward processing is possible for the failed server.

Peer recovery moves the high availability requirements away from individual servers and onto the server cluster. After such failures, the management system of the cluster dispatches new work onto the remaining servers; the only difference is the potential drop in overall system throughput. If a server fails, all that is required is to complete work that was active on the failed server and redirect requests to an alternate server.

By default, peer recovery is disabled until you enable failover of transaction log recovery in the cluster configuration, and restart the cluster members. After you enable transaction log recovery, WebSphere Application Server supports two styles for the initiation of transaction peer recovery: automated and manual. You determine which style is more appropriate, based on your deployment considerations, and specify that style by configuring the appropriate high availability policy. This high availability policy is referred to elsewhere in these topics as the *policy for the transaction service*.

Automated peer recovery

This style is the default for peer recovery initiation. If an application server fails, WebSphere Application Server automatically selects a server to perform peer recovery processing on its behalf, and passes recovery back to the failed server when it restarts. To use this model, all you have to do is enable transaction log recovery and configure the recovery log location for each cluster member.

Manual peer recovery

You must explicitly configure this style of peer recovery. If an application server fails, you use the administrative console to select a server to perform recovery processing on its behalf.

Peer recovery example

The following diagrams illustrate the peer recovery process that takes place if a single server fails. Figure 2 shows three stable servers running in a WebSphere Application Server cluster. The workload is balanced between these servers, which results in locks held by the back-end database on behalf of each server.

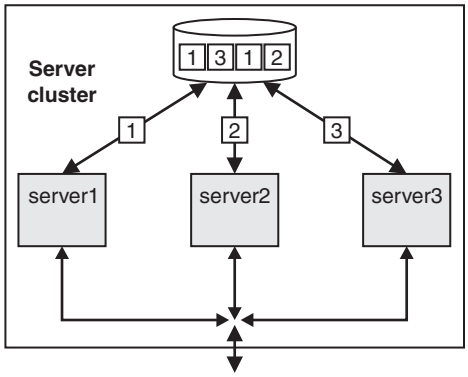


Figure 11. Server cluster up and running, just before server failure

Figure 3 shows the state of the system after server 1 fails without clearing locks from the database. Servers 2 and 3 can run their existing transactions to completion and release existing locks in the back-end database, but further access might be impaired because of the locks still held on behalf of server 1. In practice, some level of access by servers 2 and 3 is still possible, assuming appropriately configured lock granularity, but for this example assume that servers 2 and 3 attempt to access locked records and become blocked.

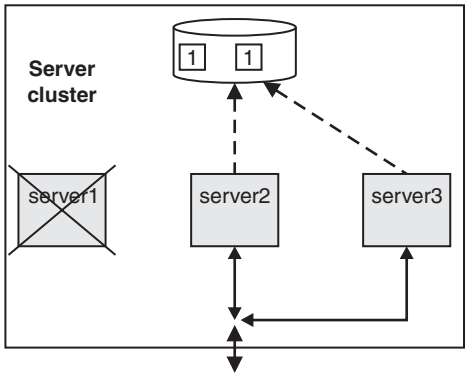


Figure 12. Server 1 fails. Servers 2 and 3 become blocked as a result.

Figure 4 shows a peer recovery process for server 1 running inside server 3. The transaction service portion of the recovery process retrieves the information that is persisted by server 1, and uses that information to complete any in-doubt transactions. In this figure, the peer recovery process is partially complete as some locks are still held by the database on behalf of server 1.

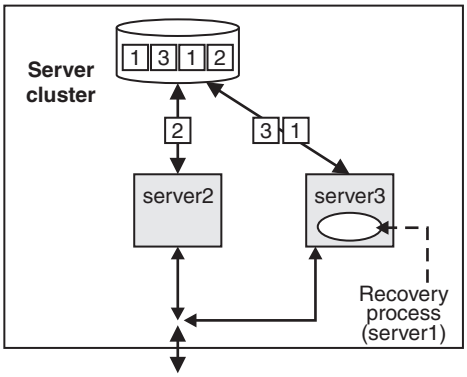


Figure 13. Peer recovery process started in server 3

Figure 5 shows the state of the server cluster when the peer recovery process is complete. The system is in a stable state with just two servers, between which the workload is balanced. Server 1 can be restarted, and will have no recovery processing of its own to perform.

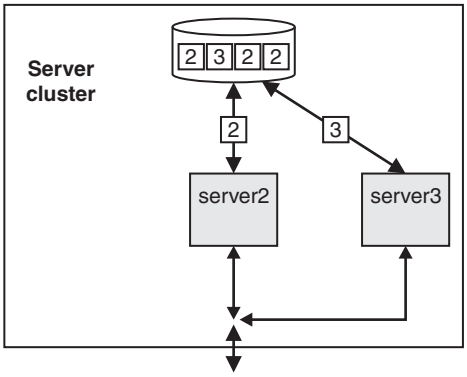


Figure 14. Server cluster stable again with just two servers: server 2 and server 3.

Deployment considerations for transactional high availability:

Before you use the high availability (HA) function, you must consider deployment issues such as your file system type, or where you plan to store the transaction recovery logs. In particular, your file system type can have important consequences for your recovery configuration.

Common configuration

Transaction peer recovery requires a common configuration of the resource providers between the participating server members to perform peer recovery between servers. Therefore, peer recovery processing can only take place between members of the same server cluster. Although a cluster can contain servers that are at different versions of WebSphere Application Server, peer recovery can only be performed between servers in the cluster that are at Version 6 or later.

Physical storage

For application servers to perform transaction peer recovery for each other, they must be able to access the transaction recovery logs of all the other members in the cluster. Ensure that the log files are stored on a medium that is accessible by all members of the cluster, and that each cluster member has a unique log file location on this medium. This medium, and access to it, for example through a local area network

(LAN), must support the file-based force operation that is used by the recovery log service to force data to disk. After the force operation is complete, information must be persistently stored on physical disk media.

For example, you can use IBM Network attached storage (NAS) (http://www.ibm.com/servers/storage/nas/index.html) mounted on each node, and shared SCSI drives, but not simple network share. All nodes must have read and write access to the recovery logs.

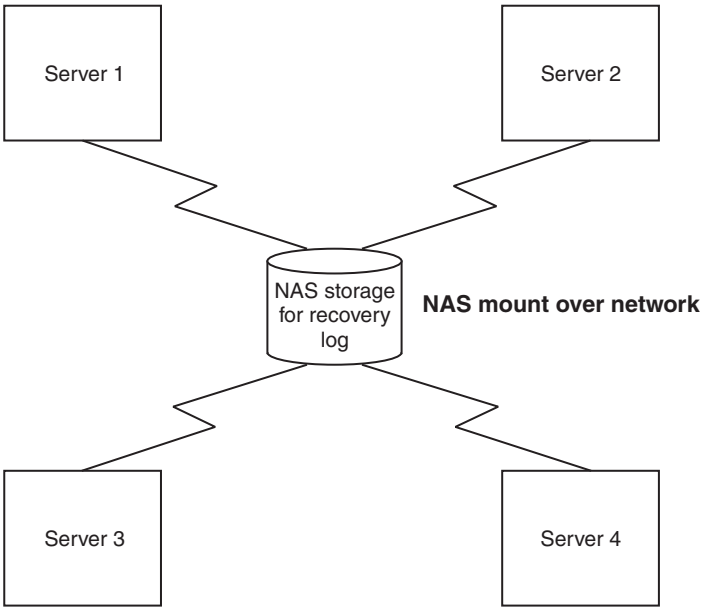


Figure 15. Recovery logs on NAS storage are available to all servers

In addition, configure the mechanism by which the remote log files are accessed, to exploit any fault tolerance in the underlying file system. For example, by using the Network File System (NFS) and hard mounting the remote directory containing the log files using the -o hard option of the NFS mount command, the NFS client will retry a failed operation until the NFS server becomes available again.

Two types of potential server failure exist: software failure and hardware failure. Software failures generally do not affect other application servers directly. Even servers on the same physical hardware can perform peer recovery processing. If a hardware failure occurs, all the servers that are deployed on the failed hardware become unavailable. Servers on other hardware are required to handle peer recovery processing. Any HA configuration requires that servers are deployed across multiple and discrete hardware systems.

File system

The file system type is an important deployment consideration as it is the main factor in deciding whether to use automated or manual peer recovery. For more information, see “How to choose between automated and manual transaction peer recovery.”

How to choose between automated and manual transaction peer recovery:

Your type of file system is the dominant factor in deciding which kind of transaction peer recovery to use. Different file systems have different behaviors, and the file locking behavior in particular is important when choosing between automated and manual peer recovery.

WebSphere Application Server high availability (HA) support uses a heartbeat mechanism to determine whether servers are still running. Servers are considered failed if they stop responding to heartbeat requests. Some scenarios, such as system overloading and network partitioning (explained elsewhere in this topic), can cause servers to stop responding to heartbeats, even though the servers are still running. WebSphere Application Server uses file locking technology to prevent such events from causing concurrent access to transaction recovery logs, because access to a recovery log by more than one server can lead to loss of data integrity.

However, not all file systems provide the necessary file locking semantics, specifically that file locks are released when a server fails. For example, Network File System Version 4 (NFSv4) provides this release behavior, whereas Network File System Version 3 (NFSv3) does not.

NFSv4 releases locks held on behalf of a host in case that host fails. Peer recovery can occur automatically without the need to restart the failed hardware. Therefore, this version of NFS is better suited for use with automated peer recovery.

NFSv3 holds file locks on behalf of a failed host until that host can restart. In this context, the host is the physical machine running the application server that requested the lock and it is the restart of the host, not the application server, that eventually triggers the locks to release.

To illustrate file locking on NFSv3, consider the behavior when a cluster member fails:

1. Server H is running on host H and holds an exclusive file lock for its own recovery log files.
2. Server P is running on host P and holds an exclusive file lock for its own recovery log files.
3. Host H fails, taking server H with it. The NFS lock manager on the file server holds the locks that are granted to server H on its behalf.
4. A peer recovery event is triggered in server P for server H by WebSphere Application Server.
5. Server P attempts to gain an exclusive file lock for this peer recovery log, but is unable to do so as it is held on behalf of server H. The peer recovery process is blocked.
6. At an unspecified time, host H is restarted. The locks held on its behalf are released.
7. The peer recovery process in server P is unblocked and granted the exclusive file locks that are needed to perform peer recovery.
8. Peer recovery takes place in server P for server H.
9. Server H is restarted.
10. If peer recovery is still in progress in server P, the recovery is halted.
11. Server P releases the exclusive lock on the recovery logs and returns ownership of the recovery logs back to server H.
12. Server H obtains the exclusive lock and can now perform standard transaction logging.

Because of this behavior, on NFSv3 you must disable file locking to use automated peer recovery. Disabling file locking can lead to concurrent access to recovery logs so it is vital that you protect your system from system overloading and network partitioning first. Alternatively, you can configure manual peer recovery, where you prevent concurrent access by manually triggering peer recovery processing only for servers that have actually failed.

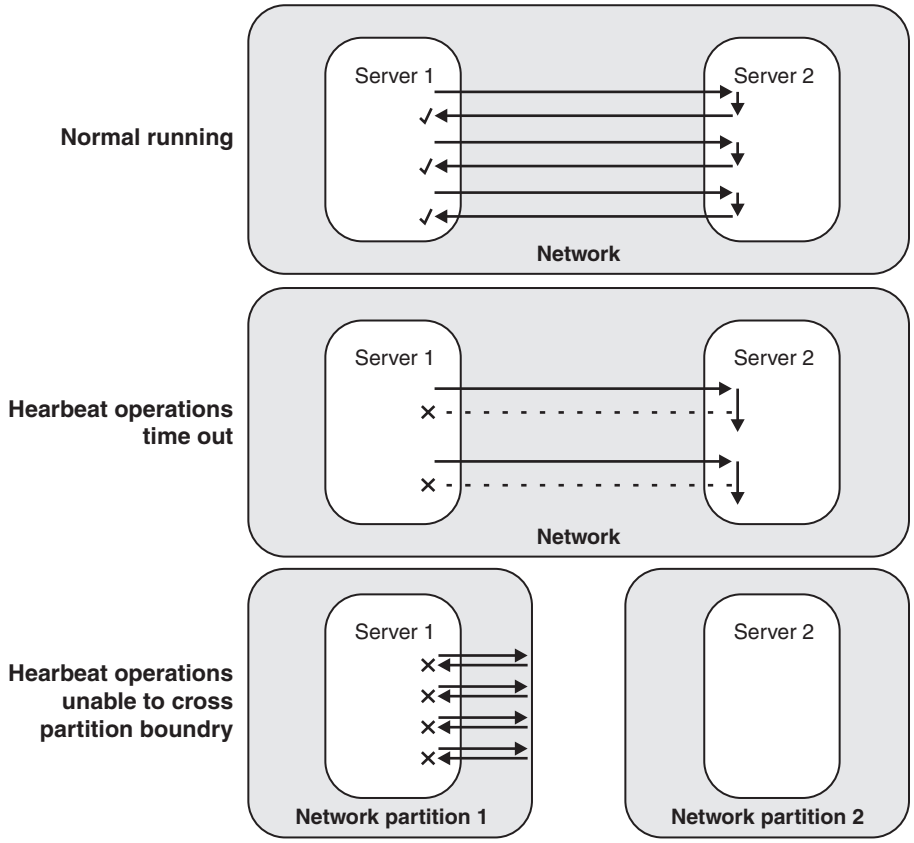
System overloading

System overloading occurs when a machine becomes very heavily loaded such that response times are extremely poor and requests begin to time out. Several potential causes exist for such overloading, including:

- The server is underpowered and cannot handle the workload.
- The server received a temporary surge of requests.
- Insufficient physical memory is available. As a result, the operating system is too busy paging to give the application server the required CPU time.

Network partitioning

Network partitioning occurs when a communications failure in a network results in two smaller networks that are independent and cannot contact each other.



During normal running, two servers on the network exchange heartbeats. During system overloading, heartbeat operations time out, giving the appearance of a server failure. After network partitioning, each server is in a separate network and heartbeats cannot pass between them, also giving the appearance of a server failure.

Figure 16. Heartbeats in a system running normally, compared to heartbeats after the apparent server failures of system overloading and network partitioning

High availability policies for the transaction service:

WebSphere Application Server provides integrated high availability (HA) support in which system subcomponents, such as the transaction service, are made highly available. An HA policy provides the logic that governs the manner in which each WebSphere Application Server HA component behaves within the overall HA framework. In the case of the transaction service, the transaction HA policy provides the logic to determine which servers own a recovery log at any time.

Typically, transaction policies assign ownership of a recovery log to the server that originally created it (the home server) and that server can then use the recovery log for both recovery and normal transactional activity. In the event that the home server is unavailable or fails, ownership can pass to a peer server to perform recovery processing.

Conceptually, a policy can be thought of as consisting of two key components, a policy type and a policy configuration.

Policy type

The policy type determines whether peer recovery initiation is manual or automated. The policy essentially provides the logic for determining updated recovery log ownership in the event of a server failure. The following WebSphere Application Server policy types are used for transaction peer recovery (other HA policy types exist, but are not used by the transaction service):

Static Ownership of the recovery log is defined in the WebSphere Application Server configuration. At run time, the static policy assigns ownership accordingly. Any changes to ownership require a change to the static configuration and therefore this policy type is used for manually initiated peer recovery.

One-of-N
Ownership of the recovery log is determined dynamically by the WebSphere Application Server HA framework and assigned to exactly one of the N cluster members. This policy type is used for automated peer recovery.

Transaction compensation and business activity support

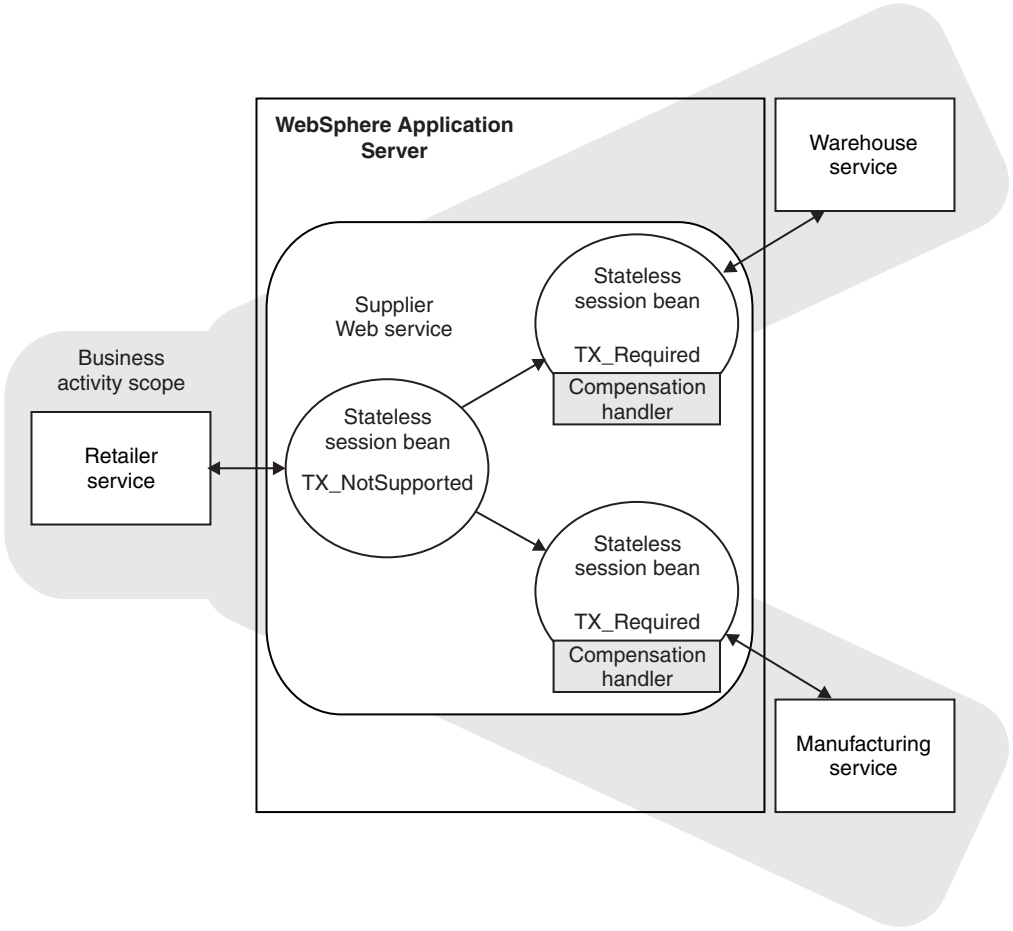
A *business activity* is a collection of tasks that are linked together so that they have an agreed outcome. Unlike atomic transactions, activities such as sending an e-mail can be difficult or impossible to roll back atomically, and therefore require a compensation process in the event of an error. The WebSphere Application Server business activity support provides this compensation ability through *business activity scopes*.

When to use business activity support

Use the business activity support when you have an application that requires compensation. An application requires compensation if its operations cannot be atomically rolled back. Typically, this scenario is because of one of the following reasons:

- The application uses multiple non-extended-architecture (XA) resources.
- The application uses more than one atomic transaction, for example, enterprise beans that have **Requires new** as the setting for the **Transaction** field in the container transaction deployment descriptor.
- The application does not run under a global transaction.

The following diagram shows a simple Web service application that uses the business activity support. The Retailer, Warehouse and Manufacturing services are running in non-WebSphere Application Server environments. The Retailer service calls the Supplier service, running on WebSphere Application Server, which delegates tasks to the Warehouse and Manufacturing services. The implementation of the Supplier service contains a stateless session bean, which calls other stateless session beans that are associated with the Warehouse and Manufacturing services, and that perform work that can be compensated. These other session beans each have a *compensation handler*, a piece of logic that is associated with an application component at run time, and performs compensation activity such as resending an e-mail.



Application design considerations

Business activity contexts are propagated with application messages, and can therefore be distributed between application components that are not co-located in the same server. Unlike atomic transaction contexts, business activity contexts are propagated on both synchronous (blocking) call-response messages and asynchronous one-way messages. An application component that runs under a business activity scope is responsible for ensuring that any asynchronous work it initiates is complete before the component's own processing is complete. An application that initiates asynchronous work using a fire-and-forget message pattern must not use business activity scopes, because such applications have no means of detecting whether this asynchronous processing has completed.

Only enterprise beans that have container-managed transactions can use the business activity functionality. Enterprise beans that exploit business activity scopes can offer Web service interfaces, but can also offer standard enterprise bean local or remote Java interfaces. Business activity context is propagated in Web service messages using a standard, interoperable Web Services Business Activity (WS-BA) `CoordinationContext` element. WebSphere Application Server can also propagate business activity context on RMI calls to enterprise beans when Web services are not being used, but this form of the context is not interoperable with non-WebSphere Application Server environments. You might want to use this homogeneous scenario if you require compensation for an application that is internal to your business. If you want to use business activity compensation in a heterogeneous environment, expose your application components as Web services.

Business activity contexts can be propagated across firewalls and outside the WebSphere Application Server domain. The topology that you use to achieve this propagation can affect the high availability and affinity behavior of the business activity transaction.

Application development and deployment considerations

WebSphere Application Server provides a programming model for creating business activity scopes, and for associating compensation handlers with those business activity scopes. WebSphere Application Server also provides an application programming interface to specify compensation data, and check or alter the status of a business activity. To use the business activity support you must set certain application deployment descriptors appropriately, provide a compensation handler class if required, and enable business activity support on any servers that run the application.

Note: Applications can exploit the business activity support only if you deploy them to a WebSphere Application Server at Version 6.1 or later. Applications cannot use the business activity support if you deploy them to a cluster that includes WebSphere Application Server Version 6.0.x servers.

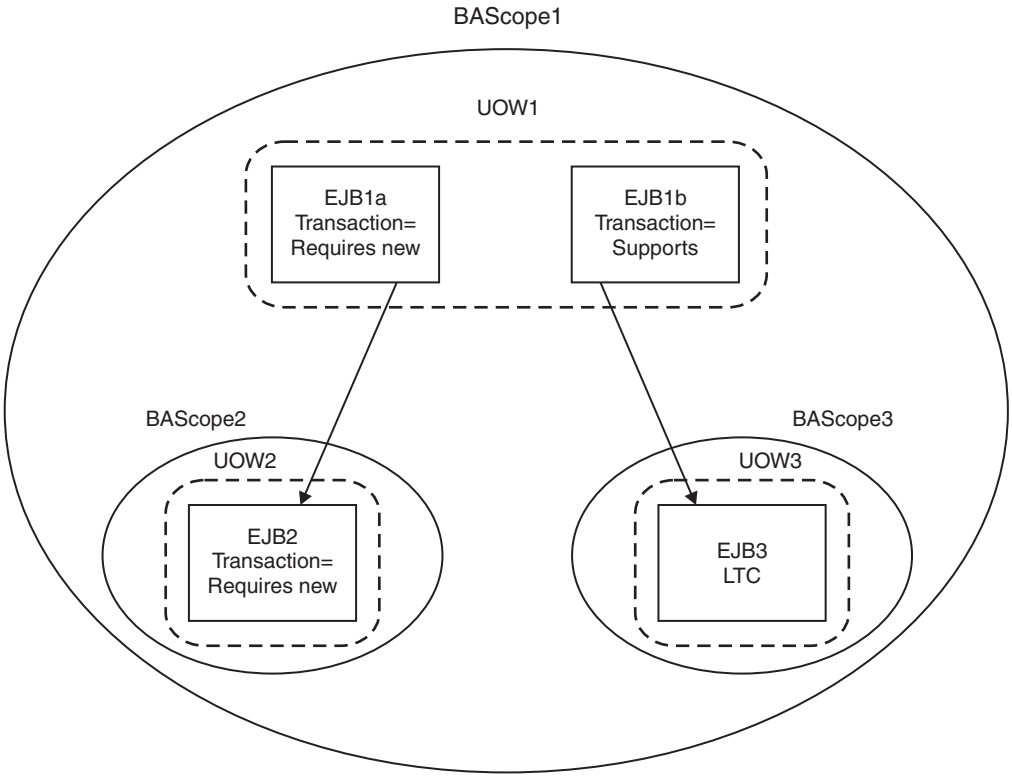
Business activity scopes

The scope of a business activity is that of a core WebSphere Application Server unit of work: a global transaction, an activity session, or local transaction containment (LTC). A business activity scope is not a new unit of work (UOW); it is an attribute of an existing core UOW. Therefore, a one-to-one relationship exists between a business activity scope and a UOW.

In a WS-BA deployment, the UOW must be container-managed:

- The UOW can be a container-managed transaction (CMT) enterprise bean that creates a global transaction.
- The UOW can be a local transaction containment (LTC) where the container is responsible for initiating and ending resource manager local transactions (RMLTs). That is, in the transactional deployment descriptor attributes, the Local Transaction attribute Resolver must be set to ContainerAtBoundary. To use WS-BA, you must not set the Resolver attribute to Application.

Any core UOW can have a business activity scope associated with it. If a component running under a UOW that is associated with a business activity scope calls another component, that request propagates the business activity scope; any work done by the new component is associated with the same business activity scope as the calling component. The called component can create a new UOW, for example if an enterprise bean has a **Transaction** setting of **Requires new**, or runs under the same UOW as the calling component. If a new UOW is started then a new business activity scope is created and associated with the new UOW. The newly created business activity scope is a child of the business activity scope associated with the calling UOW. In the following diagram, EJB1a running under UOW1 calls two components: EJB1b that also runs under UOW1, and EJB2 that creates a new UOW, UOW2. The enterprise bean EJB1b, calls another enterprise bean, EJB3, which creates another new UOW, UOW3. Because each new UOW is created by a calling component whose UOW already has an association with business activity scope BAScope1, the newly created UOWs are associated with new inner business activity scopes, BAScope2 and BAScope3.



Inner business activity scopes must complete before the outer business activity scope completes. Inner business activity scopes, for example BAScope2, have an association with the outer business activity scope, in this case BAScope1. Each business activity scope is directed to close if its associated UOW completes successfully, or to compensate if its associated UOW fails. If BAScope2 completes successfully, any active compensation handlers that are owned by BAScope2 are moved to BAScope1, and are directed in the same way as the completion direction of BAScope1: either compensate or close. If BAScope2 fails, the active compensation handlers are compensated automatically, and nothing is moved to the outer BAScope1. When an inner business activity scope fails, as a result of its associated UOW failing, an application server exception is thrown to the to calling application component, running in the outer UOW.

For example, if the inner UOW fails it might throw a TransactionRolledBackException exception. If the calling application can handle the exception, for example by retrying the called component or by calling another component, then the calling UOW, and its associated business activity scope, can complete successfully even though the inner business activity scope failed. If the application design requires the calling UOW to fail, and for its associated business activity scope to be compensated, then the calling application component must cause its UOW to fail, for example by allowing any system exception from the UOW that failed to be handled by its container.

When the outer business activity scope completes, its success or failure determines the completion direction (close or compensate) of any active compensation handlers that are owned by the outer business activity scope, including those promoted by the successful completion of inner business activity scopes. If the outer business activity scope completes successfully, it drives all active compensation handlers to close. If the outer business activity scope fails, it drives all active compensation handlers to compensate.

This compensation behavior is summarized in the following table.

Table 38. Compensation behavior for a single business activity scope

Inner business activity scope	Outer business activity scope	Compensation behavior
Succeeds	Succeeds	Any compensation handlers that are owned by the inner business activity scope wait for the outer UOW to complete. When the outer UOW succeeds, the outer business activity scope drives all compensation handlers to close.
Fails	Succeeds	Any active compensation handlers that are owned by the inner business activity scope are compensated. An exception is thrown to the outer UOW; if this exception is caught, when the outer UOW succeeds, the outer business activity scope drives all remaining active compensation handlers to close.
Fails	Fails	Any active compensation handlers that are owned by the inner business activity scope are compensated. An exception is thrown to the outer UOW; if this exception is not caught, the outer business activity scope fails. When the outer business activity scope fails, either because of the unhandled exception or for some other reason, all remaining active compensation handlers are compensated.
Succeeds	Fails	Any compensation handlers that are owned by the inner business activity scope wait for the outer UOW to complete. When the outer UOW fails, the outer business activity scope drives all compensation handlers to compensate.

When a UOW with an associated business activity scope completes, the business activity scope always completes in the same direction as the UOW that it is associated with. The only way that you can influence the direction of the business activity scope is to influence the UOW that it is associated with, which you can do by using the setCompensateOnly method of the business activity API.

A compensation handler that is registered within a transactional UOW might initially be inactive, depending on the method invoked from the business activity API. Inactive handlers in this situation become active when the UOW in which that handler is declared completes successfully. A compensation handler that is registered outside a transactional UOW always becomes active immediately. For more information, see Business activity API.

Each business activity scope in the diagram represents a business activity. For example, the outer business activity running under BAScope1 can be a holiday booking scenario, with BAScope2 being a flight booking activity and BAScope3 a hotel booking. If either the flight or hotel bookings fail, the overall holiday booking by default also fails. Alternatively if, for example, the flight booking fails, you might want your application to try booking a flight using another component that represents a different airline. If the overall holiday booking fails, the application can use compensation handlers to cancel any flights or hotels that are already successfully booked.

Use of business activity scopes by application components

Application components do not use business activity scopes by default. You use the WebSphere Application Server assembly tools to specify the use of a business activity scope and to identify any compensation handler class for the component:

Default configuration
: If a business activity context is present on a request received by a component with no business activity scope configuration, the context is stored by the container but never used during the method scope of the target component. A new business activity scope is not created. If the target component invokes another component, the stored business activity context is propagated and can be used by other compensating components.

Run enterprise bean methods under a business activity scope
: Any business activity context present on the incoming request is received by the container and made available to the target component. If a new UOW is created for the target method, for example because the enterprise bean method has a **Transaction** setting of **Requires new**, the received business activity scope becomes an outer business activity scope to a newly created business activity. If the UOW is propagated from the calling component and used by the method,

then the received business activity scope is used by the method. If a business activity scope does not exist on the invocation, a new business activity scope is created and used by the method.

To create a business activity scope when an enterprise bean is invoked, you must configure the enterprise bean to run enterprise bean methods under a business activity scope. You must also configure the deployment descriptors for the method being invoked, to specify the creation of a new UOW upon invocation. For instructions on how to perform these actions, see Creating an application that uses the Web Services Business Activity support.

JTA support

Java Transaction API (JTA) support provides application programming interfaces (APIs) in addition to the UserTransaction interface that is defined in the JTA 1.1 specification.

These interfaces include the TransactionSynchronizationRegistry interface, which is defined in the JTA 1.1 specification, and the following API extensions:

- SynchronizationCallback interface
- ExtendedJTATransaction interface
- UOWSynchronizationRegistry interface
- UOWManager interface

Note: This release features additional Java Transaction API (JTA) support. JTA 1.1 is supported through the introduction of the TransactionSynchronizationRegistry interface. System-level application components, such as persistence managers, resource adapters, enterprise beans, and Web application components, can use the TransactionSynchronizationRegistry interface to register with a JTA transaction. Also, the UOWSynchronizationRegistry interface and the UOWManager interface are introduced so that all types of units of work (UOWs) that the product supports can register with a JTA transaction and can be manipulated.

The APIs provide the following functionality:

- Access to global and local transaction identifiers associated with the thread.

  The global identifier is based on the transaction identifier in the CosTransactions::PropagationContext: object and the local identifier identifies the transaction uniquely in the local Java virtual machine (JVM).
- A transaction synchronization callback that any enterprise application component can use to register an interest in transaction completion.

  Advanced applications can use this callback to flush updates before transaction completion and clear up state after transaction completion. Java EE (and related) specifications position this function typically as the domain of the enterprise application containers.

  Components such as persistence managers, resource adapters, enterprise beans, and Web application components can register with a JTA transaction.

The following information is an overview of the interfaces that the JTA support provides. For more detailed information, see the generated API documentation.

SynchronizationCallback interface

An object implementing this interface is enlisted once through the ExtendedJTATransaction interface, and receives notification of transaction completion.

Although an object implementing this interface can run on a Java platform for enterprise applications server, there is no specific enterprise application component active when this object is called. So, the object has limited direct access to any enterprise application resources. Specifically, the object has no access to the java: namespace or to any container-mediated resource. Such an object can cache a reference to an enterprise application component (for example, a stateless session bean) that it delegates to. The object would then have all the normal access to enterprise application resources. For example, you

could use the object to acquire a Java Database Connectivity (JDBC) connection and flush updates to a database during the beforeCompletion method.

ExtendedJTATransaction interface

This interface is a WebSphere programming model extension to the Java EE JTA support. An object implementing this interface is bound, by enterprise application containers in WebSphere Application Server that support this interface, at java:comp/websphere/ExtendedJTATransaction. Access to this object, when called from an Enterprise JavaBeans (EJB) container, is not restricted to component-managed transactions.

An application uses a Java Naming and Directory Interface (JNDI) lookup of java:comp/websphere/ExtendedJTATransaction to get an ExtendedJTATransaction object, which the application uses as shown in the following example:

```
ExtendedJTATransaction exJTA = (ExtendedJTATransaction)ctx.lookup("
 java:comp/websphere/ExtendedJTATransaction");
SynchronizationCallback sync = new SynchronizationCallback();
exJTA.registerSynchronizationCallback(sync);
```

The ExtendedJTATransaction object supports the registration of one or more application-provided SynchronizationCallback objects. Depending on how the callback is registered, each registered callback is called at one of the following points:

- At the end of every transaction that runs on the application server, whether the transaction is started locally or imported
- At the end of the transaction for which the callback was registered

Note: In this release, the registerSynchronizationCallbackForCurrentTran method is deprecated. Use the registerInterposedSynchronization method of the TransactionSynchronizationRegistry interface instead.

TransactionSynchronizationRegistry interface

This interface is defined in the JTA 1.1 specification. System-level application components, such as persistence managers, resource adapters, enterprise beans, and Web application components, can use this interface to register with a JTA transaction. Then, for example, the component can flush a cache when a transaction completes.

To obtain the TransactionSynchronizationRegistry interface, use a JNDI lookup of java:comp/TransactionSynchronizationRegistry.

Note: Use the registerInterposedSynchronization method to register a synchronization instance, rather than the registerSynchronizationCallbackForCurrentTran method of the ExtendedJTATransaction interface, which is deprecated in this release.

UOWSynchronizationRegistry interface

This interface provides the same functionality as the TransactionSynchronizationRegistry interface, but applies to all types of units of work (UOWs) that WebSphere Application Server supports:

- JTA transactions
- local transaction containments (LTCs)
- ActivitySession contexts

System-level application server components such as persistence managers, resource adapters, enterprise beans, and Web application components can use this interface to register with a JTA transaction. The component can do the following:

- Register synchronization objects with special ordering semantics.

- Associate resource objects with the UOW.
- Get the context of the current UOW.
- Get the current UOW status.
- Mark the current UOW for rollback.

To obtain the UOWSynchronizationRegistry interface, use a JNDI lookup of java:comp/websphere/UOWSynchronizationRegistry. This interface is available only in a server environment.

The following example registers an interposed synchronization with the current UOW:

```
// Retrieve an instance of the UOWSynchronizationRegistry interface from JNDI.
final InitialContext initialContext = new InitialContext();
final UOWSynchronizationRegistry uowSyncRegistry =
  (UOWSynchronizationRegistry)initialContext.lookup("java:comp/websphere/UOWSynchronizationRegistry");

// Instantiate a class that implements the javax.transaction.Synchronization interface
final Synchronization sync = new SynchronizationImpl();

// Register the Synchronization object with the current UOW.
uowSynchronizationRegistry.registerInterposedSynchronization(sync);
```

UOWManager interface

The UOWManager interface is equivalent to the JTA TransactionManager interface, which defines the methods that allow an application server to manage transaction boundaries. Applications can use the UOWManager interface to manipulate UOW contexts in the product. The UOWManager interface applies to all types of UOWs that WebSphere Application Server supports; that is, JTA transactions, local transaction containments (LTCs), and ActivitySession contexts. Application code can run in a particular type of UOW without needing to use an appropriately configured enterprise bean. Typically, the logic that is performed in the scope of the UOW is encapsulated in an anonymous inner class. System-level application server components such as persistence managers, resource adapters, enterprise beans, and Web application components can use this interface.

WebSphere Application Server does not provide a TransactionManager interface in the API or the system programming interface (SPI). The UOWManager interface provides equivalent functionality, but WebSphere Application Server maintains control and integrity of the UOW contexts.

To obtain the UOWManager interface in a container-managed environment, use a JNDI lookup of java:comp/websphere/UOWManager. To obtain the UOWManager interface outside a container-managed environment, use the UOWManagerFactory class. This interface is available only in a server environment.

You can use the UOWManager interface to migrate a Web application to use Web components rather than enterprise beans, but maintain control over the UOWs. For example, a Web application currently uses the UserTransaction interface to begin a global transaction, makes a call to a method on a session enterprise bean that is configured as not supported to perform some non-transactional work, and then completes the global transaction. You can move the logic that is encapsulated in the session EJB method to the run method of a UOWAction implementation. Then, you replace the code in the Web component that calls the session enterprise bean with a call to the runUnderUOW method of a UOWManager interface to request that this logic is run in a local transaction. In this way, you maintain the same level of control over the UOWs as you had with the original application.

The following example performs some transactional work in the scope of a new global transaction. The transactional work is performed in an anonymous inner-class that implements the run method of the UOWAction interface. Any checked exceptions that the run method creates do not affect the outcome of the transaction.

```
// Retrieve an instance of the UOWManager interface from JNDI.
final InitialContext initialContext = new InitialContext();
final UOWManager uowManager = (UOWManager)initialContext.lookup("java:comp/websphere/UOWManager");

try
```

```
{
 // Invoke the runUnderUOW method, indicating that the logic should be run in a global
 // transaction, and that any existing global transaction should not be joined, that is,
 // the work must be performed in the scope of a new global transaction.
 uowManager.runUnderUOW(UOWSynchronizationRegistry.UOW_TYPE_GLOBAL_TRANSACTION, false, new UOWAction()
 {
   public void run() throws Exception
   {
   // Perform transactional work here.
   }
 });
}

catch (UOWActionException uowae)
{
 // Transactional work resulted in a checked exception being thrown.
}

catch (UOWException uowe)
{
 // The completion of the UOW failed unexpectedly. Use the getCause method of the
 // UOWException to retrieve the cause of the failure.
}
```

Local transaction containment considerations

IBM WebSphere Application Server supports local transaction containment (LTC), which you can configure using local transaction extended deployment descriptors. This topic describes the advantages that this support provides to application programmers, and the relevant settings to use. It also lists points to consider when deciding the best way to configure transaction support for local transactions.

The following sections describe the advantages that LTC support provides, and how to set the local transaction extended deployment descriptors in each situation.

You can develop an enterprise bean or servlet that accesses one or more databases that are independent and require no coordination.

If an enterprise bean does not need to use global transactions, it is often more efficient to deploy the bean with the deployment descriptor for the container transaction type set to NotSupported instead of Required.

With the extended local transaction support of the application server, applications can perform the same business logic in an unspecific transaction context as they can in a global transaction. An enterprise bean, for example, runs in an unspecified transaction context if it is deployed with a container transaction type of NotSupported or Never.

The extended local transaction support provides a container-managed, implicit local transaction boundary, within which the container commits application updates and cleans up their connections. You can design applications with more independence from deployment concerns. This makes using a container transaction type of Supports much simpler, for example, when the business logic might be called either with or without a global transaction context.

An application can follow a get-use-close pattern of connection usage, regardless of whether the application runs in a transaction. The application can depend on the close action behaving in the same way in all situations, that is, the close action does not cause a rollback to occur on the connection if there is no global transaction.

There are many scenarios where ACID coordination of multiple resource managers is not needed. In such scenarios, running business logic in a Transaction policy of NotSupported performs better than in a policy of Required. This benefit is applied through setting the deployment descriptor, in the Local Transactions section, of the Resolver attribute to ContainerAtBoundary. With this setting, application interactions with resource providers, such as databases, are managed within implicit resource manager local transactions (RMLT) that the container both starts and ends. The container commits RMLTs at the containment boundary that is specified by the Boundary attribute in the Local Transactions section; for example, at the end of a method. If the application returns control to the container by an exception, the container rolls back any RMLTs that it has started.

This usage applies to both servlets and enterprise beans.

You can use local transactions in a managed environment that guarantees cleanup.

Applications that want to control RMLTs, by starting and ending them explicitly, can use the default setting of Application for the Resolver extended deployment descriptor in the Local Transactions section. In this situation, the container ensures connection cleanup at the boundary of the local transaction context.

Java platform for enterprise applications specifications that describe application use of local transactions do so in the manner provided by the default settings of Application for the Resolver extended deployment descriptor, and Rollback for the Unresolved action extended deployment descriptor, in the Local Transactions section. When the Unresolved action extended deployment descriptor in the Local Transactions section is set to Commit, the container commits any RMLTs that the application starts but that do not complete when the local transaction containment ends (for example, when the method ends). This usage applies to both servlets and enterprise beans.

You can coordinate multiple one-phase resource managers.

For resource managers that do not support XA transaction coordination, a client can use ActivitySession-bounded local transaction contexts. Such contexts give a client the same ability to control the completion direction of the resource updates by the resource managers as the client has for transactional resource managers. A client can start an ActivitySession and call its entity beans in that context. Those beans can perform their RMLTs within the scope of that ActivitySession and return without completing the RMLTs. The client can later complete the ActivitySession in a commit or rollback direction and cause the container to drive the ActivitySession-bounded RMLTs in that coordinated direction.

You can use shareable LTCs to reduce the number of connections you require.

Application components can share LTCs. If components obtain connections to the same resource manager, they can share that connection if they run under the same global transaction or shareable LTC. To configure two components to run under the same shareable LTC, set the Shareable attribute of the Local Transactions section in the deployment descriptor of each component. Make sure that the resource reference in the deployment descriptor for each component uses the default value of Shareable for the res-sharing-scope element, if this element is specified. A shareable LTC can reduce the numbers of RMLTs an application uses. For example, an application that makes frequent use of Web module include calls can share resource manager connections between those Web modules, exploiting either shareable LTCs, or a global transaction, reducing lock contention for resources.

Examples of local transaction support configurations

The following list gives scenarios that use local transactions, and points to consider when deciding the best way to configure the transaction support for an application.

- You want to start and end global transactions explicitly in the application (bean-managed transaction session beans and servlets only).

  For a session bean, set the Transaction type to Bean (to use bean-managed transactions) in the deployment descriptor of the component. You do not need to do this for servlets.
- You want to access several XA resources atomically across one or more bean methods.

  In the deployment descriptor of the component, in the Container Transactions section, set the container transaction type to Required, RequiresNew, or Mandatory.
- You want to access several non-XA resources in a method and want to manage them independently.

  In the deployment descriptor of the component, in the Local Transactions section, set the Resolver attribute to Application and set the Unresolved action attribute to Rollback. In the Container Transactions section, set the container transaction type to NotSupported.
- You want to use a non-XA resource with multiple two-phase RRS resources.

  A non-XA resource in a transaction along with RRS resources is supported any time a global transaction is active. A global transaction is active when the deployment descriptor for the container transaction type

is set to Supports, Required, RequiresNew, or Mandatory. Global transactions also are active for component-managed deployments. The container manages the completion of the non-XA resource local transaction together with the RRS resources.

Transaction service custom property

The product provides the `DISABLE_TRANSACTION_TIMEOUT_GRACE_PERIOD` and `RLS_log stream_COMPRESS_INTERVAL` transaction service custom properties.

Transaction Service custom properties can be specified in the administrative console by clicking **Servers >Server Types > Application Servers >** *server* **> Container Services > Transaction service > Custom Properties > New**.

DISABLE_TRANSACTION_TIMEOUT_GRACE_PERIOD

Specifies whether there is a delay between a transaction timeout and the abnormal ending of the servant region that was running the transaction.

If you set this value to false, a global transaction that times out is marked rollback-only. The transaction server gives the associated application an additional period of time, approximately four minutes, to complete. If the application completes in this time, the transaction is rolled back. If the application does not complete in this time, the application and associated servant region are ended abnormally, with an ABENDEC3 or ABENDSEC3 error.

Set this value to true to remove the delay, and abnormally end the application and servant region immediately.

Data Type	Boolean
Acceptable values	true, false
Default	false

DISABLE_WSTX_RMFAIL_LOGGING

Specifies whether an RMFAIL message is sent to the error log file when a WS-AT participant fails to send a response within the asynchronous response timeout period, causing an XAER_RMFAIL transaction exception to occur.

If you set this value to false, an RMFAIL message is sent to the error log file when a WS-AT participant fails to send a response within the asynchronous response timeout period.

Set this value to true if you do not want an RMFAIL message sent to the error log file when a WS-AT participant fails to send a response within the asynchronous response timeout period.

Data Type	Boolean
Acceptable values	true, false
Default	false

RLS_log stream_COMPRESS_INTERVAL

Specifies, in seconds, the interval at which the recovery log service attempts to compress any log streams application components are using. The Transaction Service (XA partner log) and the Compensation Service components can be configured to use the recovery log service.

The log stream is checked for compression once per interval. This operation can cause unnecessary CPU usage if the log stream is not being used.

Note:

- If you do not use a log stream for the compensation service, set this property to a value that is higher than the default value.
- If your recovery log service uses log streams, do not set this property to too high a value. If the recovery log service log streams become full before the compression interval expires, transactions might start to fail until the log streams are compressed.
- If none of your components are configured to use a log stream, you set this property to 0 (zero) to disable this function.

Data Type	Integer
Acceptable values	0 - 2,147,483,647 (0 disables the function)
Default	30 seconds

Transaction service exceptions

The exceptions that the WebSphere Application Server transaction service can throw are listed with a summary of each exception.

The exceptions are grouped as follows:
- Standard exceptions
- Heuristic exceptions

If the EJB container catches a system exception from the business method of an enterprise bean, and the method is running within a container-managed transaction, the container rolls back the transaction before passing the exception on to the client. For more information about how the container handles the exceptions thrown by the business methods for beans with container-managed transaction demarcation, see the section *Exception handling* in the Enterprise JavaBeans 2.0 specification. That section specifies the container's action as a function of the condition under which the business method executes and the exception thrown by the business method. It also illustrates the exception that the client receives and how the client can recover from the exception.

Standard exceptions

The standard exceptions such as TransactionRequiredException, TransactionRolledbackException, and InvalidTransactionException are defined in the Java Transaction API (JTA) 1.1 specification.

InvalidTransactionException
This exception indicates that the request carried an invalid transaction context.

TransactionRequiredException exception
This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.

TransactionRolledbackException exception
This exception indicates that the transaction associated with processing of the request has been rolled back, or marked for roll back. Thus the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

Heuristic exceptions

A heuristic decision is a unilateral decision made by one or more participants in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the Transaction Service. Heuristic decisions are an issue only after the participant has been prepared and the second phase of commit processing is underway. Heuristic decisions are normally made only in unusual circumstances, such as repeated failures by the transaction manager to communicate with a resource manage during two-phase commit. If a heuristic decision is taken, there is a risk that the decision differs from the consensus outcome, resulting in a loss of data integrity.

The following list provides a summary of the heuristic exceptions. For more detail, see the Java Transaction API (JTA) 1.1 specification.

HeuristicRollback exception
: This exception is raised on the commit operation to report that a heuristic decision was made and that all relevant updates have been rolled back.

HeuristicMixed exception
: This exception is raised on the commit operation to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

CICS tuning tips for z/OS

These recommendations apply only to WebSphere applications that access CICS.

The LGDFINT system initialization parameter specifies the log defer interval used by CICS log manager when determining how long to delay a forced journal write request before invoking the MVS system logger. The value is specified in milliseconds. Performance evaluations of typical CICS transaction workloads have shown that the default setting of 5 milliseconds gives the best balance between response time and central processor cost. Be aware that CICS performance can be adversely affected by a change to the log defer interval value. Too high a value will delay CICS transaction throughput due to the additional wait before invoking the MVS system logger. An example of a scenario where a reduction in the log defer interval might be beneficial to CICS transaction throughout would be where many forced log writes are being issued, and little concurrent task activity is occurring. Such tasks will spend considerable amounts of their elapsed time waiting for the log defer period to expire. In such a situation, there is limited advantage in delaying a call to the MVS system logger to write out a log buffer, since few other log records will be added to the buffer during the delay period.

- Set the LGDFINT system initialization parameter to 5.

  While CICS is running, you can use the CEMT SET SYSTEM[LOGDEFER(*value*)] command to alter the LGDFINT setting dynamically.
- Set the CICS RECEIVECOUNT value high enough to handle all concurrent EXCI pipes on the system.

  The default value is 4. You set this value in the EXCI sessions resource definition.

For more detailed information on CICS, refer to the *CICS Performance Guide*.

GRS tuning tips for z/OS

WebSphere Application Server for z/OS uses global resource serialization (GRS) to communicate information between servers in a sysplex. When there are multiple servers defined in a system or a sysplex, a request may end up on the wrong server. To determine where the transaction is running, WebSphere Application Server uses GRS. Therefore, if you are using global transactions, WebSphere Application Server will issue an enqueue for that transaction at the start of the transaction and hold on to that enqueue until the transaction ends. WebSphere Application Server for z/OS uses GRS enqueues for the following:

- Two-phase commit transactions involving more than one server
- HTTP sessions in memory
- Stateful EJBs
- ″Sticky″ transactions to keep track of pseudo-conversational states.
- If you **are not** in a sysplex, you should configure GRS=NONE.
- If you **are** in a sysplex, we strongly recommend GRS=STAR.

This requires configuring GRS to use the coupling facility. All of the WebSphere Application Server enqueues are issued with RNL=NO, which prevents misconfiguring the GRSRNLxx with inappropriate values. See the GRS documentation for details on setting this up.

Developing components to use transactions

These topics provide information about developing WebSphere application components to use transactions

About this task

The way that applications use transactions depends on the type of application component, as follows:
- A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (component-managed transactions where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) and application client components use component-managed transactions.

Use the following tasks to develop WebSphere application components that use transactions:
- Configuring transactional deployment attributes. This task determines whether EJB components use container- or bean-managed transactions by setting an appropriate value on the Transaction type deployment attribute. You can also configure other transactional deployment descriptor attributes.
- Using component-managed transactions. If you want a session bean, Web component, or application client component to manage its own transactions, you must write the code that explicitly demarcates the boundaries of a transaction. There are some limitations to the transaction support available to application client components, as described in Client support for transactions

Configuring transactional deployment attributes

You can configure the transactional deployment descriptor attributes associated with an EJB or Web module, to enable an enterprise application to use transactions.

Before you begin

You must have an enterprise archive (EAR) file for an application component that can be deployed in the application server.

About this task

You can configure the deployment attributes of an application by using an assembly tool.

This topic describes the use of Rational Application Developer to configure the deployment attributes of an application.

To set transactional attributes in the deployment descriptor for an application component (enterprise bean or servlet), complete the following steps.

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, use the Import wizard to import the EAR file into the assembly tool. To start the Import wizard:
   a. Click **File** → **Import** → **EAR file**.
   b. Click **Next**, then select the EAR file.
   c. Click **Finish**.
3. In the Project Explorer view of the Java EE perspective, right-click the component instance, then click **Open With** → **Deployment Descriptor Editor**. To locate the component instance, use the appropriate step:
   - For a session bean, expand **EJB Modules** → ***ejb_module_instance*** → **Deployment Descriptor** → **Session Beans**, then select the bean instance.
   - For a servlet, expand **Web Modules** → ***web_application*** → **Deployment Descriptor** → ***web component***, then select the servlet instance.

A property dialog notebook for the deployment descriptor of the component is displayed in the property pane.

4. Optional: For session beans only, set the “Transaction type” attribute, which defines the transactional manner in which the container invokes a method. You can set this attribute to Container or Bean, as follows:
   - To use container-managed transactions, set the attribute to Container.
   - To use bean-managed transactions, set the attribute to Bean.
5. In the deployment descriptor notebook, select the **Bean** tab. Optionally, in the WebSphere Extensions section, configure the Local Transaction attributes. To enable management of local transaction containments, configure the following component extensions attributes. These attributes configure, for the component, the behavior of the container’s local transaction containment (LTC) environment that the container establishes whenever a global transaction is not present.

Boundary

This setting specifies the containment boundary at which all contained resource manager local transactions (RMLTs) must be completed. Possible values are BeanMethod or ActivitySession.

- BeanMethod: This is the default value. If you select this option, RMLTs must be resolved within the same bean method in which they were started.
- [For EJB components only] ActivitySession: RMLTs must be resolved within the scope of any ActivitySession in which they were started or, if no ActivitySession context is present, within the same bean method in which they were started.

Note: The ActivitySession option is not supported in the Web container.

Resolver

This setting specifies the component responsible for initiating and ending RMLTs. Possible values are Application or ContainerAtBoundary.

- Application: This is the default value. The application is responsible for starting RMLTs and for completing them within the local transaction containment (LTC) boundary. Any RMLTs that are not completed by the end of the LTC boundary are cleaned up by the container according to the value of the Unresolved action attribute.
- ContainerAtBoundary: The container is responsible for starting RMLTs and for completing them within the LTC boundary. The container begins an RMLT when a connection is first used within the LTC scope, and completes it automatically at the end of the LTC scope. If Boundary is set to ActivitySession, the RMLTs are enlisted as ActivitySession resources and directed to complete by the ActivitySession. If Boundary is set to BeanMethod, the RMLTs are committed at the end of the method by the container.

Unresolved action

Specifies the direction that the container requests RMLTs to take, if those transactions are unresolved at the end of the LTC boundary scope and the Resolver is set to Application. Possible values are Rollback or Commit.

- Rollback: This is the default value. At end of the LTC boundary scope, the container instructs all unresolved RMLTs to roll back.
- Commit: At the end of the LTC boundary scope, the container instructs all unresolved RMLTs to commit. The container instructs the RMLTs to commit only in the absence of an un-handled exception. If the application method that is running in the local transaction context ends with an exception, any unresolved RMLTs are rolled back by the container. This is the same behavior as for global transactions.

Shareable

Specifies whether the component can share an LTC. A new LTC is started only if a shareable LTC does not already exist. Applications that use shareable LTCs cannot explicitly commit or rollback resource manager connections that are used in a shareable LTC (although they can use connections that have an autoCommit capability). If an application starts any non-autocommit work in an LTC for which the Resolver attribute is set to Application, and the Shareable attribute is set to true, an exception is thrown at run time. For example, on a JDBC

Connection, non-autocommit work is work that the application performs after using the setAutoCommit(false) method to switch off the autocommit option on the connection. Enterprise beans that use bean managed transactions (BMT) cannot be assembled with the Shareable attribute set on the LTC configuration.

6. In the WebSphere Extensions section, configure the Global Transaction attributes. These attributes configure, for the component, behavior in the presence of a global transaction.

   Component Transaction Timeout
   For enterprise beans using container-managed transactions only, specifies the transaction timeout, in seconds, for any new global transaction that the container starts on behalf of the enterprise bean. For transactions started on behalf of the component, the Component Transaction Timeout setting overrides the default total transaction lifetime timeout that is configured in the transaction service settings for the application server.

   The following attributes enable WS-AtomicTransaction and WS-BusinessActivity support for JAX-RPC applications only:

   Use Web Services Atomic Transaction
   For enterprise beans only, when this attribute is selected, if the application component makes any Web service requests, any transaction context is propagated with the Web service requests in accordance with the WebSphere WS-AtomicTransaction support described in Web Services Atomic Transaction support in the application server. When this attribute is not selected, Web service requests do not carry transaction context.

   Send Web Services Atomic Transaction on requests
   For Web components only, when this attribute is selected, if the application component makes any Web service requests, any transaction context is propagated with the Web service requests in accordance with the WebSphere WS-AtomicTransaction support described in Web Services Atomic Transaction support in the application server. When this attribute is not selected, Web service requests do not carry transaction context.

   Execute using Web Services Atomic Transaction on incoming requests
   For Web components only, when this attribute is selected, Web application components are prepared to run under a received WS-AtomicTransaction context. A Web application component can run under a received WS-AtomicTransaction context in a similar way to an enterprise bean deployed with a container transaction type of Supports. When this attribute is not selected, the container of the Web application component suspends any received transaction context, in a similar way to the behavior of an EJB container for an enterprise bean deployed with a container transaction type of NotSupported.

   If your application uses JAX-WS, enable support for WS-AtomicTransaction or WS-BusinessActivity by creating a policy set, adding the WS-Transaction policy type to the policy set, and attaching the policy set to the service or client.

   If a policy set that is attached to a client includes the WS-Transaction policy type, any active global transaction context is propagated with a Web service request, in a similar way to the deployment descriptors Use Web Services Atomic Transaction and Send Web Services Atomic Transaction on requests, described earlier in this topic. Also, when the WS-Transaction policy type is included, the service runs under any received WS-AtomicTransaction context, in a similar way to the deployment descriptor Execute using Web Services Atomic Transaction on incoming requests, described earlier in this topic.

7. For EJB components only, for container-managed transactions, configure how the container manages the transaction boundaries when delegating a method invocation to the business method of an enterprise bean:
   a. In the deployment descriptor notebook, select the **Assembly** tab. The Container Transactions section displays a table of the methods for enterprise beans.
   b. For each method of the enterprise bean, set the container transaction type to an appropriate value. The default value for the container transaction type is Required, meaning that the method invocation occurs in the context of a transaction. This transaction is either the (local or remote)

client component's transaction or, if the client component does not run in a transaction, a new transaction started by the component's container.

If the application uses ActivitySessions, how the container manages transaction boundaries when delegating a method invocation depends on both the container transaction type that you set in this task, and the ActivitySession kind attribute, which is described in "Setting EJB module ActivitySession deployment attributes" on page 1512. For more detail about the relationship between these two properties, see "ActivitySession and transaction container policies in combination" on page 1501.

8. For Web services applications that use a SOAP/JMS binding and participates in WS-AtomicTransactions, set the container transaction type of the message-driven bean named "JMS router MDB" to a value of NotSupported, as described in the previous step. Web service applications that use a SOAP/JMS binding include a router message-driven bean named "JMS router MDB" in the assembled EAR. If a Web service uses a SOAP/JMS binding and participates in WS-AtomicTransactions, as described in Web Services Atomic Transaction support in the application server, set the container transaction type of the "JMS router MDB" to a value of NotSupported.

   For Web services applications that use a SOAP/HTTP binding and participate in WS-AtomicTransactions, you do not need to do this.
9. For client application components only, if required, enable support for transaction demarcation by the client. In the deployment descriptor notebook, select the **Allow JTA demarcation** check box. This option directs the client container to bind the Java Transaction API (JTA) UserTransaction interface into JNDI at java:comp/UserTransaction for the client component. There are constraints on transaction support in the client container, which are described in "Client support for transactions" on page 1460.
10. Save your changes to the deployment descriptor.
    a. Close the deployment descriptor editor.
    b. When prompted, click **Yes** to save changes to the deployment descriptor.
11. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
12. From the menu of the project, click **Deploy** to generate EJB deployment code.
13. Optional: Test your completed module on an application server installation. Right-click a module, click **Run on Server**, and follow the instructions in the resulting wizard.

    Note: Use the **Run On Server** option for unit testing only. The assembly tool controls the application server installation, and when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool, for example the administrative console, to deploy the EAR file onto the application server that is to run the application.

Using component-managed transactions

This topic describes how to enable a session bean, servlet, or application client component to use component-managed transactions, to manage its own transactions directly instead of letting the container manage the transactions.

About this task

Note: Entity beans cannot manage transactions (so cannot use bean-managed transactions).

To enable a session bean, servlet, or application client component to use component-managed transactions, complete the following steps:

1. For session beans, set the **Transaction type** attribute in the component's deployment descriptor to `Bean`, as described in Setting transactional attributes in the deployment descriptor.

2. For application client components, enable support for transaction demarcation by setting the **Allow JTA Demarcation** attribute in the component's deployment descriptor, as described in Setting transactional attributes in the deployment descriptor.
3. Write the component code to actively manage transactions

   For stateful session beans, a transaction started in a given method does not need to be completed (that is, committed or rolled back) before completing that method. The transaction can be completed at a later time, for example on a subsequent call to the same method, or even within a different method. However, constructing the application so a transaction is begun and completed within the same method call is usually preferred, because it simplifies application debugging and maintenance.

   The following code extract shows the standard code required to obtain an object encapsulating the transaction context, and involves the following basic steps:
   - A javax.transaction.UserTransaction object is created by calling a lookup on ″java:comp/UserTransaction″.
   - The UserTransaction object is used to demarcate the boundary of a transaction by using transaction methods such as begin and commit as needed. If an application component begins a transaction, it must also complete that transaction either by invoking the commit method or the rollback method.

   Code example: Getting an object that encapsulates a transaction context

```
...
import javax.transaction.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
...
    public float doSomething(long arg1)throws NamingException {
        InitialContext initCtx = new InitialContext();
        UserTransaction userTran = (UserTransaction)initCtx.lookup(
           "java:comp/UserTransaction");
        ...
        //Use userTran object to call transaction methods
        userTran.begin ();
        //Do transactional work
        ...
        userTran.commit ();
        ...
    }
    ...
}
```

Managing active and prepared transactions using scripting

You can use scripting to manage active and prepared transactions that might need administrator action.

Before you begin

Before you start this task, the wsadmin tool must be running. See Starting the wsadmin scripting client for more information.

About this task

In normal circumstances, every effort is made to finish a transaction. However, because of RRS and native contexts finishing, finishing the transaction may not be possible. In this case, the transaction is marked rollback_only so that it rolls back at the next available window. In other situations, you might need to finish a transaction manually. For example, you might want to finish a transaction that is stuck polling a resource manager that you know will not become available again in the desired time frame.

Note: If you choose to complete a transaction on an application server, it is recorded as having completed in the transaction service logs for that server, so it is not eligible for recovery during server start up. If you complete a transaction, you are responsible for cleaning up any in-doubt transactions on the resource managers affected.

The TransactionService Managed Bean (MBean), see API documentation - Application programming interfaces (package: Public MBean Interfaces, class: TransactionService), is used to list transactions in various states by invoking one of the following methods:

- listOfTransactions: Lists all non-completed transactions. Under no circumstances should you attempt to alter the state of active transactions (using commit or rollback for example)
- listManualTransactions: Lists transactions awaiting administrative completion. You can choose to commit or rollback transactions in this state
- listRetryTransactions: Lists transactions with some resources being retried. You can choose to finish (abandon retrying) transactions in this state
- listHeuristicTransactions: Lists transactions that have completed heuristically. You can choose to clear the transaction from the list
- listImportedPreparedTransactions: Lists transactions that have been imported and prepared but not yet committed. You can choose to commit or rollback transactions in this state

Each entry in the returned list contains the following attributes:

- **Local Transaction Identifier**
- **Status**, which can be interpreted by calling getPrintableStatus on the Transaction MBean
- **Global Transaction Identifier**
- **Heuristic Outcome**, which can take one of the following values:
  - 8 (HEURISTIC_COMMIT)
  - 9 (HEURISTIC_ROLLBACK)
  - 10 (HEURISTIC_MIXED)
  - 11 (HEURISTIC_HAZARD)

The TransactionService MBean, see API documentation - Application programming interfaces (package: Public MBean Interfaces, class: TransactionService), can also used to gather more information about the properties of the transaction service, by obtaining the following attributes:

- **transactionLogDirectory** The directory into which the log file(s) used for transaction service are placed
- **totalTranLifetimeTimeout** The total lifetime of the transaction (in seconds) before the container rolls it back. This value applies to Container Managed Transaction (CMT) beans only
- **asyncResponseTimeout** The total lifetime of the transaction (in seconds) before the container rolls it back
- **enableFileLocking** Enables the use of file locks when opening the transaction service recovery log
- **enableProtocolSecurity** Causes transaction service protocol messages to be sent securely
- **maximumTransactionTimeout** The maximum transaction timeout allowed for imported transactions. This value applies to Container Managed Transaction (CMT) beans, Bean Managed Transaction (BMT) beans, and imported transactions
- **clientInactivityTimeout** The number of seconds a transaction can remain inactive before it is rolled back
- **heuristicRetryLimit** The maximum number of times to retry transaction completion
- **heuristicRetryWait** The number of seconds to wait between retrying transaction completion
- **httpProxyPrefix** The HTTP prefix for WS-Transaction port SOAP addresses
- **httpsProxyPrefix** The HTTPS prefix for WS-Transaction port SOAP addresses
- **propogatedOrBMTTranLifetimeTimeout** The number of seconds a transaction can remain inactive before it is rolled back

- **LPSHeuristicCompletion** The transaction completion action to be taken when the outcome is unknown

The Transaction MBean, see API documentation - Application programming interfaces (package: Public MBean Interfaces, class: Transaction), can be used to commit, rollback, finish or remove from the list of heuristically completed transactions depending on the transaction's state, by invoking one of the following methods:

- commit: Heuristically commits the transaction
- rollback: Heuristically rolls back the transaction
- finish: Abandons retrying resources for the transaction
- removeHeuristic: Clears the transaction from the list

The Transaction MBean, see API documentation - Application programming interfaces (package: Public MBean Interfaces, class: Transaction), can also be used to gather more information about a transaction, by invoking the following methods:

- getPrintableStatus: Return the transaction status
- getGlobalTranName: Get the global identifier for the transaction
- listResources: List the resources for the transaction

The following script is an example of how to use the TransactionService MBean and Transaction MBean. The script should be run only against an application server, and not against the deployment manager or node agent.

Example

Working with manual transactions: example jacl script.

```
# get the TransactionService MBean
set servicembean [$AdminControl queryNames type=TransactionService,*]

# get the Transaction MBean
set mbean [$AdminControl queryNames type=Transaction,*]

set input 0
while {$input >= 0} {
        # invoke the listManualTransactions method
        set tranManualList [$AdminControl invoke $servicembean listManualTransactions]

        if {[llength $tranManualList] > 0} {
                puts "----Manual Transaction details---------------"
                set index 0
                foreach tran $tranManualList {
                        puts "   Index= $index tran= $tran"
                        incr index
                }
                puts "----End of Manual Transactions ---------------"
                puts "Select index of transaction to commit/rollback:"
                set input [gets stdin]
                if {$input < 0} {
                        puts "No index selected, exiting."
                } else {
                        set tran [lindex $tranManualList $input]
                        set commaPos [expr [string first "," $tran ]-1]
                        set localTID [string range $tran 0 $commaPos]
                        puts "Enter c to commit or r to rollback Transaction $localTID"
                        set input [gets stdin]
                        if {$input=="c"} {
                                puts "Committing transaction=$localTID"
                                $AdminControl invoke $mbean commit $localTID
                        }
                        if {$input=="r"} {
                                puts "Rolling back transaction=$localTID"
                                $AdminControl invoke $mbean rollback $localTID
                        }
                }
        } else {
                puts "No Manual transactions found, exiting"
                set input -1
```

```
        }
        puts " "

}
```

Working with manual transactions: example Jython script.

```
import sys
def wsadminToList(inStr):
        outList=[]
        if (len(inStr)>0 and inStr[0]=='[' and inStr[-1]==']'):
                tmpList = inStr[1:-1].split(" ")
        else:
                tmpList = inStr.split("\n")  #splits for Windows or Linux
        for item in tmpList:
                item = item.rstrip();        #removes any Windows "\r"
                if (len(item)>0):
                        outList.append(item)
        return outList
#endDef

servicembean = AdminControl.queryNames("type=TransactionService,*" )
mbean = AdminControl.queryNames("type=Transaction,*" )
input = 0

while (input >= 0):
        tranList = wsadminToList(AdminControl.invoke(servicembean, "listManualTransactions" ))

        tranLength = len(tranList)
        if (tranLength > 0):
                print "----Manual Transaction details---------------"
                index = 0
                for tran in tranList:
                        print "   Index=" , index , " tran=" , tran
                        index = index+1
                #endFor
                print "----End of Manual Transactions ---------------"
                print "Select index of transaction to commit/rollback:"
                input = sys.stdin.readline().strip()
                if (input == ""):
                        print "No index selected, exiting."
                        input = -1
                else:
                        tran = tranList[int(input)]
                        commaPos = (tran.find(",") -1)
                        localTID = tran[0:commaPos+1]
                        print "Enter c to commit or r to rollback transaction ", localTID
                        input = sys.stdin.readline().strip()
                        if (input == "c"):
                                print "Committing transaction=", localTID
                                AdminControl.invoke(mbean, "commit", localTID )
                        #endIf
                        elif (input == "r"):
                                print "Rolling back transaction=", localTID
                                AdminControl.invoke(mbean, "rollback", localTID )
                        #endIf
                        else:
                                input = -1
                        #endelse
                #endElse
        else:
                print "No transactions found, exiting"
                input = -1
        #endElse
        print " "

#endWhile
```

Using one-phase and two-phase commit resources in the same transaction

Use these topics to help you coordinate the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction.

About this task

You can coordinate the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction. You can have multiple interactions that involve the one-phase commit resource in the same transaction, but only one such resource can be involved. This coordination is enabled by the *last participant support*.

At transaction commit, the two-phase commit resources are prepared first using the two-phase commit protocol, and if this is successful the one-phase commit-resource is then called to commit. The two-phase commit resources are then committed or rolled back depending on the response of the one-phase commit resource.

For more information about using one-phase and two-phase commit resources within the same transaction, see the following topics:

- "Approaches to coordinating access to one-phase commit and two-phase commit capable resources in the same transaction"
- "Assembling an application to use one-phase and two-phase commit resources in the same transaction" on page 1491
- "Configuring an application server to log heuristic reporting" on page 1493

Approaches to coordinating access to one-phase commit and two-phase commit capable resources in the same transaction

Last participant support enables the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction. You can have multiple interactions that involve the one-phase commit resource in the same transaction, but only one such resource can be involved.

At transaction commit, the two-phase commit resources are prepared first using the two-phase commit protocol, and if this is successful, the one-phase commit-resource is then called to commit. The two-phase commit resources are then committed or rolled back, depending on the response of the one-phase commit resource.

Note: If the global transaction is distributed across multiple application servers *that are all running at WebSphere Application Server version 5.1 or later*, you can exploit last participant support to coordinate a one-phase commit capable resource and any number of two-phase commit capable resources in the same transaction, in a limited number of scenarios.

- The main scenario is where the one-phase commit resource provider is accessed in the application server process (the "transaction root" server) in which the transaction is started.

  In this scenario, last participant support can coordinate a one-phase commit capable resource and any number of two-phase commit capable resources in the same transaction.
- If the one-phase commit resource provider is accessed in a different application server (a "transaction subordinate" server) from the one in which the transaction was started; for example, as a result of a transactional invocation on a remote EJB interface where the EJB implementation accesses a one-phase commit resource provider.

  In this scenario, the transaction typically cannot be committed. To be able to commit (as part of a global transaction) a one-phase commit resource enlisted on a transaction subordinate server, the transaction service must delegate coordination responsibility from the transaction root to the subordinate server. This occurs only if no other resources were registered with the transaction root server.

Last participant support introduces an increased risk of an heuristic outcome to the transaction. That is, the transaction manager cannot be sure that all resources were completed in the same direction (either committed or rolled back). For this reason, to enable an application to coordinate access to one-phase and

two-phase commit capable resources in the same transaction, you configure the application to accept the heuristic hazard, that is, accept the increased risk of an heuristic outcome.

An heuristic outcome occurs if the transaction service (JTS) receives no response from the commit one-phase flow on the one-phase commit resource. In this situation, the transaction service cannot determine whether changes for the one-phase commit resource were committed or rolled back, so cannot drive reliably the correct outcome of the global transaction on the other two-phase commit resources.

You can configure the transaction service for an application server to accept the heuristic hazard, or you can configure applications individually to accept the heuristic hazard. You can configure applications individually either when they are assembled, or after they are deployed.

You can configure the transaction service for an application server to indicate whether or not to log that it is about to commit the one-phase commit resource. This does not reduce the heuristic hazard, but ensures that any failure, and subsequent recovery, of the application server during the one-phase commit phase occurs with knowledge of whether or not the one-phase commit resource was asked to commit:

- If the one-phase commit resource was asked to commit, a heuristic outcome is reported to the activity log.
- If the one-phase commit resource was not asked to commit, then the transaction is rolled back consistently.

Assembling an application to use one-phase and two-phase commit resources in the same transaction

Use this task to assemble an application to use one-phase and two-phase commit resources in the same transaction.

Before you begin

This task description assumes that you have an EAR file for an application component, that can be deployed in WebSphere Application Server. For more details about assembling applications, see Assembling applications.

About this task

To enable an application to use one-phase and two-phase commit capable resources in the same transaction, you must configure the deployment attributes of the application to accept the heuristic hazard, that is, the increased risk of an heuristic outcome. You can configure the deployment attributes of an application by using an assembly tool.

You can also configure an application to accept the heuristic hazard after deployment, by using the administrative console and the Last participant support extension settings. Alternatively, you can configure the transaction service for an application server to accept the heuristic hazard.

This topic describes the use of Rational Application Developer to configure the deployment attributes of an application.

To configure an application to indicate that you accept the increased risk of an heuristic outcome, complete the following steps:

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the application EAR file.

   Note: Ensure that you set the target server as WebSphere Application Server Version 7.0.

   For example, to change attributes of an existing application, use the Import wizard to import the EAR file into the assembly tool. To start the Import wizard:

a. Click **File** → **Import** → **EAR file**.
b. Click **Next**, then select the EAR file.
c. In the Target server field, select `WebSphere Application Server v7.0`.
d. Click **Finish**.

3. In the Project Explorer view of the Java EE perspective, complete the following steps:
   a. Expand the Enterprise Application instance.
   b. Right click on the Deployment Descriptor.
   c. Click **Open With** → **Deployment Descriptor Editor**.

   A property dialog notebook for the component is displayed in the property pane.
4. Complete the following steps to display the Extended Services tab.
   a. Close the Enterprise Application Deployment Descriptor editor.
   b. In the toolbar, select **Windows** → **Preferences**.
   c. In the left pane, select **Capabilities**.
   d. In the right pane, expand **Advanced Java EE** and select the **WebSphere PME Development** option.
   e. Click **Apply**.
   f. Open the Enterprise Application Deployment Descriptor editor.
5. On the Extended Services tab, in the Last Participant Support section, select the **Last participant support** check box.
6. Save your changes to the deployment descriptor.
   a. Close the Deployment Descriptor Editor.
   b. When prompted, click **Yes** to save changes to the deployment descriptor.
7. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
8. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
9. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

   Note: Use **Run On Server** only for unit testing. The assembly tool controls the WebSphere Application Server installation and, when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console as described in Deploying and administering enterprise applications.

Last participant support extension settings

Use this page to configure settings for last participant support. Last participant support is an extension to the transaction service that enables a single one-phase resource to participate in a two-phase transaction with one or more two-phase resources. Values on this panel are ignored if you select **Use configuration information in binary** on the Application binaries panel.

To view this administrative console page, click **Applications** > **Enterprise Applications** > *application_name* > **[Detail Properties] Last participant support extension**

Accept heuristic hazard:

Specifies whether an application accepts the possibility of a heuristic hazard occurring in a two-phase transaction that contains a one-phase resource.

Default
Cleared

Range

Selected
The application accepts the increased risk of an heuristic outcome.

Cleared
The application does not accept the increased risk of an heuristic outcome.

Configuring an application server to log heuristic reporting

About this task

To enable an application server to log “about to commit one-phase resource” events from transactions that involve a one-phase commit resource and two-phase commit resources, use the Administrative console to complete the following steps:

1. Start the Administrative console
2. In the navigation pane, select **Servers-> Manage Application Servers-> *your_app_server*** This displays the properties of the application server, *your_app_server*, in the content pane.
3. Select the Transaction Service tab, to display the properties page for the transaction service, as two notebook pages:
   Configuration
   The values of properties defined in the configuration file. If you change these properties, the new values are applied when the application server next starts.
   Runtime
   The runtime values of properties. If you change these properties, the new values are applied immediately, but are overwritten with the Configuration values when the application server next starts.
4. Select the Configuration tab, to display the transaction-related configuration properties.
5. Select the **Enable logging for heuristic reporting** checkbox.
6. Click **OK**.
7. Stop then restart the application server.

Transaction exceptions that involve both single- and two-phase commit resources

The exceptions that can be thrown by transactions that involve single- and two-phase commit resources are the same as those that can be thrown by transactions involving only two-phase commit resources.

The exceptions that can be thrown are listed in the Reference: Generated API documentation found in the Reference section of the WebSphere Application Server Version 6.1 Information Center.

Last Participant Support: Resources for learning

Use the links in this topic to find relevant supplemental information about Last Participant Support. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming specifications

- J2EE Activity Service for Extended Transactions
- Java Transaction API (JTA) 1.0.1

Other

- WebSphere Business Integration Server Foundation
- List of IBM WebSphere Redbooks
- WebSphere technical library, including links to white papers

Chapter 15. Learn about WebSphere programming extensions

Use this section as a starting point to investigate the WebSphere programming model extensions for enhancing your application development and deployment.

See the *Developing and deploying applications* PDF book for a brief description of each WebSphere extension.

Your applications can use the Eclipse extension framework. Your applications are extensible as soon as you define an extension point and provide the extension processing code for the extensible area of the application. You can also plug an application into another extensible application by defining an extension that adheres to the target extension point requirements. The extension point can find the newly added extension dynamically and the new function is seamlessly integrated in the existing application. It works on a cross Java Platform, Enterprise Edition (Java EE) module basis.

The application extension registry uses the Eclipse plug-in descriptor format and application programming interfaces (APIs) as the standard extensibility mechanism for WebSphere applications. Developers that build WebSphere application modules can use WebSphere Application Server extensions to implement Eclipse tools and to provide plug-in modules to contribute functionality such as actions, tasks, menu items, and links at predefined extension points in the WebSphere application.

ActivitySessions

Using the ActivitySession service

This topic is an overview of the tasks involved in implementing WebSphere enterprise applications that use ActivitySessions.

About this task

The ActivitySession service provides an alternative unit-of-work scope to the scope that is provided by global transaction contexts. ActivitySessions provide a scoping mechanism for units of work, and both an ActivitySession and a transaction have the same following characteristics:

- They can be bean-managed or container-managed
- They can be distributed across application servers
- They can be used as the context for managing EJB activation policy and lifecycle

An ActivitySession differs significantly from a transaction in the manner of its interaction with resource managers. An ActivitySession is used to scope or coordinate local transactions. That is, an ActivitySession can be used to request multiple one-phase resource managers to come to an application- or container-determined outcome. Unlike a transaction, an ActivitySession has no notion of a prepare phase or any notion of recovery at a service level.

The WebSphere EJB container and deployment tools support ActivitySessions as an extension to the Java platform for enterprise applications programming model. Enterprise beans can be deployed with lifecycles that are influenced by ActivitySession context, as an alternative to transaction context. An enterprise bean with an ActivitySession-scoped lifecycle can participate in a resource manager local transaction (RMLT) that has a duration of the ActivitySession rather than an individual method on the bean (which is all that is possible under the standard Java platform for enterprise applications model). Applications can then be composed of several enterprise beans with ActivitySession-based activation, with each bean participating in extended local transactions with one or more resource managers. At the end of the ActivitySession each of the local transactions can be directed to a common outcome by the ActivitySession manager.

You can configure the WebSphere containers and deployable applications to support enterprise beans that operate under application- or container-initiated ActivitySessions rather than, or in addition to, transactions.

Use these tasks to implement WebSphere enterprise applications that use ActivitySessions:

- “Developing an enterprise application to use ActivitySessions” on page 1509
- “Developing an enterprise bean or enterprise application client to manage ActivitySessions” on page 1511
- “Setting EJB module ActivitySession deployment attributes” on page 1512
- Disabling or enabling the ActivitySession service
- Configuring the default ActivitySession timeout for an application server

What to do next

For more information about implementing WebSphere enterprise applications that use ActivitySessions, see the following topics:

- “The ActivitySession service”
  - “ActivitySession and transaction contexts” on page 1500
  - “Usage model for using ActivitySessions with HTTP sessions” on page 1498
- “ActivitySession service application programming interfaces” on page 1507
- “Samples: ActivitySessions” on page 1508
- “Setting Web module ActivitySession deployment attributes” on page 1514
- Troubleshooting ActivitySessions

The ActivitySession service

The ActivitySession service provides an alternative unit-of-work (UOW) scope to that provided by global transaction contexts. An ActivitySession context can be longer-lived than a global transaction context and can encapsulate global transactions.

Support for the ActivitySession service is shown in the following figure:

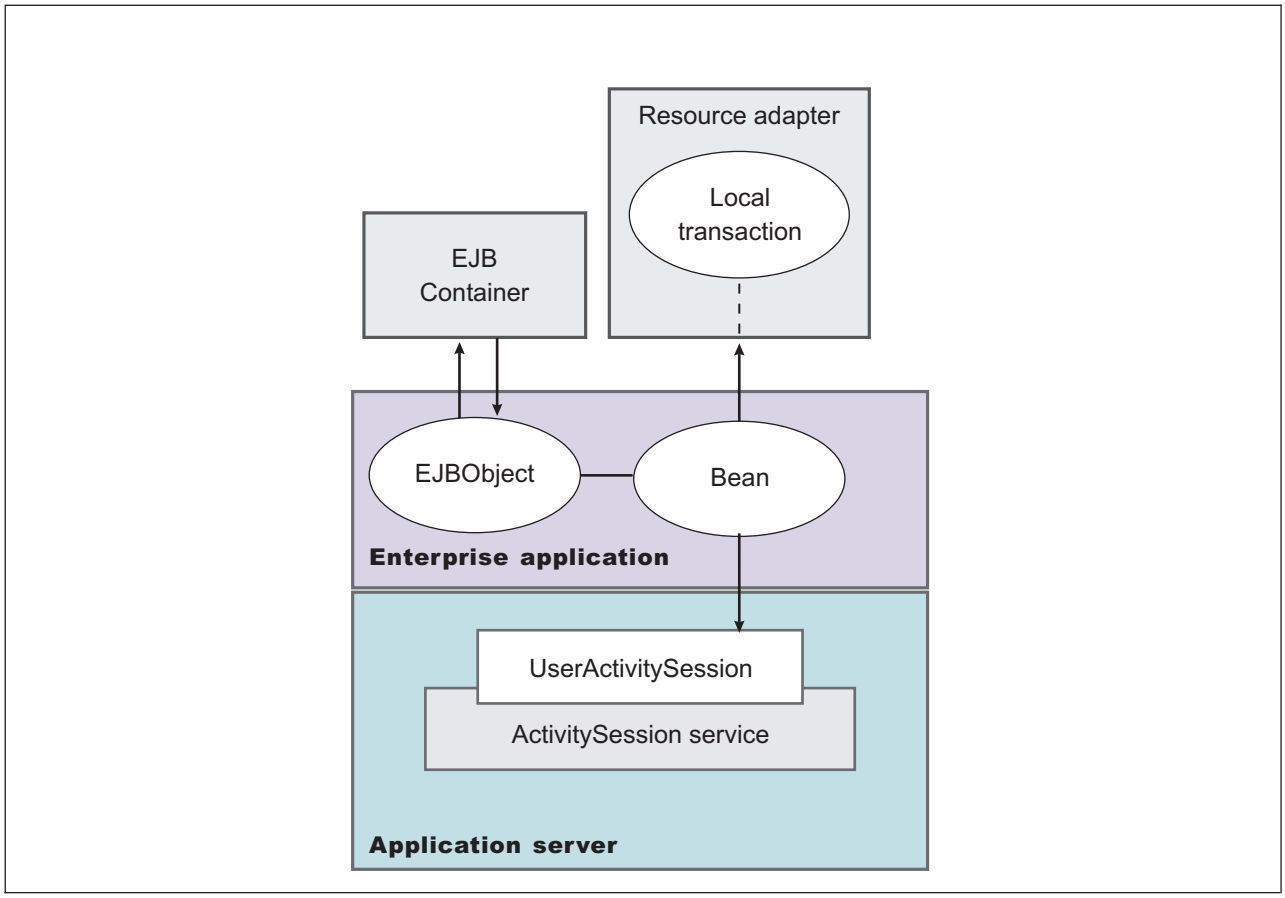


Figure 17. The ActivitySession service. This figure shows the main components of the ActivitySession service within WebSphere Application server. For an overview of these components, see the text that accompanies this figure.

Although the purpose of a global transaction is to coordinate multiple resource managers, global transaction context is often used by enterprise applications as a “session” context through which to access EJB instances. An ActivitySession context is such a session context, and can be used in preference to a global transaction in cases where coordination of two-phase commit resource managers is not needed. Further, an ActivitySession can be associated with an HttpSession to extend a “client session” to an HTTP client.

ActivitySession support is available to Web, EJB, and Java platform for enterprise applications client components. EJB components can be divided into beans that exploit container-managed ActivitySessions and beans that use bean-managed ActivitySessions.

The ActivitySession service provides a UserActivitySession application programming interface available to enterprise application components that use bean-managed ActivitySessions for application-managed demarcation of ActivitySession context. The ActivitySession service also provides a system programming interface for container-managed demarcation of ActivitySession context and for container-managed enlistment of one-phase resources (RMLTs) in such contexts.

The UserActivitySession interface is obtained by a JNDI lookup of java:comp/websphere/UserActivitySession. This interface is not available to enterprise beans that use container-managed ActivitySessions, and any attempt by such beans to obtain the interface results in a NotFound exception.

A common scenario is an enterprise application accessing one or more enterprise beans backed by non-transactional (one-phase commit) resources. The application, or its container, uses the UserActivitySession interface to define the demarcation boundaries within which operations against the enterprise beans are grouped and to control whether those grouped operations should be checkpointed or

discarded. The business logic of the enterprise beans does not need to use any ActivitySession interfaces. The container into which the enterprise beans are deployed ensures that updates to the underlying one-phase resource managers are coordinated.

The application can checkpoint an ActivitySession to create a new point of consistency within the ActivitySession without ending the ActivitySession. The application can also use a reset operation to return work performed in the ActivitySession back to the last point of consistency. The application can end the ActivitySession with an operation to either checkpoint or reset all resources.

Usage model for using ActivitySessions with HTTP sessions:

This topic describes how a Web application that runs in the WebSphere Web container can participate in an ActivitySession context.

If the Web application is designed such that several servlet invocations occur as part of the same logical application, then the servlets can use the HttpSession to preserve state across servlet invocations. The ActivitySession context is one state that can be suspended into the HttpSession and resumed on a future invocation of a servlet that accesses the HttpSession.

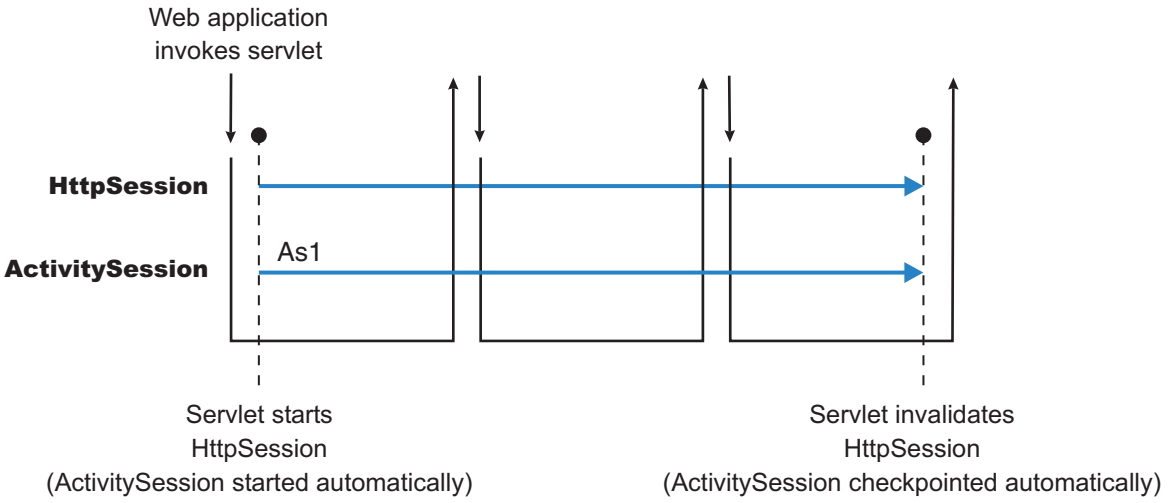
An ActivitySession is associated automatically with an HttpSession, so can be used to extend access to the ActivitySession over multiple HTTP invocations, over inclusion or forwarding of servlets, and to support Enterprise JavaBeans (EJB) activation periods that can be determined by the lifecycle of the Web HTTP client. An ActivitySession context stored in an HttpSession can also be used to relate work for the ActivitySession back to a specific Web HTTP client.

The Web container manages ActivitySessions based on deployment descriptor attributes associated with servlets in the Web application module. The two usage models are:

- The Web container starts and ends ActivitySessions.

  The Web application invokes a servlet that has been configured for container control of ActivitySessions.
  - If an HttpSession exists then it has an associated ActivitySession.
  - If an HttpSession does not exist, the servlet can start an HttpSession, which causes an ActivitySession to be started automatically and associated with the HttpSession.

  A servlet cannot start a new HttpSession until an existing HttpSession has been ended. Within an HttpSession, the Web application can invoke other servlets that can use the associated ActivitySession context. When the Web application invokes a servlet that ends the HttpSession, the ActivitySession is ended automatically. This is shown in the following diagram:

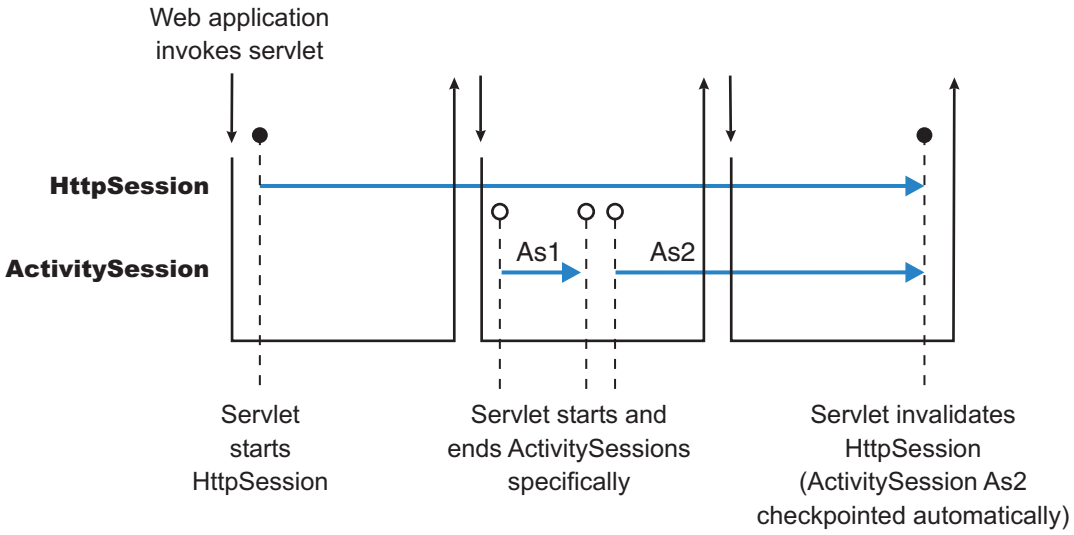


- The Web application starts and ends ActivitySessions.

The Web application invokes a servlet that has been configured for application control of ActivitySesions.
- If an HttpSession exists and has an associated ActivitySession, the servlet can use or end that ActivitySession context.
- If an HttpSession does not exist, the servlet can start an HttpSession, but this does not automatically start an ActivitySession.
- If an HttpSession exists but does not have an associated ActivitySession, the servlet can start a new ActivitySession. This automatically associates the ActivitySession with the HttpSession. The ActivitySession lasts either until the ActivitySession is specifically ended or until the HttpSession is ended.

The servlet cannot start a new ActivitySession until an existing ActivitySession has been ended. The servlet cannot start a new HttpSession until an existing HttpSession has been ended.

Within an HttpSession, the Web application can invoke other servlets that can use or end an existing ActivitySession context or, if no ActivitySession exists start a new ActivitySession. When the Web application invokes a servlet that ends the HttpSession, the ActivitySession is ended automatically. This is shown in the following diagram:



A Web application can invoke servlets configured for either usage model.

The following points apply to both usage models:
- To end an HttpSession (and any associated ActivitySession), the Web application must invalidate that session. This causes the ActivitySession to be checkpointed.
- Any downstream enterprise beans activated within the context of an ActivitySession can be held in memory rather than passivated between servlet invocations, because the client effectively becomes the Web HTTP client.
- Web applications can be composed of many servlets, and each servlet in the Web application can be configured with a value for ActivitySessionControl. ActivitySessionControl determines whether the servlet or its container starts any ActivitySessions.
- An ActivitySession context that encapsulates an active transaction context cannot be associated with an HttpSession, because a transaction can hold database locks and should be designed to be shortlived. If an application moves an active transaction to an HttpSession, the transaction is rolled back and the ActivitySession is suspended into the HTTPSession. In general, you should design applications to use ActivitySessions or other constructs as the long-lived entities and ACID transactions as short-duration entities within these.
- Only one ActivitySession can be associated with an HttpSession at any time, for the duration of the ActivitySession. An ActivitySession associated with an HttpSession remains associated for the duration

of that ActivitySession, and cannot be replaced with another until the first ActivitySession is completed. The ActivitySession can be accessed by multiple servlets if they have shared access to the HttpSession.
- ActivitySessions are not persistent. If a persistent HttpSession exists longer than the server hosting it, any cached ActivitySession is terminated when the hosting server ends.
- If the HttpSession times out before the associated ActivitySession has ended, then the ActivitySession is reset[1]. This rolls back the ActivitySession resources to the last point of consistency:
  - If the Web application invoked a servlet that has been configured for container control of ActivitySessions, the ActivitySession resources are rolled back completely.
  - If the Web application invoked a servlet that has been configured for application control of ActivitySessions, the ActivitySession resources are rolled back to the last checkpoint taken by the servlet, or completely if no checkpoint has been taken.
- If the ActivitySession times out, it is reset to the last point of consistency (see previous item), then the HttpSession is ended.

ActivitySession and transaction contexts:

This topic describes the hierarchical relationship between transaction and ActivitySession contexts. This relationship, defined by the ActivitySession service, requires that any transaction context be either wholly inside or wholly outside an ActivitySession context.

An ActivitySession context is very similar to a transaction context and extends the lifecycle choices for activation of enterprise beans; it can encapsulate one or more transactions. The ActivitySession context is a distributed context that, like the transaction context, can be bean- or container-managed. An ActivitySession context is used mainly by a client to scope the lifecycle of an enterprise bean that it uses either beyond or in the absence of individual transactions started by that client.

ActivitySessions have a lower overhead than transactions and can be used instead of transactions that are only used to scope the lifecycle of a called enterprise bean. For a bean with an activation policy of ActivitySession, the duration of any resource manager local transactions (RMLTs) started by that bean can be bounded by the duration of the ActivitySession instead of the bean method in which the RMLT was started. This provides flexibility and potential for using RMLTs in an enterprise bean beyond the scenarios described in the Enterprise JavaBeans (EJB) specifications. The EJB specifications define that RMLTs need to be completed before the end of the bean method, because the bean method is the only containment boundary for local transactions available in those specifications.

The following rules defines the relationship between transactions and ActivitySessions.
- The EJB or Web container always uses a local transaction containment (LTC) if there is no global transaction present. An LTC can be method-scoped or ActivitySession-scoped.
- Before a method dispatch, the container ensures that there is always either an LTC or global transaction context, but never both contexts.
- ActivitySessions cannot be nested within each other. Any attempt to start a nested ActivitySession results in a com.ibm.websphere.ActivitySession.NotSupportedException on UserActivitySession.beginSession().
- An ActivitySession can wholly encapsulate one or more global transactions.
- The application can end an ActivitySession with an operation to either checkpoint or reset all resources. The endSession(EndModeCheckpoint) operation checkpoints the work coordinated under the ActivitySession then ends the context. The endSession(EndModeReset) operation resets, to the last point of consistency, the work coordinated under the ActivitySession then ends the context.
- An ActivitySession cannot be encapsulated by a global transaction nor should ActivitySession and global transaction boundaries overlap. Any attempt to start an ActivitySession in the presence of a global

1. Resetting an ActivitySession causes all the resources involved in the current ActivitySession to be rolled back to the last point of consistency, but allows further work within the ActivitySession. When the reset completes, the thread is associated with the same ActivitySession as it was before the reset was called. The ActivitySession resources remain associated with the ActivitySession although they cannot participate further in the ActivitySession

transaction context results in a com.ibm.websphere.ActivitySession.NotSupportedException on UserActivitySession.beginSession(). Any attempt to call endSession(EndModeCheckpoint) on an ActivitySession that contains an incomplete global transaction results in a com.ibm.websphere.ActivitySession.ContextPendingException. Neither the global transaction nor the ActivitySession context are affected. If endSession(EndModeReset) is called then the ActivitySession is reset and the global transactions marked rollback_only.
- Each global transaction wholly encapsulated by an ActivitySession is independent of every other global transaction within that ActivitySession. A rollback of one global transaction does not affect any others or the ActivitySession itself.
- ActivitySession and global transaction contexts can coexist with an ActivitySession encapsulating one or more serially-running global transactions.

ActivitySession and transaction container policies in combination:

This topic provides details about the relationship between the deployment descriptor properties that determine how the container manages ActivitySession boundaries.

If an enterprise bean uses ActivitySessions, how the EJB container manages ActivitySession boundaries when delegating a method invocation depends on both the **ActivitySession kind** and **Container transaction type** deployment descriptor attributes configured for the enterprise bean. The following table lists the relationship between these two properties.

In each row, the final column describes the behavior that the EJB container takes with respect to global transaction and ActivitySession context, based on the following abbreviations:
S*n* An ActivitySession, where *n* indicates the ActivitySession instance.
T*n* A transaction, where *n* indicates the transaction instance.

In every case where the container does not start or leave a global transaction context associated with the thread, it starts (or obtains from the bean instance) a local transaction containment and associates that with the thread. The duration of the local transaction containment is determined by a combination of the local-transaction boundary descriptor (configured as part of the application deployment descriptor, and not shown in the following table) and the presence or not of an ActivitySession context, as described in ActivitySessions and transaction contexts.

The rows highlighted in bold are not allowed.

Table 39. Container behavior for activitysession and transaction policies deployment settings

Bean ActivitySession policy(ActivitySession kind)	Bean transaction policy(Container transaction type)	Received contexts	Container behavior
Required	Required	None	Start S1, Start T1
		S1	Start T1
		T1	Suspend T1, Start S1, Start T2
		S1, T1	No Action
	Requires new	None	Start S1, Start T1
		S1	Start T1
		T1	Suspend T1, Start S1, Start T2
		S1, T1	Suspend T1, Start T2
	Supports	None	Start S1
		S1	No Action
		T1	Suspend T1, Start S1
		S1, T1	No Action
	Not supported	None	Start S1
		S1	No Action
		T1	Suspend T1, Start S1
		S1, T1	Suspend T1
	Mandatory	None	Exception
		S1	Exception
		T1	Exception
		S1, T1	No action
	Never	None	Start S1
		S1	No Action
		T1	Suspend T1, Start S1
		S1, T1	Exception

Table 39. Container behavior for activitysession and transaction policies deployment settings (continued)

Bean ActivitySession policy(ActivitySession kind)	Bean transaction policy(Container transaction type)	Received contexts	Container behavior
Requires new	Required	None	Start S1 + T1
		S1	Suspend S1, Start S2 + T1
		T1	Suspend T1, Start S1 + T2
		S1 + T1	Suspend S1 + T1, Start S2 + T2
	Requires new	None	Start S1 + T1
		S1	Suspend S1, Start S2 + T1
		T1	Suspend T1, Start S1 + T2
		S1 + T1	Suspend S1 + T1, Start S2 + T2
	Supports	None	Start S1
		S1	Suspend S1, Start S2
		T1	Suspend T1, Start S1
		S1, T1	Suspend S1 + T1, Start S2
	Not supported	None	Start S1
		S1	Suspend S1, Start S2
		T1	Suspend T1, Start S1
		S1, T1	Suspend S1 + T1, Start S2
	Mandatory	**None**	**Exception**
		S1	**Exception**
		T1	**Exception**
		S1, T1	**Exception**
	Never	None	Start S1
		S1	Suspend S1, Start S2
		T1	Suspend T1, Start S1
		S1, T1	Suspend S1 + T1, Start S2

Table 39. Container behavior for activitysession and transaction policies deployment settings (continued)

Bean ActivitySession policy(ActivitySession kind)	Bean transaction policy(Container transaction type)	Received contexts	Container behavior
Supports	Required	None	Start T1
		S1	Start T1
		T1	No Action
		S1, T1	No Action
	Requires new	None	Start T1
		S1	Start T1
		T1	Suspend T1, Start T2
		S1, T1	Suspend T1, Start T2
	Supports	None	No Action
		S1	No Action
		T1	No Action
		S1, T1	No Action
	Not supported	None	No Action
		S1	No Action
		T1	Suspend T1
		S1, T1	Suspend T1
	Mandatory	None	Exception
		S1	Exception
		T1	No Action
		S1, T1	No Action
	Never	None	No Action
		S1	No Action
		T1	Exception
		S1, T1	Exception

Table 39. Container behavior for activitysession and transaction policies deployment settings (continued)

Bean ActivitySession policy(ActivitySession kind)	Bean transaction policy(Container transaction type)	Received contexts	Container behavior
Not supported	Required	None	Start T1
		S1	Suspend S1, Start T1
		T1	No Action
		S1, T1	Suspend S1 + T1, Start T2
	Requires new	None	Start T1
		S1	Suspend S1, Start T1
		T1	Suspend T1, Start T2
		S1, T1	Suspend S1 + T1, Start T2
	Supports	None	No Action
		S1	Suspend S1
		T1	No Action
		S1, T1	Suspend S1 + T1
	Not supported	None	No Action
		S1	Suspend S1
		T1	Suspend T1
		S1, T1	Suspend S1 + T1
	Mandatory	None	Exception
		S1	Exception
		T1	No Action
		S1,T1	Exception
	Never	None	No Action
		S1	Suspend S1
		T1	Exception
		S1, T1	Suspend S1 + T1

Table 39. Container behavior for activitysession and transaction policies deployment settings (continued)

Bean ActivitySession policy(ActivitySession kind)	Bean transaction policy(Container transaction type)	Received contexts	Container behavior
Mandatory	Required	None	Exception
		S1	Start T1
		T1	Exception
		S1, T1	No Action
	Requires new	None	Exception
		S1	Start T1
		T1	Exception
		S1, T1	Suspend T1, Start T2
	Supports	None	Exception
		S1	No Action
		T1	Exception
		S1, T1	No Action
	Not supported	None	Exception
		S1	No Action
		T1	Exception
		S1, T1	Suspend T1
	Mandatory	None	Exception
		S1	Exception
		T1	Exception
		S1, T1	No Action
	Never	None	Exception
		S1	No Action
		T1	Exception
		S1,T1	Exception

Table 39. Container behavior for activitysession and transaction policies deployment settings (continued)

Bean ActivitySession policy(ActivitySession kind)	Bean transaction policy(Container transaction type)	Received contexts	Container behavior
Never	Required	None	Start T1
		S1	Exception
		T1	No Action
		S1, T1	Exception
	Requires new	None	Start T1
		S1	Exception
		T1	Suspend T1, Start T2
		S1,T1	Exception
	Supports	None	No Action
		S1	Exception
		T1	No Action
		S1,T1	Exception
	Not supported	None	No Action
		S1	Exception
		T1	Suspend T1
		S1,T1	Exception
	Mandatory	None	Exception
		S1	Exception
		T1	No Action
		S1,T1	Exception
	Never	None	No Action
		S1	Exception
		T1	Exception
		S1,T1	Exception
Bean managed	Bean managed	None	No Action
		S1	Suspend S1
		T1	Suspend T1
		S1, T1	Suspend S1 + T1

ActivitySession service application programming interfaces

The ActivitySession service consists of an application programming interface available to Web applications, session Enterprise JavaBeans (EJBs), and Java platform for enterprise applications client applications for application-managed demarcation of ActivitySession context.

Applications use the UserActivitySession interface, which provides demarcation scope methods.

ActivitySession API

The ActivitySession service provides the UserActivitySession interface for use by EJB Session beans using bean-managed context demarcation, Web application components configured with **ActivitySession control**=`Web Application`, and Java platform for enterprise applications client applications. This UserActivitySession interface defines the set of ActivitySession operations available to an application component. You can obtain an implementation of this interface by using a JNDI lookup of the URL ″java:comp/websphere/UserActivitySession″. The UserActivitySession interface is used to begin and end ActivitySessions and to query various attributes of the active ActivitySession that is associated with the thread.

For more information about the ActivitySession API, see WebSphere Application Server application programming interface reference information.

The ActivitySession API and the implementation of its interfaces is contained in the com.ibm.websphere.ActivitySession package.

Programming Examples

The following code extract provides a basic example of using the UserActivitySession interface:

```
// Get initial context
  InitialContext ic = new InitialContext();
// Lookup UserActivitySession
  UserActivitySession uas = (UserActivitySession)ic.lookup("java:comp/websphere/UserActivitySession");

// Set the ActivitySession timeout to 60 seconds
  uas.setSessionTimeout(60);
// Start a new ActivitySession context
  uas.beginSession();
// Do some work under this context
  MyBeanA beanA.doSomething();
  ...
  MyBeanB beanB.doSomethingElse();
// End the context
  uas.endSession(EndModeCheckpoint);
```

Samples: ActivitySessions

This topic describes the ActivitySession samples provided with WebSphere Application Server.

MasterMind sample

This sample is based on the game MasterMind. It consists of the following components:

- A servlet, configured with ActivitySession contol set to Container, that accesses a stateful session bean.
- A stateful session bean, configured with an activation policy of ActivitySession containing transient state data.

The servlet begins an HttpSession at the start of each new game, and ends it at the end of each game; therefore an ActivitySession lasts for the duration of each game. The ActivitySession activation policy stops the bean from being passivated and therefore the transient data remains in memory. This is to demonstrate HttpSession/ActivationSession association in the web container, and an ActivitySession-scoped activation policy.

Enterprise application client container and a CMP entity bean backed by a one-phase commit datasource

In this sample, the entity bean is configured with the following properties:

- TX_NOT_SUPPORTED
- An ActivitySession container managed policy of REQUIRES
- An LTC boundary of ActivitySession
- An LTC Resolution Control of ContainerAtBoundary

The client accesses the UserActivitySession, begins an ActivitySession, updates two instances of the bean, then ends the ActivitySession. It does this twice using EndModeReset then EndModeCheckpoint. This sample demonstrates the following functionality:

- Client access to the UserActivitySession interface
- Multiple RMLTs being scoped to the ActivitySession and automatically taking their completion direction from that of the ActivitySession

The entity bean also holds a transient variable incremented by each method call (gets and sets for the persistent data). This value is checked before the end of the ActivitySession to show that the same bean instance is used. The client checks for the correct results.

An enterprise application client container and two session beans with different ActivitySession types

This sample consists of an enterprise application client container and the following session beans:

- SLB1, a stateless session bean configured with an ActivitySession Type of Bean.

- SFB2, a stateful session bean configured with ActivitySession Type of Requires, an LTC boundary of ActivitySession, LTC Resolution Contol of APPLICATION, and an LTC Unresolved Action of ROLLBACK.

Both beans are configured with TX_NOTSUPPORTED.

This sample performs the following steps:
1. The client starts SLB1
2. SLB1 accesses the UserActivitySession interface, begins an ActivitySession, then calls a method on SFB2
3. SFB2 accesses the UserActivitySession interface, begins an ActivitySession, calls a method on SFB2
4. SFB2 gets a connection (setAutoCommit false) then uses JDBC to update a single-phase datasource.
5. SLB1 then optionally calls a seperate method on SFB2 to finish the work either committing or rolling-back the RMLT.
6. SLB1 then ends the ActivitySession with an EndModeCheckpoint.

This sample demonstrates the following functionality:
- The ActivitySession completion direction is unconnected to the direction of the RMLTs, although the RMLTs containment is bound to the ActivitySession.
- The container using the unresolved action when an RMLT is not completed.
- A bean-managed ActivitySessions bean using the UserActivitySession interface.

The sample checks for correct results and reports them back to the client.

ActivitySession service: Resources for learning

Use the links in this topic to find relevant supplemental information about ActivitySessions. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming model and decisions

- WebSphere Application Server application programming interface reference information

Programming specifications

- J2EE Activity Service for Extended Transactions
- Java Transaction API (JTA) 1.0.1

Other

- WebSphere Business Integration Server Foundation
- List of IBM WebSphere Redbooks
- WebSphere technical library, including links to white papers

Developing an enterprise application to use ActivitySessions

This topic provides an overview of the high-level tasks for using ActivitySessions in enterprise applications.

About this task

You should consider the following points before using ActivitySessions in enterprise applications:
- An application that is accessed under an ActivitySession context can receive a javax.transaction.InvalidTransactionException RemoteException, thrown by the Enterprise JavaBeans

(EJB) container when servicing any application method. This exception occurs when an instance of an enterprise bean that has an ActivitySession-based activation policy becomes involved with concurrent global and local transactions.

- To enable an enterprise bean to participate in an ActivitySession context and support ActivitySession-based operations, it must be configured with an ActivationPolicy of ACTIVITY_SESSION. A bean configured with ActivationPolicy of either TRANSACTION or ONCE cannot participate in an ActivitySession context.
- A session bean can either use container-managed ActivitySessions or implement bean-managed ActivitySessions; entity beans can only use container-managed ActivitySessions. A bean is deployed to be bean-managed or container-managed with respect to ActivitySession management by setting its transaction type deployment attribute to be bean-managed or container-managed when deploying the enterprise bean. A bean that uses bean-managed transactions can use bean-managed ActivitySessions; a bean that uses container-managed transactions can use container-managed ActivitySessions.
- If you want a session bean or an enterprise application client to manage its own ActivitySessions, you must write the code that explicitly demarcates the boundaries of an ActivitySession, as described in Developing an enterprise bean or J2EE client to manage ActivitySessions.

The following high level tasks illustrate how to use an ActivitySession in an enterprise application:

- Develop an enterprise application that uses one or more enterprise beans that are persisted to non-transactional datastores. Use this task when you have an application that needs to coordinate multiple one-phase resource managers. For example, for two or more entity enterprise beans whose persistence is delegated to LocalTransaction resource adapters.

  In this scenario, the enterprise beans used by the application have an Activation policy of ActivitySession and a local transaction containment policy with a boundary of ActivitySession and resolution-control of ContainerAtBoundary. The synchronization of the EJB state data is synchronized, by the container, with the one-phase resource managers at ActivitySession completion and no application code is required to be aware of ActivitySession support.
- Develop an enterprise application in which an enterprise bean accesses a resource manager multiple times in different business methods. Use this task when you have an application that needs to extend a resource manager local transaction (RMLT) over several business methods of an enterprise bean instance.

  In this scenario, the enterprise beans used by the application have an Activation policy of ActivitySession and a local transaction containment policy with a boundary of ActivitySession and resolution-control of Application. The application logic starts and ends the RMLTs, for example using the javax.resource.cci.LocalTransaction interface offered by a LocalTransaction Connector, but is not constrained to start and commit the LocalTransaction in the same method.
- Develop an enterprise client application to use an ActivitySession to scope EJB activation and load-balancing. Use this task when you have an application client that needs to access an entity bean instance several times in the same client session, either without needing to run under a transaction context, or with the need to run under a number of distinct and serially-executed transactions.

  In this scenario, the enterprise beans used by the application client have an Activation policy of ActivitySession and a local transaction containment policy appropriate to the function of the enterprise bean. The enterprise client application can represent a period of user activity, for example a signon period, during which a number of interactions occur with one or more enterprise beans. If the enterprise client application begins an ActivitySession and invokes the enterprise beans within the scope of the UOW represented by the ActivitySession, then the enterprise bean instances are activated by the container on the ActivitySession boundary and remain in the active state until passivated by the container at the end of the ActivitySession. Workload affinity management based on the ActivitySession is a platform quality of service. Global transactions can begin and end within the ActivitySession, if they are wholly encapsulated by the ActivitySession and run serially. EJB instances activated at the ActivitySession boundary remain active across the serial global transactions.
- Develop a Web application client to participate in an ActivitySession context. A Web application that runs in the WebSphere Web container can participate in an ActivitySession context. Web applications can use the UserActivitySession interface to begin and end an ActivitySession context. Also, the

ActivitySession can be associated with an HttpSession, thereby extending access to the ActivitySession over multiple HTTP invocations and supporting EJB activation periods that can be determined by the lifecycle of the Web HTTP client.

The Web container manages ActivitySessions based on deployment descriptor attributes associated with the Web application module.

Example

For examples of using ActivitySessions in enterprise applications, see ActivitySessions samples

Developing an enterprise bean or enterprise application client to manage ActivitySessions

Use this task to write the code needed by a session EJB or enterprise application client to manage an ActivitySession, based on the example code extract provided.

About this task

In most situations, an enterprise bean can depend on the EJB container to manage ActivitySessions within the bean. In these situations, all you need to do is set the appropriate ActivitySession attributes in the EJB module deployment descriptor, as described in Configuring EJB module ActivitySession deployment attributes. Further, in general, it is practical to design your enterprise beans so that all ActivitySession management is handled at the enterprise bean level.

However, in some cases you may need to have a session bean or enterprise application client participate directly in ActivitySessions. You then need to write the code needed by the session bean or enterprise application client to manage its own ActivitySessions.

Note: Session beans that use BMT and have an **Activate at** setting of `Activity session` can manage ActivitySessions. Entity beans cannot manage ActivitySessions; the EJB container always manages ActivitySessions within entity beans.

When preparing to write code needed by a session bean or enterprise application client to manage ActivitySessions, consider the points described in ActivitySessions and transaction contexts.

To write the code needed by a session EJB or enterprise application client to manage an ActivitySession, complete the following steps based on the example code extract below:

1. Get an initial context for the ActivitySession.
2. Get an implementation of the UserActivitySession interface, by a JNDI lookup of the URL java:comp/websphere/UserActivitySession. The UserActivitySession interface is used to begin and end ActivitySessions and to query various attributes of the active ActivitySession associated with the thread.
3. Set the timeout, in seconds, after which any subsequently started ActivitySessions are automatically completed by the ActivitySession service. If the session bean or enterprise application client does not specifically set this value, the default timeout (300 seconds) is used.

   The default timeout can also be overridden for each application server, on the ***server*-> Activity Session Service** panel of the administrative console.
4. Start the ActivitySession, by calling the beginSession() method of the UserActivitySession.
5. Within the ActivitySession, call business methods to do the work needed. You can also call other methods of UserActivitySession to manage the ActivitySession; for example, to get the status of the ActivitySession or to checkpoint all the ActivitySession resources involved in the ActivitySession.
6. End the ActivitySession, by calling the endSession() method of the UserActivitySession.

Example

The following code extract provides a basic example of using the UserActivitySession interface:

```
// Get initial context
  InitialContext ic = new InitialContext();
// Lookup UserActivitySession
  UserActivitySession uas = (UserActivitySession)ic.lookup("java:comp/websphere/UserActivitySession");

// Set the ActivitySession timeout to 60 seconds
  uas.setSessionTimeout(60);
// Start a new ActivitySession context
  uas.beginSession();
// Do some work under this context
  MyBeanA beanA.doSomething();
  ...
  MyBeanB beanB.doSomethingElse();
// End the context
  uas.endSession(EndModeCheckpoint);
```

Setting EJB module ActivitySession deployment attributes

Use this task to set the ActivitySession deployment attributes for an enterprise bean to enable the bean to participate in an ActivitySession context and support ActivitySession-based operations.

Before you begin

This task description assumes that you have an Enterprise Archive (EAR) file, which contains an application enterprise bean that can be deployed in WebSphere Application Server. For more details about assembling applications, see assembling applications.

About this task

You configure the deployment attributes of an application by using an assembly tool. This topic describes the use of Rational Application Developer to configure the ActivitySession deployment attributes. These attributes are in addition to other deployment attributes, like Load at (which specifies when the bean loads its state from the database). For more detail about the fields in the assembly tool, and for associated task help, refer to the Rational Application Developer information.

To set the ActivitySession deployment attributes for an enterprise bean, complete the following steps:

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the application EAR file.

   Note: Ensure that you set the target server as WebSphere Application Server Version 7.0.

   For example, to change attributes of an existing application, use the Import wizard to import the EAR file into the assembly tool. To start the Import wizard:

   a. Click **File** → **Import** → **EAR file**.
   b. Click **Next**, then select the EAR file.
   c. In the Target server field, select `WebSphere Application Server v7.0`.
   d. Click **Finish**.
3. In the Project Explorer view of the Java EE perspective, right-click the EJB module for the enterprise bean instance, then click **Open With** → **Deployment Descriptor Editor**. A property dialog notebook for the enterprise bean instance is displayed in the property pane.
4. In the property pane, select the Beans tab.
5. Select the bean that you want to change.
6. In the WebSphere Extensions section, under **Bean Cache**, set the **Activate at** attribute to **ActivitySession**:

An enterprise bean with this activation policy is activated and passivated as follows:
- On an ActivitySession boundary, if an ActivitySession context is present on activation.
- On a transaction boundary, if a transaction context, but no ActivitySession context, is present on activation.
- Otherwise, on an invocation boundary.

7. In the Local Transactions group box, set the **Boundary** attribute to **ActivitySession**: When this setting is used, the local transaction must be resolved within the scope of any ActivitySession in which it was started or, if no ActivitySession context is present, within the same bean method in which it was started.
8. For entity beans, or session beans, set the ActivitySessions properties for each EJB method.
   a. In the property pane, select the ActivitySession tab.
   b. In the **Configure ActivitySession policies** field, click **Add** or **Edit** to set the **ActivitySession kind** attribute for methods of the enterprise bean. This specifies how the container must manage the ActivitySession boundaries when delegating a method invocation to an enterprise bean's business method:

      Never The container invokes bean methods without an ActivitySession context.
      - If the client invokes a bean method from within an ActivitySession context, the container throws an InvalidActivityException exception, which is a javax.rmi.RemoteException.
      - If the client invokes a bean method from outside an ActivitySession context, the container behaves in the same way as if the **Not Supported** value was set. The client must call the method without an ActivitySession context.

      Mandatory

      The container always invokes the bean method within the ActivitySession context associated with the client. If the client attempts to invoke the bean method without an ActivitySession context, the container throws an ActivityRequiredException exception to the client. The ActivitySession context is passed to any EJB object or resource accessed by an enterprise bean method.

      The ActivityRequiredException exception is javax.rmi.RemoteException.

      Requires new

      The container always invokes the bean method within a new ActivitySession context, regardless of whether the client invokes the method within or outside an ActivitySession context. The new ActivitySession context is passed to any enterprise bean objects or resources that are used by this bean method.

      Any received ActivitySession context is suspended for the duration of the method and resumed after the method ends. The container starts a new ActivitySession before method dispatch and completes it after the method ends.

      Required

      The container invokes the bean method within an ActivitySession context. If a client invokes a bean method from within an ActivitySession context, the container invokes the bean method within the client ActivitySession context. If a client invokes a bean method outside an ActivitySession context, the container creates a new ActivitySession context and invokes the bean method from within that context. The ActivitySession context is passed to any enterprise bean objects or resources that are used by this bean method.

      Not supported

      The container invokes bean methods without an ActivitySession context. If a client invokes a bean method from within an ActivitySession context, the container suspends the association between the ActivitySession and the current thread before invoking the method on the enterprise bean instance. The container then resumes the suspended association when the method invocation returns. The suspended ActivitySession context is not passed to any enterprise bean objects or resources that are used by this bean method.

      Supports

      If the client invokes the bean method within an ActivitySession, the container invokes the

bean method within an ActivitySession context. If the client invokes the bean method without a ActivitySession context, the container invokes the bean method without an ActivitySession context. The ActivitySession context is passed to any enterprise bean objects or resources that are used by this bean method.

c. Click **Next**.

d. Select the methods to which the ActivitySession kind policy is to be applied.

e. Click **Finish**.

How the container manages the ActivitySession boundaries when delegating a method invocation depends on both the **ActivitySession kind** set here, and the **Container transaction type**, as described in "Configuring transactional deployment attributes" on page 1482. For more detail about the relationship between these two properties, see "ActivitySession and transaction container policies in combination" on page 1501.

9. Save your changes to the deployment descriptor.
   a. Close the Deployment Descriptor Editor.
   b. When prompted, click **Yes** to save changes to the deployment descriptor.
10. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
11. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
12. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

    Note: Use **Run On Server** for unit testing only. The assembly tool controls the WebSphere Application Server installation and, when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console as described in Deploying and administering enterprise applications.

Setting Web module ActivitySession deployment attributes

Use this task to set the ActivitySession deployment attributes for a Web application to start UserActivitySessions and perform work scoped within ActivitySessions.

Before you begin

This task assumes that you have an Enterprise Archive (EAR) file that contains an application enterprise bean that can be deployed in WebSphere Application Server. For more details about assembling applications, see Assembling applications.

About this task

You can configure the deployment attributes of an application by using an assembly tool. This topic describes the use of Rational Application Developer to configure the deployment attributes.

To set the ActivitySession deployment attributes for a Web application, complete the following steps:

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.

2. Create or edit the Web module. For example, to change attributes of an existing module, click **File** → **Open**, then select the archive file for the module. For example, to change attributes of an existing module, use the Import wizard to import the EAR or WAR file into the assembly tool. To start the Import wizard:
   a. Click **File** → **Import**.
   b. Expand the Web folder, click **WAR file**, then click **Next**.
   c. Select the WAR file, then click **Finish**.
3. In the Project Explorer view of the Java EE perspective, right-click the component instance, right-click **Deployment Descriptor Editor**, then click **Open With** . A property dialog notebook for the Web module is displayed in the property pane.
4. In the property pane, select the Extended services tab.
5. Select the servlet that you want to change.
6. In the ActivitySession section, set the **ActivitySession control kind** attribute to Application, Container, or None.

   Application

   The Web application is responsible for starting and ending ActivitySessions, as follows:
   - If an HttpSession is active when an application begins an ActivitySession, the container associates the ActivitySession with the HttpSession.
   - If an ActivitySession is started in the absence of an HttpSession, the application must ensure it is completed before the dispatched method completes; otherwise, an exception results.
   - If an HttpSession is associated with a request dispatched to an application with this ActivitySession control value, and if that HttpSession has an ActivitySession associated with it, the container dispatches the request in the context of that ActivitySession. For example, the container resumes the ActivitySession context onto the thread before the dispatch.
   - A Web application can use both transactions and ActivitySessions. Any transactions started within the scope of an ActivitySession must be ended by the Web component that started them and within the same request dispatch.

   Container

   A servlet has no access to UserActivitySessions. Any HttpSession started by the servlet has an ActivitySession automatically associated with it by the container, and this ActivitySession is put onto the thread of execution. If such a servlet is dispatched by a request that has an HttpSession containing no ActivitySession, then the container starts an ActivitySession and associates it with the HttpSession and the thread.

   A Web application can use both transactions and ActivitySessions. Any transactions started within the scope of an ActivitySession must be ended by the Web component that started them and within the same request dispatch.

   None A servlet has no access to UserActivitySession. An HttpSession started by the servlet does not have an ActivitySession automatically associated with it by the container. If such a servlet is dispatched by a request that has an HttpSession containing an ActivitySession, then the container dispatches the request in the context of that ActivitySession. For example, the container resumes the ActivitySession context onto the thread before the dispatch.
7. To apply the changes and close the assembly tool, click **OK**. Otherwise, to apply the values but keep the property dialog open for additional edits, click **Apply**.
8. Save your changes to the deployment descriptor.
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to save changes to the deployment descriptor.
9. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
10. From the popup menu of the project, click **Deploy** to generate EJB deployment code.

11. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

    Note: Use **Run On Server** for unit testing only. The assembly tool controls the WebSphere Application Server installation and, when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the WAR file; for example, using the administrative console as described in Deploying and managing applications.

Application profiling

Task overview: Application profiling

You can use application profiling to configure multiple access intent policies on the same entity bean.

About this task

Application profiling reflects the fact that different units of work have different use patterns for enlisted entities and can require different kinds of support from the server runtime environment. For more information, see Application Profiling.

1. Assembling applications for application profiles. This topic describes how to configure tasks, create application profiles, and configure tasks on profiles.
2. Managing application profiles. This topic describes how to add and remove tasks from application profiles using the administrative console.
3. Using the TaskNameManager API. This topic describes how to programmatically set the current task name, but you should use this technique sparingly. Wherever possible, use the declarative method instead, which results in more portable function.

Application profiling

You can use application profiling to identify particular units of work to the product runtime environment. The run time can tailor its support to the exact requirements of that unit of work.

Application profiling requires accurate knowledge of an application's transactional configuration and the interaction of the application with its persistent state during the course of each transaction.

You can execute the analysis in either closed world or open world mode. A closed-world analysis assumes that all possible clients of the application are included in the analysis and that the resulting analysis is complete and correct. The results of a closed-world analysis report the set of all transactions that can be invoked by a web, JMS, or application client. The results exclude many potential transactions that never execute at run time.

An open-world analysis assumes that not all clients are available for analysis or that the analysis cannot return complete or accurate results. An open-world analysis returns the complete set of possible transactions.

The results of an analysis persist as an application profiling configuration. The assembly tool establishes container managed tasks for servlets, JavaServer Pages (JSP) files, application clients, and Message Driven Beans (MDBs). Application profiles for the tasks are constructed with the appropriate access intent for the entities enlisted in the transaction represented by the task. However, in practice, there are many

situations where the tool returns at best incomplete results. Not all applications are amenable to static analysis. Some factory and command patterns make it impossible to determine the call graphs. The tool does not support the analysis of *ActivitySessions*.

You should examine the results of the analysis very carefully. In many cases you must manually modify them to meet the requirements of the application. However, the tool can be an effective starting place for most applications and may offer a complete and quick configuration of application profiles for some applications.

Access intent is the only runtime component that makes use of the application profiling functionality. For example, you can configure one transaction to load an entity bean with strong update locks and configure another transaction to load the same entity bean without locks.

Application profiling introduces two new concepts in order to achieve this function: *tasks* and *profiles*.

Tasks A task is a configurable name for a unit of work. *Unit of work* in this case means either a transaction or an ActivitySession. The task name is typically assigned declaratively on a J2EE component that can initiate a unit of work. Most commonly, the task is configured on a method of an Enterprise JavaBeans file that is declared either for container-managed transactions or bean-managed transactions. Any unit of work that begins in the scope of a configured task is associated with that task name. A unit of work can only be named when it is initiated, and the name cannot change for the lifetime of that unit of work. A unit of work ignores any subsequent task name configurations at any point after it has begun. The task is used for the duration of its unit of work to identify configured policies specific to that unit of work.

Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

For a Version 6.x client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to *true* in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the launchClient command.

Profiles

A profile is simply a mapping of a task to a set of access intent policies that are configured on entity beans. When an invocation on a bean (whether by a finder method, a CMR getter, or a dynamic query) requires data to be retrieved from the back end system, the current task associated with the request is used to determine the exact requirement of the transaction. The same bean loads and behaves differently in the context of the task-to-profile mapping. Each profile provides the developer an opportunity to reconfigure the application's access intent. If a request is operating in the absence of a task, the runtime environment uses either a method-level access intent (if any) or a bean-level default access intent.

Note: The application profile configuration is application scope configuration data. If any Enterprise JavaBean (EJB) module contains an application profile configuration, all other EJB modules are implicitly regulated by the Application Profiling service even if they do not contain application profile configuration data.

For example, an application has two EJB modules: EJBModule1 and EJBModule2.

The EJBModule1 has an application profile named AppProfile1. This AppProfile1 is registered by a task named task1. This task1 becomes a *known-to-application task* and is honored when associated with a unit of work within this application. With the presence of

any known-to-application task, method level access intent configurations are ignored and only bean level access intent configurations are applied.

The EJBModule2 contains no application profile configuration data. All entity beans are **not** configured with bean level access intent explicitly, but some methods have method level access intent configurations. If an entity bean in the EJBModule2 is loaded in a unit of work that is associated with task1, the bean-level access intent configuration is applied and method level access intent configuration is ignored. Because the bean level access intent is not set explicitly, the default bean level access intent, which is wsPessimisticUpdate-WeakestLockAtLoad, is applied.

Tasks and units of work considerations:

The application profiling function works under the unit of work (UOW) concept. UOW in this case means either a transaction or an ActivitySession.

The task name on a method is used only when a UOW is begun, because of that method being invoked. This gives it a more predictable data access pattern based on the active unit of work. To be more specific, this approach ensures that a bean type with only one configured access intent is loaded within a UOW, because a bean is configured with only one access intent within an application profile. This configured access intent for a bean type is determined at assembly time and is enforced by the Application Profile service.

A task name is always associated with a unit of work, and that task name does not change for the duration of that UOW. When a UOW associated with a method is begun because of that method being invoked, if a task name is associated with the method then that task name is used to name the UOW. A task assigned to a unit of work is considered a named UOW.

If a task name is not associated with the method that began the UOW, then a default access intent is used and the UOW is unnamed. A unit of work can only be named when the UOW is begun and that task name remains for the life of the UOW. Furthermore, the task assigned to a UOW can never be changed for the life of that UOW. Any task names associated with a method are ignored if that method does not begin a UOW (either container managed or component managed).

It is not possible to change the task name assigned to a unit of work. However, it is possible that in a call sequence consisting of many different application calls a different task name might need to be used for different calls. In this case it is important for the deployer to begin a new UOW and associate with the UOW the necessary task name. For example, assume you have the following beans: sb1 is a session bean, eb2 and eb3 are container managed persistence (CMP) entity beans. When sb1 is called, a transaction is begun and task 't1' is associated with it. Further assume that sb1 then calls eb2 and eb3. If neither eb2 or eb3 create a unit of work, then these beans execute within the UOW context from sb1 and as such its task name (t1). If eb2 or eb3 need to execute within a task name other than t1, then these beans must define a unit of work and associate with it the appropriate task name.

Note that if an application deployer does not specifically configure a transaction on a method, WebSphere Application Server creates a global transaction by default. This is important because if a task is defined on a method, but a UOW is not specifically configured on that method, the EJB container automatically creates a global transaction on behalf of that method. As such, this task name is associated with the UOW and any application profiles mapped to this task are used.

Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

Application profiles:

An application profile is the set of access intent policies that should be selectively applied for a particular unit of work (a transaction or *ActivitySession*).

Application profiling enables applications to run under different sets of policies depending on the active task under which the application is operating.

The active task depends upon the current unit of work mechanism. If the current unit of work is a global transaction, then the task is the name associated with that transaction. If the global transaction was not named when it was initiated, then there is no active task anywhere in the scope of that transaction.

If the current unit of work is a local transaction associated with an ActivitySession, then the task is the name associated with that ActivitySession. If the ActivitySession was not named when it was initiated, then there is no active task for any local transaction bound to that ActivitySession. If the current unit of work is a local transaction that is not associated with an ActivitySession, then the task is the name associated with that local transaction. If the local transaction was not associated with a task when the local transaction was initiated, then there is no active task for the duration of that local transaction. In other words, the active task is the task associated with the unit of work on the thread that is coordinating database resources. If the controlling unit of work was not associated with a task when that unit of work was initiated, then there is no active task in the scope of that unit of work.

Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

For a Version 6.x client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to *true* in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the launchClient command.

Consider an application that centralizes the student records for a school district. These records are frequently accessed by the school district's central office in order to generate reports. The report generation process would be optimized if it held no locks with the back end system, and if the records could be read into memory with as few back end operations as possible. Occasionally, however, the records are updated by the students' instructors. Without the ability to distinguish between transactions, the developer is forced to assume a worst-case scenario and, wishing to use pessimistic concurrency, lock the records for all transactions.

Using the application profiling service, the developer can configure in as many ways as necessary the access intent under which the students' records are loaded. Under one profile, the records can be configured with an exclusive pessimistic update intent, not only locking-out competing transactions but ensuring that the student is not removed from the system before the transaction completes. Under another profile, the records can be configured with an optimistic intent as part of an object graph that is read from the back end system in a single database operation. The task represented by the pessimistic profile receives the strong-locking semantics required for certain transactions, while the task represented by the optimistic profile receives the performance benefits appropriate for other transactions.

Application profiling performance considerations:

Application profiling enables assembly configuration techniques that improve your application run time, performance and scalability. You can configure tasks that identify incoming requests, identify access intents determining concurrency and other data access characteristics, and profiles that map the tasks to the access intents.

The capability to configure the application server can improve performance, efficiency and scalability, while reducing development and maintenance costs. The application profiling service has no tuning parameters, other than a checkbox for disabling the service if the service is not necessary. However, the overhead for the application profile service is small and should not be disabled, or unpredictable results can occur.

Access intents enable you to specify data access characteristics. The WebSphere runtime environment uses these hints to optimize the access to the data, by setting the appropriate isolation level and concurrency. Various access intent hints can be grouped together in an access intent policy.

In the product, it is recommended that you configure bean level access intent for loading a given bean. Application profiling enables you to configure multiple access intent policies on the entity bean, if desired. Some callers can load a bean with the intent to read data, while others can load the bean for update. The capability to configure the application server can improve performance, efficiency, and scalability, while reducing development and maintenance costs.

Access intents enable the EJB container to be configured providing optimal performance based on the specific type of enterprise bean used. Various access intent hints can be specified declaratively at deployment time to indicate to WebSphere resources, such as the container and persistence manager, to provide the appropriate access intent services for every EJB request.

The application profiling service improves overall entity bean performance and throughput by fine tuning the run time behavior. The application profiling service enables EJB optimizations to be customized for multiple user access patterns without resorting to ″worst case″ choices, such as pessimistic update on a bean accessed with the findByPrimaryKey method, regardless of whether the client needs it for read or for an update.

Application profiling provides the capability to define the following hierarchy: **Container-Managed Tasks** > **Application Profiles** > **Access Intent Policies** > **Access Intent Overrides**. Container-managed tasks identify units of work (UOW) and are associated with a method or a set of methods. When a method associated with the task is invoked, the task name is propagated with the request. For example, a UOW refers to a unique path within the application that can correspond to a transaction or ActivitySession. The name of the task is assigned declaratively to a Java EE client or servlet, or to the method of an enterprise bean. The task name identifies the starting point of a call graph or subgraph; the task name flows from the starting point of the graph downstream on all subsequent IIOP requests, identifying each subsequent invocation along the graph as belonging to the configured task. As a best practice, wherever a UOW starts, for example, a transaction or an ActivitySession, assign a task to that starting point.

The application profile service associates the propagated tasks with access intent policies. When a bean is loaded and data is retrieved, the characteristics used for the retrieval of the data are dictated by the application profile. The application profile configures the access intent policy and the overrides that should be used to access data for a specific task.

Access intent policies determine how beans are loaded for specific tasks and how data is accessed during the transaction. The access intent policy is a named group of access intent hints. The hints can be used, depending on the characteristics of the database and resource manager. Various access intent hints applied to the data access operation govern data integrity. The general rule is, the more data integrity, the more overhead. More overhead causes lower throughput and the opportunity for simultaneous data access from multiple clients.

If specified, access intent overrides provide further configuration for the access intent policy.

Best practices

Application profiling is effective in a variety of different scenarios including:

- **The same bean is loaded with different data access patterns**

  The same bean or set of beans can be reused across applications, but each of those applications has differing requirements for the bean or for beans within the invocation graph. One application can require that beans be loaded for update, while another application requires beans be loaded for read only. Application profiling enables deploy time configuration for beans to distinguish between EJB loading requirements.
- **Different clients have different data access requirements**

  The same bean or set of beans can be used for different types of client requests. When those clients have different requirements for the bean, or for beans within the invocation graph, application profiling can be used to tailor the bean loading characteristics to the requirements of the client. One client can require beans be loaded for update, while another client requires beans be loaded for read only. Application profiling enables deploy time configuration for beans to distinguish between EJB loading requirements.

Monitoring tools

You can use the Tivoli Performance Viewer, database and logs as monitoring tools.

You can use the Tivoli Performance Viewer to monitor various metrics associated with beans in an application profiling configuration. The following sections describe at a high level the Tivoli Performance Viewer metrics that reflect changes when access intents and application profiling are used:

- **Collection scope**

  The enterprise beans group contains EJB life cycle information, either a cumulative value for a group of beans, or for specific beans. You can monitor this information to determine the difference between using the ActivitySession scope versus the transaction scope. For the transaction scope, depending on how the container transactions are defined, activates and passivates can be associated with method invocations. The application could use the ActivitySession scope to reduce the frequency of activates and passivates. For more information, see ″Using the ActivitySession service.″
- **Collection increment**

  The enterprise beans group contains EJB life cycle information, either a cumulative value for a group of beans, or for specific beans. You can monitor *Num Activates* to watch the number of enterprise beans activated for a particular findByPrimaryKey operation. For example, if the collection increment is set to 10, rather than the default 25, the *Num Activates* value shows 25 for the initial findByPrimaryKey, before any result set iterator runs. If the number of activates rarely exceeds the collection increment, consider reducing the collection increment setting.
- **Resource manager prefetch increment**

  The resource manager prefetch increment is a hint acted upon by the database engine to depend upon the database. The Tivoli Performance Viewer does not have a metric available to show the effect of the resource manager prefetch increment setting.
- **Read ahead hint**

  The enterprise beans group contains EJB life cycle information, either a cumulative value for a group of beans, or for specific beans. You can monitor *Num Activates* to watch the number of enterprise beans activated for a particular request. If a read ahead association is not in use, the *Num Activates* value shows a lower initial number. If a read ahead association is in use, the *Num Activates* value represents the number of activates for the entire call graph.

Database tools are helpful in monitoring the different bean loading characteristics that introduce contention and concurrency issues. These issues can be solved by application profiling, or can be made worse by the misapplication of access intent policies.

Database tools are useful for monitoring locking and contention characteristics, such as locks, deadlocks and connections open. For example, for locks the DB2 Snapshot Monitor can show statistics for lock waits, lock time-outs and lock escalations. If excessive lock waits and time-outs are occurring, application profiling can define specific client tasks that require a more string level of locking, and other client tasks that do not require locking. Or, a different access intent policy with less restrictive locking could be applied. After applying this configuration change, the snapshot monitor shows less locking behavior. Refer to information about the database you are using on how to monitor for locking and contention.

The **application server logs** can be monitored for information about rollbacks, deadlocks, and other data access or transaction characteristics that can degrade performance or cause the application to fail.

Application profiling tasks

Tasks are named units of work. They are the mechanism by which the runtime environment determines which access intent policies to apply when an entity bean's data is loaded from the back end system.

Application profiles enable developers to configure an entity bean with multiple access intent policies; if there are *n* instances of profiles in a given application, each bean can be configured with as many as *n* access intent policies.

A task is associated with a transaction or an ActivitySession at the initiation of the unit of work. The task, which cannot change for the lifetime of the unit of work, is always available anywhere within the scope of that unit of work to apply the access intent policy configured for that particular unit of work.

If an enterprise application is configured to use application profiling in any part of the application, then application profiling is active and method-level access intent configurations are ignored when units of works are associated with known-to-application tasks.

If an entity bean is loaded in a unit of work that is not associated with a task, or is associated with a task that is unassociated with an application profile, the default bean-level access intent or the method-level access intent configuration is applied. If a unit of work is associated with a task that is configured with an application profile, the bean-level access intent configuration within the appropriate application profile is applied.

Note: The application profile configuration is application scope configuration data. If any enterprise Javabean (EJB) module contains an application profile configuration, all other EJB modules are implicitly regulated by the Application Profiling service even if they do not contain application profile configuration data.

For example, an application has two EJB modules: EJBModule1 and EJBModule2.

The EJBModule1 has an application profile named AppProfile1. This AppProfile1 is registered by a task named task1. This task1 becomes a *known-to-application task* and is honored when associated with a unit of work within this application. With the presence of any known-to-application task, method level access intent configurations are ignored and only bean level access intent configurations are applied.

The EJBModule2 contains no application profile configuration data. All entity beans are **not** configured with bean level access intent explicitly, but some methods have method level access intent configurations. If an entity bean in the EJBModule2 is loaded in a unit of work that is associated with task1, the bean-level access intent configuration is applied and method level access intent configuration is ignored. Because the bean level access intent is not set explicitly, the default bean level access intent, which is wsPessimisticUpdate-WeakestLockAtLoad, is applied.

The active task depends upon the current unit of work mechanism. If the current unit of work is a global transaction, then the task is the name associated with that transaction. If the global transaction was not named when it was initiated, then there is no active task anywhere in the scope of that transaction.

If the current unit of work is a local transaction associated with an ActivitySession, then the task is the name associated with that ActivitySession. If the ActivitySession was not named when it was initiated, then there is no active task for any local transaction bound to that ActivitySession. If the current unit of work is a local transaction that is not associated with an ActivitySession, then the task is the name associated with that local transaction. If the local transaction was not associated with a task when the local transaction was initiated, then there is no active task for the duration of that local transaction. In other words, the active task is the task associated with the unit of work on the thread that is coordinating database resources. If the controlling unit of work was not associated with a task when that unit of work was initiated, then there is no active task in the scope of that unit of work.

For example, consider a school district application that calls through a session bean in order to interact with student records. One method on the session bean allows administrators to modify the students' records; another method supports student requests to view their own records. Without application profiling, the two tasks would operate anonymously and the runtime environment would be unable to distinguish work operating on behalf of one task or the other. To optimize the application, a developer can configure one of the methods on the session bean with the task ″updateRecords″ and the other method on the session bean with the task ″readRecords″. When registered with an application profile that has the student bean configured with the appropriate locking access intent, the ″updateRecords″ task is assured that it is not unnecessarily blocking transactions that need to only read the records. For more information about the relationships between tasks and units of work, see “Tasks and units of work considerations” on page 1518.

Tasks can be configured to be managed by the container or to be programmatically established by the application. Container managed tasks can be configured on servlets, JavaServer Pages (JSP) files, application clients, and the methods of Enterprise JavaBeans (EJB). Configured container-managed tasks are only associated with units of work that the container initiates after the task name is set. Application managed tasks can be configured on all J2EE components. In the case of enterprise beans they must be bean managed transactions.″

Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

For a Version 6.x client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to *true* in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the launchClient command.

Application profiling interoperability

Using application profiling with 5.x compatibility mode or in a clustered environment with mixed product versions and mixed platforms can affect its behavior in different ways.

The effect of 5.x Compatibility Mode

Application profiling supports *forward* compatibility. Application profiles created in previous versions of WebSphere Application Server (Enterprise Edition 5.0 or WebSphere Business Integration Server Foundation 5.1) can only run in WebSphere Application Server Version 6 or later if the Application Profiling 5.x Compatibility Mode attribute is turned on. If the 5.x Compatibility Mode attribute is off, Version 5 application profiles might display unexpected behavior.

Similarly, application profiles that you create using the latest version of WebSphere Application Server are not compatible with Version 5 or earlier versions. Even applications configured with application profiles run on Version 6.x servers with the Application Profiling 5.x Compatibility Mode attribute turned on cannot interact with applications configured with profiles run on Version 5 servers.

Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

For a Version 6.x client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to **true** in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the *launchClient* command.

Considerations for a clustered environment

In a clustered environment with mixed product versions and mixed platforms, applications configured with application profiles might exhibit unexpected behavior because previous versions of server members cannot support the application profiling of Version 6.x.

If a clustered environment contains both Version 5.x and 6.x server members, and if any applications are configured with application profiles, the Application Profiling 5.x Compatibility Mode attribute must be turned on in Version 6 server members. Still, this cluster can only support Version 5 application profiling behavior. To support applications configured with Version 6 application profiles in a cluster environment, all server members in the cluster must be at the Version 6.x level.

WebSphere Application Server Enterprise Edition Version 5.0.2

If you use WebSphere Application Server Enterprise Edition 5.0.2, you must apply WebSphere Application Server Version 5 service pack 7 or later service pack to enable Application Profiling interoperability.

Assembling applications for application profiling

To enable application profiling, you must configure tasks, create an application profile, and declaratively configure a unit of work on necessary methods.

Before you begin

Application profiling enables multiple access intent policies to be configured on the same entity bean, each specified for a particular unit of work. You can use the one of the default policies or create your own. To create your own access intent policy, see the topic, Creating a custom access intent policy, in the assembly tool information center.

1. Configure tasks. Declaratively configure tasks as described in the following topics that are located in the assembly tool information center:
   - Configuring container-managed tasks for Enterprise Java Beans.
   - Configuring container-managed tasks for web components.
   - Configuring container-managed tasks for application clients.

   On rare occasions, you might find it necessary to configure tasks *programmatically*. Application profiling supports this requirement with a simple interface that enables a task name to be set before a unit of work is programmatically initiated. Setting a task name and then initiating a transaction or ActivitySession causes the task to be associated with the new unit of work. This interface cannot be used within Enterprise JavaBeans that are configured for container-managed transactions or container-managed ActivitySessions because units of work can only be associated with a task at the exact time that the unit of work is initiated. The call to set the task name must therefore be invoked before the unit of work is begun. Units of work cannot be named after they are begun. See Using the TaskNameManager interface.

   Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units

of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

For a Version 6.0 client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to *true* in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the launchClient command.

2. Create an application profile. See the assembly tool information center to complete this task.
3. Declaratively configure a unit of work on necessary methods. In step one of this article, you defined a task on a method. The task defined on a method only becomes active when a unit of work is begun on that method's behalf. The method must begin a new unit of work for the configured task to be applied. If the method runs under an imported unit of work, then the configured task on the method is ignored and the task (if any) associated with the imported unit of work is used. If the container begins a new unit of work when the method executes, then it is associated with the configured task name. Therefore, the last step in assembling applications for application profiling is to define a unit of work on any method that has a task name (and eventually an Application Profile) associated with it. A unit of work can either be a transaction or an ActivitySession. "Defining container transactions for EJB modules" on page 209 describes how to configure a transaction on an EJB module. "Configuring transactional deployment attributes" on page 1482 describes how to define other transaction attributes. "Using the ActivitySession service" on page 1495 describes how to use and create an ActivitySession unit of work. For more information about the relationships between tasks and units of work, see "Tasks and units of work considerations" on page 1518.

What to do next

To complete the following tasks using assembly tools review the assembly tool documentation at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html:

- Automatic configuration of application profiling

  The assembly tool includes a static analysis engine that can assist you in configuring application profiling. The tool examines the compiled classes and the deployment descriptor of a Java EE application to determine the entry point of transactions, calculate the set of entities enlisted in each transaction, and determine whether the entities are read or updated during the course of each identified transaction.
- Automatically configure application profiles and tasks.

  Automatically configure application profiling for an application through static analysis.
- Apply profile-scoped access intent policies to entity beans.

  Configure entities with access intent for an application profile.
- Create a custom access intent policy.

  Define a custom access intent policy, which can be configured for Enterprise JavaBeans (EJB) 2.x and 3.0 entity beans.
- Create an application profile.

  An application profile contains a set of access intent policies applied to an application's entity beans. The access intent policies are only applied for requests that are associated with tasks configured on the application profile.
- Configure container-managed tasks for application clients.

  For application clients that programmatically begin either a transaction or ActivitySession only, you must configure an application client's container-managed task to associate requests from the client with an application profile.
- Configure container-managed tasks for Web components.

For Web components that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession only, you can configure Web components application-managed tasks to associate requests from a servlet or JavaServer Pages (JSP) file with application profiles.

- Configure container-managed tasks for Enterprise JavaBeans.

  For methods that cause a new transaction or ActivitySession to be started either by the container or programmatically by the EJB developer, you can configure an enterprise bean's container-managed tasks to associate requests from the bean with application profiles.
- Configure container-managed tasks for application clients.

  For application clients that programmatically begin either a transaction or ActivitySession only, you must configure an application client's container-managed task to associate requests from the client with an application profile.
- Configure application-managed tasks for Web components.

  For Web components that programmatically begin either a transaction or ActivitySession only, you can configure a Web component's container-managed task to associate requests from a servlet or JSP file with an application profile.
- Configure application-managed tasks for Enterprise JavaBeans.

  For Enterprise JavaBeans that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession only, you can configure EJB application-managed tasks to associate requests from the bean with application profiles.

Asynchronous beans

Using asynchronous beans

The asynchronous beans feature adds a new set of APIs that enable Java 2 Platform Enterprise Edition J2EE applications to run asynchronously inside an Integration Server.

About this task

This topic provides a brief overview of the tasks involved in using asynchronous beans. For a more detailed description of the asynchronous beans model, review the conceptual topic Asynchronous beans. For detailed information on the programming model for supported asynchronous beans interfaces, see the topic Work managers.

1. Configure work managers.
2. Configure timer managers.
3. Assemble applications that use asynchronous beans work managers.
4. Develop work objects to run code in parallel.
5. Develop event listeners.
6. Develop asynchronous scopes.

Asynchronous beans

An asynchronous bean is a Java object or enterprise bean that can run asynchronously by a Java Platform, Enterprise Edition (Java EE) application, using the Java EE context of the asynchronous bean creator.

Asynchronous beans can improve performance by enabling a Java EE program to decompose operations into parallel tasks. Asynchronous beans support the construction of stateful, active Java EE applications. These applications address a segment of the application space that Java EE has not previously addressed (that is, advanced applications that require application threading, active agents within a server application, or distributed monitoring capabilities).

Asynchronous beans can run using the Java EE security context of the creator Java EE component. These beans also can run with copies of other Java EE contexts, such as:

- Internationalization context
- Application profiles, which are not supported for Java EE 1.4 applications and deprecated for Java EE 1.3 applications
- Work areas

Asynchronous bean interfaces

Four types of asynchronous beans exist:

Work object

There are two work interfaces that essentially accomplish the same goal. The legacy Asynchronous Beans work interface is com.ibm.websphere.asynchbeans.Work, and the CommonJ work interface is commonj.work.Work. A work object runs parallel to its caller using the work manager `startWork` or `schedule` method (`startWork` for legacy Asynchronous Beans and `schedule` for CommonJ). Applications implement work objects to run code blocks asynchronously. For more information on the Work interface, see the generated API documentation.

Timer listener

This interface is an object that implements the commonj\timers\TimerListener interface. Timer listeners are called when a high-speed transient timer expires. For more information on the TimerListener interface, see the generated API documentation.

Alarm listener

An alarm listener is an object that implements the com.ibm.websphere.asynchbeans.AlarmListener interface. Alarm listeners are called when a high-speed transient alarm expires. For more information on the AlarmListener interface, see the generated API documentation.

Event listener

An event listener can implement any interface. An event listener is a lightweight, asynchronous notification mechanism for asynchronous events within a single Java virtual machine (JVM). An event listener typically enables Java EE components within a single application to notify each other about various asynchronous events.

Supporting interfaces

Work manager

Work managers are thread pools that administrators create for Java EE applications. The administrator specifies the properties of the thread pool and a policy that determines which Java EE contexts the asynchronous bean inherits.

CommonJ Work manager

The CommonJ work manager is similar to the work manager. The difference between the two is that the CommonJ work manager contains a subset of the asynchronous beans work manager methods. Although CommonJ work manager functions in a Java EE 1.4 environment, each JNDI lookup of a work manager does not return a new instance of the WorkManager. All the JNDI lookup of work managers within a scope have the same instance.

Timer manager

Timer managers implement the commonj.timers.TimerManager interface, which enables Java EE applications, including servlets, EJB applications, and JCA Resource Adapters, to schedule future timer notifications and receive timer notifications. The timer manager for Application Servers specification provides an application-server supported alternative to using the J2SE `java.util.Timer` class, which is inappropriate for managed environments.

Event source

An event source implements the com.ibm.websphere.asynchbeans.EventSource interface. An event source is a system-provided object that supports a generic, type-safe asynchronous notification server within a single JVM. The event source enables event listener objects, which implement any interface to be registered. For more information on the EventSource interface, see the generated API documentation.

Event source events

Every event source can generate its own events, such as listener count changed. An application

can register an event listener object that implements the class com.ibm.websphere.asynchbeans.EventSourceEvents. This action enables the application to catch events such as listeners being added or removed, or a listener throwing an unexpected exception. For more information on the EventSourceEvents class, see the generated API documentation.

Additional interfaces, including alarms and subsystem monitors, are introduced in the topic Developing Asynchronous scopes, which discusses some of the advanced applications of asynchronous beans.

Transactions

Every asynchronous bean method is called using its own transaction, much like container-managed transactions in typical enterprise beans. It is very similar to the situation when an Enterprise Java Beans (EJB) method is called with TX_NOT_SUPPORTED. The runtime starts a local transaction before invoking the method. The asynchronous bean method is free to start its own global transaction if this transaction is possible for the calling Java EE component. For example, if an enterprise bean creates the component, the method that creates the asynchronous bean must be TX_BEAN_MANAGED.

When you call an entity bean from within an asynchronous bean, for example, you must have a global transactional context available on the current thread. Because asynchronous bean objects start local transactional contexts, you can encapsulate all entity bean logic in a session bean that has a method marked as TX_REQUIRES or equivalent. This process establishes a global transactional context from which you can access one or more entity bean methods.

If the asynchronous bean method throws an exception, any local transactions are rolled back. If the method returns normally, any incomplete local transactions are completed according to the unresolved action policy configured for the bean. EJB methods can configure this policy using their deployment descriptor. If the asynchronous bean method starts its own global transaction and does not commit this global transaction, the transaction is rolled back when the method returns.

Access to Java EE component metadata

If an asynchronous bean is a Java EE component, such as a session bean, its own metadata is active when a method is called. If an asynchronous bean is a simple Java object, the Java EE component metadata of the creating component is available to the bean. Like its creator, the asynchronous bean can look up the java:comp namespace. This look up enables the bean to access connection factories and enterprise beans, just as it would if it were any other Java EE component. The environment properties of the creating component also are available to the asynchronous bean.

The `java:comp` namespace is identical to the one available for the creating component; the same restrictions apply. For example, if the enterprise bean or servlet has an EJB reference of `java:comp/env/ejb/MyEJB`, this EJB reference is available to the asynchronous bean. In addition, all of the connection factories use the same resource-sharing scope as the creating component.

Connection management

An asynchronous bean method can use the connections that its creating Java EE component obtained using java:comp resource references. (For more information on resource references, see References). However, the bean method must access those connections using a get, use or close pattern. There is no connection caching between method calls on an asynchronous bean. The connection factories or datasources can be cached, but the connections must be retrieved on every method call, used, and then closed. While the asynchronous bean method can look up connection factories using a global Java Naming and Directory Interface (JNDI) name, this is not recommended for the following reasons:

- The JNDI name is hard coded in the application (for example, as a property or string literal).
- The connection factories are not shared because there is no way to specify a sharing scope.

For code examples that demonstrate both the correct and the incorrect ways to access connections from asynchronous bean methods, see the topic Example: Asynchronous bean connection management.

Deferred start of Asynchronous Beans

Asynchronous beans support deferred start by allowing serialization of Java EE service context information. The `WorkWithExecutionContext createWorkWithExecutionContext(Work r)` method on the WorkManager interface will create a snapshot of the Java EE service contexts enabled on the WorkManager. The resulting `WorkWithExecutionContext` object can then be serialized and stored in a database or file. This is useful when it is necessary to store Java EE service contexts such as the current security identity or Locale and later inflate them and run some work within this context. The `WorkWithExecutionContext` object can run using the startWork() and doWork() methods on the WorkManager interface.

All `WorkWithExecutionContext` objects must be deserialized by the same application that serialized it. All EJBs and classes must be present in order for Java to successfully inflate the objects contained within.

Deferred start and security

The asynchronous beans security service context might require Common Secure Interoperability Version 2 (CSIv2) identity assertion to be enabled. Identity assertion is required when a `WorkWithExecutionContext` object is deserialized and run to Java Authentication and Authorization Service (JAAS) subject identity credential assignment. Review the following topics to better understand if you need to enable identity assertion, when using a `WorkWithExecutionContext` object:

- Configuring Common Secure Interoperability Version 2 and Security Authentication Service authentication protocol
- Identity Assertion

There are also issues with interoperating with `WorkWithExecutionContext` objects from different versions of the product. See Interoperating with asynchronous beans .

JPA-related limitations

Use of asynchronous beans within a JPA extended persistence context is not supported.

A JPA extended persistence context is inconsistent with the scheduling and multi-threading capabilities of asynchronous beans and will not be accessible from an asynchronous bean thread.

Likewise, an asynchronous bean should not be created such that it takes a javax.persistence.EntityManager (or subclass) as a parameter since EntityManager instances are not intended to be thread safe.

Work managers:

A work manager is a thread pool created for Java Platform, Enterprise Edition (Java EE) applications that use asynchronous beans.

Using the administrative console, an administrator can configure any number of work managers. The administrator specifies the properties of the work manager, including the Java EE context inheritance policy for any asynchronous beans that use the work manager. The administrator binds each work manager to a unique place in Java Naming and Directory Interface (JNDI). You can use work manager objects in any one of the following interfaces:

- Asynchronous beans
- CommonJ work manager (For details, see the CommonJ work manager section in this article.)

The selected type of interface is resolved during the JNDI lookup time. The interface type is the value that you specify in the ResourceRef, rather than the interface type specified in the configuration object. For example, you can have one ResourceRef for each interface per configuration object, and each ResourceRef lookup returns that appropriate type of instance.

The work managers provide a programming model for the Java EE 1.4 applications. For more information, see the Programming model section in this article.

When writing a Web or Enterprise JavaBeans (EJB) component that uses asynchronous beans, the developer should include a resource reference in each component that needs access to a work manager. For more information on resource references, see the article References. The component looks up a work manager using a logical name in the component, java:comp namespace, just as it looks up a data source, enterprise bean or connection factory.

The deployer binds physical work managers to logical work managers when the application is deployed.

For example, if a developer needs three thread pools to partition work between bronze, silver, and gold levels, the developer writes the component to pick a logical pool based on an attribute in the client application profile. The deployer has the flexibility to decide how to map this request for three thread pools. The deployer might decide to use a single thread pool on a small machine. In this case, the deployer binds all three resource references to the same work manager instance (that is, the same JNDI name). A larger machine might support three thread pools, so the deployer binds each resource reference to a different work manager. Work managers can be shared between multiple Java EE applications installed on the same server.

An application developer can use as many logical work managers as necessary. The deployer chooses whether to map one physical work manager or several to the logical work manager defined in the application.

All Java EE components that need to share asynchronous scope objects must use the same work manager. These scope objects have an affinity with a single work manager. An application that uses asynchronous scopes should verify that all of the components using scope objects use the same work manager.

When multiple work managers are defined, the underlying thread pools are created in a Java virtual machine (JVM) only if an application within that JVM looks up the work manager. For example, there might be ten thread pools (work managers) defined, but none are actually created until an application looks these pools up.

Note: Asynchronous beans do not support submitting work to remote JVMs.

CommonJ Work Manager

The CommonJ work manager is similar to the work manager. The difference between the two is that the CommonJ work manager contains a subset of the asynchronous beans work manager methods. Although CommonJ work manager functions in a Java EE 1.4 environment, the interface does not return a new instance for each JNDI naming lookup, since this specification is not included in the Java EE specification.

Remote start of work. The CommonJ Work specification optional feature for work running remotely is not supported. Even if a unit of work implements the `java.io.Serializable` interface, the unit of work does not run remotely.

How to look up a work manager

An application can look up a work manager as follows. Here, the component contains a resource reference named `wm/myWorkManager`, which was bound to a physical work manager when the component was deployed:

```
InitialContext ic = new InitialContext();
WorkManager wm = (WorkManager)ic.lookup("java:comp/env/wm/myWorkManager");
```

Inheritance Java EE contexts

Asynchronous beans can inherit the following Java EE contexts.

Internationalization context
When this option is selected and the internationalization service is enabled, and the internationalization context that exists on the scheduling thread is available on the target thread.

Work area
When this option is selected, the work area context for every work area partition that exists on the scheduling thread is available on the target thread.

Application profile (deprecated)
Application profile context is not supported and not available for Java EE 1.4 applications. For Java EE 1.3 applications, when this option is selected, the application profile service is enabled, and the application profile service property, **5.x compatibility mode**, is selected. The application profile task that is associated with the scheduling thread is available on the target thread for Java EE 1.3 applications. For Java EE 1.4 applications, the application profile task is a property of its associated unit of work, rather than a thread. This option has no effect on the behavior of the task in Java EE 1.4 applications. The scheduled work that runs in a Java EE 1.4 application does not receive the application profiling task of the scheduling thread.

Security
The asynchronous bean can be run as anonymous or as the client authenticated on the thread that created it. This behavior is useful because the asynchronous bean can do only what the caller can do. This action is more useful than a RUN_AS mechanism, for example, which prevents this kind of behavior. When you select the **Security** option, the JAAS subject that exists on the scheduling thread is available on the target thread. If not selected, the thread runs anonymously.

Component metadata
Component metadata is relevant only when the asynchronous bean is a simple Java object. If the bean is a Java EE component, such as an enterprise bean, the component metadata is active.

The contexts that can be inherited depend on the work manager used by the application that creates the asynchronous bean. Using the administrative console, the administrator defines the sticky context policy of a work manager by selecting the services on which the work manager is to be made available.

Programming model

Work managers support the following programming models.

- **CommonJ Specification**. The Application Server Version 6.0 CommonJ programming model uses the WorkManager and TimerManager to manage threads and timers asynchronously in the Java EE 1.4 environment.
- **Asynchronous beans and CommonJ specification extensions**. The current asynchronous beans Event Source, asynchronous scopes, subsystem monitors and Java EE Context interfaces are a part of the CommonJ extension.

The following table describes the method mapping between the CommonJ and Asynchronous beans APIs. You can change the current asynchronous beans interfaces to use the CommonJ interface, while maintaining the same functions.

CommonJ package	API	Asynchronous beans package	API
Work manager		Work manager	
Asynchronous beans	Field - IMMEDIATE (long)		Field - IMMEDIATE (int)
	Field - INDEFINITE		Field - INDEFINITE
	schedule(Work) throws WorkException, IllegalArgumentException		startWork(Work) throws WorkException, IllegalArgumentException
	schedule(Work, WorkListener) throws WorkException, IllegalArgumentException **Note:** Configure the work manager work timeout property to the value you previously specified as `timeout_ms` on `startWork`. The default timeout value is INDEFINITE.		startWork(Work, timeout_ms, WorkListener) throws WorkException, IllegalArgumentException
	waitForAll(workItems, timeout_ms)		join(workItems, JOIN_AND, timeout_ms)
	waitForAny(workItems, timeout_ms)		join(workItems, JOIN_OR, timeout_ms)
WorkItem		WorkItem	
	getResult		getResult
	getStatus		getStatus
WorkListener		WorkListener	
	workAccepted(WorkEvent)		workAccepted(WorkEvent)
	workCompleted(WorkEvent)		workCompleted(WorkEvent)
	workRejected(WorkEvent)		workRejected(WorkEvent)
	workStarted(WorkEvent)		workStarted(WorkEvent)
WorkEvent		WorkEvent	
	Field - WORK_ACCEPTED		Field - WORK_ACCEPTED
	Field - WORK_COMPLETED		Field - WORK_COMPLETED
	Field - WORK_REJECTED		Field - WORK_REJECTED
	Field - WORK_STARTED		Field - WORK_STARTED
	getException		getException
	getType		getType
	getWorkItem().getResult() **Note:** This API is valid only after the work is complete.		getWork
Work	(extends Runnable)	Work	(Extends Runnable)
	isDaemon		*
	release		release

RemoteWorkItem	Not in this release. Use Distributed WorkManager in Extended Deployment or future release	NA	
TimerManager		AlarmManager	
	resume		*
	schedule(Listener, Date)		create(Listener, context, time) ** need to convert the parameters
	schedule(Listener, Date, period)		
	schedule(Listener, delay, period)		
	scheduleAtFixedRate(Listener, Date, period)		
	scheduleAtFixedRate(Listener, delay, period)		
	stop		
	suspend		
Timer		Alarm	
	cancel		cancel
	getPeriod		
	getTimerListener		getAlarmListener
	scheduledExecutionTime		
TimerListener		AlarmListener	
	timerExpired(timer)		fired(alarm)
StopTimerListener		Not applicable	
	timerStop(timer)		
CancelTimerListener		Not applicable	
	timerCancel(timer)		
WorkException	(Extends Exception)	WorkException	(Extends WsException)
WorkCompletedException	(Extends WorkException)	WorkCompletedException	(Extends WorkException)
WorkRejectedException	(Extends WorkException)	WorkRejectedException	(Extends WorkException)

For more information on work manager APIs, refer to the Javadoc.

Work manager examples

Table 40. Look up work manager

Asynchronous beans	CommonJ

`InitialContext ctx = new InitialContext();`
`com.ibm.websphere.asynchbeans.WorkManager wm =`
`(com.ibm.websphere.asynchbeans.WorkManager)`
`ctx.lookup("java:comp/env/wm/MyWorkMgr");`
	`InitialContext ctx = new InitialContext();`
	`commonj.work.WorkManager wm =`
	`(commonj.work.WorkManager)`
	`ctx.lookup("java:comp/env/wm/MyWorkMgr");`

Table 41. Create your work using MyWork

Asynchronous beans	CommonJ

Table 41. Create your work using MyWork (continued)

public class MyWork implements
com.ibm.websphere.asynchbeans.Work {
public void release() {
.....
}
public void run() {
System.out.println("Running.....");
}
	public class MyWork implements
	commonj.work.Work{
	public boolean isDaemon() {
	return false;
	}
	public void release () {
	.....
	}
	public void run () {
	System.out.println("Running.....");
	}

Table 42. Submit the work

Asynchronous beans	CommonJ

MyWork work1 = new MyWork(new URI =
"http://www.example./com/1");
MyWork work2 = new MyWork(new URI =
"http://www.example./com/2");
 	
WorkItem item1;
WorkItem item2;
Item1=wm.startWork(work1);
Item2=wm.startWork(work2);
 	
// case 1: block until all items are done
ArrayList col1 = new ArrayList();
Col1.add(item1);
Col1.add(item2);
wm.join(col1, WorkManager.JOIN_AND,
(long)WorkManager.IMMEDIATE);
// when the works are done
System.out.println("work1 data="+work1.getData());
System.out.println("work2 data="+work2.getData());
 	
// case 2: wait for any of the items to complete.
Boolean ret = wm.join(col1,
WorkManager.JOIN_OR, 1000);
	MyWork work1 = new MyWork(new URI =
	"http://www.example./com/1");
	MyWork work2 = new MyWork(new URI =
	"http://www.example./com/2");
	
	WorkItem item1;
	WorkItem item2;
	Item1=wm.schedule(work1);
	Item2=wm.schedule(work2);
	
	// case 1: block until all items are done
	Collection col1 = new ArrayList();
	col1.add(item1);
	col1.add(item2);
	wm.waitForAll(col1, WorkManager.IMMEDIATE);
	// when the works are done
	System.out.println("work1 data="+work1.getData());
	System.out.println("work2 data="+work2.getData());
	
	// case 2: wait for any of the items to complete.
	Collection finished = wm.waitForAny(col1,
	// check the workItems status
	if (finished != null) {
	Iterator I = finished.iterator();
	if (i.hasNext()) {
	WorkItem wi = (WorkItem) i.next();
	if (wi.equals(item1)) {
	System.out.println("work1 =
	"+ work1.getData());
	} else if (wi.equals(item2)) {
	System.out.println("work1 =
	"+ work1.getData());
	}
	}
	}

Table 43. Create a timer manager

Asynchronous beans	CommonJ

Table 43. Create a timer manager (continued)

`InitialContext ctx = new InitialContext();`
`com.ibm.websphere.asynchbeans.WorkManager wm =`
`(com.ibm.websphere.asynchbeans.WorkManager)`
`ctx.lookup("java:comp/env/wm/MyWorkMgr");`
 	
`AsynchScope ascope;`
`Try {`
`Ascope = wm.createAsynchScope("ABScope");`
`} Catch (DuplicateKeyException ex)`
`{`
`Ascope = wm.findAsynchScope("ABScope");`
`ex.printStackTrace();`
`}`
 	
`// get an AlarmManager`
`AlarmManager aMgr= ascope.getAlarmManager();`
	`InitialContext ctx = new InitialContext();`
	`Commonj.timers.TimerManager tm =`
	`(commonj.timers.TimerManager)`
	`ctx.lookup("java:comp/env/tm/MyTimerManager");`

Table 44. Fire the timer

Asynchronous beans	CommonJ

`// create alarm`
`ABAlarmListener listener = new ABAlarmListener();`
`Alarm am =`
`aMgr.create(listener, "SomeContext", 1000*60);`
	`// create Timer`
	`TimerListener listener =`
	`new StockQuoteTimerListener("qqq",`
	`"johndoe@example.com");`
	`Timer timer = tm.schedule(listener, 1000*60);`
	
	`// Fixed-delay: schedule timer to expire in`
	`// 60 seconds from now and repeat every`
	`// hour thereafter.`
	`Timer timer = tm.schedule(listener, 1000*60,`
	`1000*30);`
	
	`// Fixed-rate: schedule timer to expire in`
	`// 60 seconds from now and repeat every`
	`// hour thereafter`
	`Timer timer = tm.scheduleAtFixedRate(listener,`
	`1000*60, 1000*30);`

Timer managers:

The timer manager combines the functions of the asynchronous beans alarm manager and asynchronous scope. So, when a timer manager is created, it internally uses an asynchronous scope to provide the timer manager life cycle functions.

You can look up the timer manager in the Java Naming and Directory Interface (JNDI) name space. This capability is different from the alarm manager that is retrieved through the asynchronous beans scope. Each lookup of the timer manager returns a new logical timer manager that can be destroyed independently of all other timer managers.

A timer manager can be configured with a number of thread pools through the administrative console. For deployment you can bind this timer manager to a resource reference at assembly time, so the resource reference can be used by the application to look up the timer manager.

The Java code to look up the timer manager is:

```
InitialContext ic = new InitialContext();
TimerManager tm = (TimerManager)ic.lookup("java:comp/env/tm/TimerManager");
```

The programming model for setting up the alarm listener and the timer listener is different. The following code example shows that difference.

Table 45. Set up the timer listener

Asynchronous beans	CommonJ

public class ABAlarmListener implements AlarmListener {
public void fired(Alarm alarm) {
System.out.println("Alarm fired. Context =" + alarm.getContext());
}
	public class StockQuoteTimerListener implements TimerListener {
	String context;
	String url;
	public StockQuoteTimerListener(String context,
	String url){
	this.context = context;
	This.url = url;
	}
	public void timerExpired(Timer timer) {
	System.out.println("Timer fired. Context ="+ ((StockQuoteTimerListener)timer.getTimerListener()) .getContext());
	}
	public String getContext() {
	return context;
	}
	}

Example: Using connections with asynchronous beans:

An asynchronous bean method can use the connections that its creating Java Platform, Enterprise Edition (Java EE) component obtained using java:comp resource references.

For more information on resource references, see the topic References. The following is an example of an asynchronous bean that uses connections correctly:

```
class GoodAsynchBean
{
 DataSource ds;
 public GoodAsynchBean()
  throws NamingException
 {
  // ok to cache a connection factory or datasource
  // as class instance data.
  InitialContext ic = new InitialContext();
  // it is assumed that the created Java EE component has this
  // resource reference defined in its deployment descriptor.
  ds = (DataSource)ic.lookup("java:comp/env/jdbc/myDataSource");
 }
 // When the asynchronous bean method is called, get a connection,
 //  use it, then close it.
 void anEventListener()
 {
  Connection c = null;
  try
  {
   c = ds.getConnection();
   // use the connection now...
  }
  finally
  {
   if(c != null) c.close();
  }
 }
}
```

The following example of an asynchronous bean that uses connections incorrectly:

```
class BadAsynchBean
{
 DataSource ds;
 // Do not do this. You cannot cache connections across asynch method calls.
 Connection c;

 public BadAsynchBean()
  throws NamingException
 {
  // ok to cache a connection factory or datasource as
  // class instance data.
  InitialContext ic = new InitialContext();
  ds = (DataSource)ic.lookup("java:comp/env/jdbc/myDataSource");
  // here, you broke the rules...
  c = ds.getConnection();
 }
 // Now when the asynch method is called, illegally use the cached connection
 // and you likely see J2C related exceptions at run time.
 // close it.
 void someAsynchMethod()
 {
  // use the connection now...
 }
}
```

Work manager service settings

Use this page to enable or disable the work manager service that manages work manager resources used by the server.

To view this administrative console page, click **Servers > Application Servers >** *server_name* **> Work Manager Service** .

Startup:

Specifies whether the server attempts to start the work manager service.

Default Selected

Range

Selected
When the application server starts, it attempts to start the work manager service automatically.

Cleared
The server does not try to start the work manager service. If work manager resources are to be used on this server, the system administrator must start the work manager service manually or select this property then restart the server.

Assembling applications that use work managers and timer managers

The work manager and timer manager objects are both supported for assembling applications that implement the asynchronous bean technology. You can assemble either work managers or time managers.

Before you begin

Configure at least one work manager or timer manager using the administrative console.

About this task

Complete the steps to either assemble work managers or time managers.

1. Assemble applications that use asynchronous beans work managers.
2. Assemble applications that use CommonJ work managers.

3. Assemble applications that use CommonJ timer managers.

Assembling applications that use a CommonJ WorkManager

When a work manager has been configured, if it references a logical work manager it must be bound to a physical work manager using an assembly tool. Then resources can be created and bound to a physical work manager.

Before you begin

Your administrator needs to configure at least one work manager using the administrative console.

About this task

If your application references one or more logical work managers, the logical work managers must be bound to one or more physical work managers using an assembly tool, such as Rational Web Developer.

1. Declare a resource reference for each work manager (required action by the application developer). This forms an EAR file. (For more information on resource references, see the topic References.)
2. Bind each resource reference to a physical work manager, using an assembly tool, such as Rational Web Developer.
3. Add a resource reference with the type commonj.work.WorkManager to the application deployment descriptor. The application can look up this work manager using its resource reference name in java:comp. Now, you can use an assembly tool or Rational Application Developer to specify which resource references are bound to the physical commonj.work.WorkManager.

   Note: The previous steps outline the same process used for data sources.

Assembling applications that use timer managers

When a work manager has been configured, if it references a logical work manager it must be bound to a physical work manager using an assembly tool. Then resources can be created and bound to a physical timer.

Before you begin

Your administrator needs to configure at least one timer manager using the administrative console.

About this task

If your application references one or more logical timer managers, the logical timer managers must be bound to one or more physical timer managers using an assembly tool, such as the Rational Application Developer.

1. Declare a resource reference for each timer manager (required action by the application developer). This forms an EAR file. (For more information on resource references, see the topic References.)
2. Bind each resource reference to a physical timer manager, using an assembly tool.
3. Add a resource reference with the type commonj.timers.TimerManager to the application deployment descriptor. The application then can look up this timer manager using its resource reference name in java:comp. The assembly tool can specify which resource references are bound to a physical timer manager.

   Note: The previous steps outline the same process used for data sources.

Assembling applications that use asynchronous beans work managers

When a work manager has been configured, if it references a logical work manager it must be bound to a physical work manager using an assembly tool. Then resources can be created and bound to a physical work managers.

Before you begin

Your administrator needs to configure at least one work manager using the administrative console.

About this task

If your application references one or more logical work managers, the logical work managers must be bound to one or more physical work managers using an assembly tool.

The CommonJ 1.1 interfaces are supported. Both asynchronous beans and CommonJ interfaces can use one configuration work manager object. The type of interface implemented is resolved during the JNDI lookup time. The type of interface used is determined by the one specified in the resource-reference, instead of the one specified in the configuration object. So, there can be one resource-reference for each interface, per configuration object. Each resource-reference lookup returns the appropriate type of instance. For example, there are two resource-references defined for the `wm/MyWorkManager:` `wm/ABWorkMgr` and `wm/CommonJWorkMgr`. The WebSphere Application Server run time returns the correct interface for each resource-reference lookup.

1. Declare a resource reference for each work manager (required action by the application developer). This action results in an EAR file. For more information on resource references, see the topic References.
2. Use an assembly tool to bind each resource reference to a physical work manager.
3. Add a resource reference with the type com.ibm.websphere.asynchbeans.WorkManager to the application deployment descriptor. The application then can look up this work manager using its resource reference name in java:comp. The assembly tool or Rational Application Developer then can specify which resource references are bound to a physical work manager.

   Note: Use the same previous steps to configure data sources.

Developing work objects to run code in parallel

You can run work objects in parallel, or in a different J2EE context, by wrapping the code in a work object.

Before you begin

Your administrator must have configured at least one work manager using the administrative console.

About this task

To run code in parallel, wrap the code in a work object.

1. Create a work object.

   A work object implements the com.ibm.websphere.asynchbeans.Work interface. For example:

   ```
   class SampleWork implements Work
   ```

2. Determine the number of work managers needed by this application component.
3. Look up the work manager or managers using the work manager resource reference (or logical name) in the java:comp namespace. (For more information on resource references, see the topic References.)

   ```
   InitialContext ic = new InitialContext();
   WorkManager wm = (WorkManager)ic.lookup("java:comp/env/wm/myWorkManager");
   ```

   The resource reference for the work manager (in this case, `wm/myWorkManager`) must be declared as a resource reference in the application deployment descriptor.
4. Call the WorkManager.startWork() method using the work object as a parameter. For example:

   ```
   Work w = new MyWork(...);
   WorkItem wi = wm.startWork(w);
   ```

The startWork() method can take a startTimeout parameter. This specifies a hard time limit for the Work object to be started. The startWork() method returns a work item object. This object is a handle that provides a link from the component to the now running work object.

5. [Optional] If your application component needs to wait for one or more of its running work objects to complete, call the WorkManager.join() method. For example:

```
WorkItem wiA = wm.start(workA);
WorkItem wiB = wm.start(workB);
ArrayList l = new ArrayList();
l.add(wiA);
l.add(wiB);
if(wm.join(l, wm.JOIN_AND, 5000)) // block for up to 5 seconds
{

// both wiA and wiB finished
}
else
{

// timeout

// we can check wiA.getStatus or wiB.getStatus to see which, if any, finished.
}
```

   This method takes an array list of work items which your component wants to wait on and a flag that indicates whether the component will wait for one or all of the work objects to complete. You also can specify a timeout value.

6. Use the release() method to signal the unit of work to stop running. The unit of work then attempts to stop running as soon as possible. Typically, this action is completed by toggling a flag using a thread-safe approach like the following example:

```
public synchronized void release()
{
 released = true;
}
```

   The Work.run() method can periodically examine this variable to check whether the loop exits or not.

Work objects

A work object is a type of asynchronous bean used by application components to run code in parallel or in a different Java Platform, Enterprise Edition (Java EE) context.

A work object implements the com.ibm.websphere.asynchbeans.Work interface. A work object is essentially a java.lang.Runnable object that is serializable and provides additional methods. For details, see the Interface Work in the generated API documentation.

A component wanting to run work in parallel, or in a different Java EE context, locates a work manager in Java™ Naming and Directory Interface (JNDI), then calls the WorkManager.startWork() method using the work object as a parameter.

The startWork() method returns a work item object. This object is a handle that provides a link from the component to the now running work object. The work item object is typically used when the component needs to wait for one or more of its running work objects to complete. The WorkManager.join() method takes an array list of work items that the component wants to wait on, and a flag indicating whether the component will wait for all or one of the work objects to complete. A timeout can be specified, which prevents the component from waiting indefinitely.

The application does not create Java SE Development Kit 6 (JDK 6) threads because they are not managed threads. Plus, these threads are not affiliated with the Java EE environment, which makes them useless inside an application server. In addition, these threads have no Java EE context (for example, a

java:comp) and are not authenticated when they fire. Work object threads are fully supported by the application server and have the same properties as other asynchronous beans.

Example: Creating work objects

You can create a work object that dynamically subscribes to a topic and any component that has access to the event source can add an event on demand.

The following is an example of a work object that dynamically subscribes to a topic:

```
class SampleWork implements Work
{
 boolean released;
 Topic targetTopic;
 EventSource es;
 TopicConnectionFactory tcf;

 public SampleWork(TopicConnectionFactory tcf, EventSource es, Topic targetTopic)
 {
  released = false;
  this.targetTopic = targetTopic;
  this.es = es;
  this.tcf = tcf;
 }

 synchronized boolean getReleased()
 {
  return released;
 }

 public void run()
 {
  try
  {
   // setup our JMS stuff.
   TopicConnection tc = tcf.createConnection();
   TopicSession sess = tc.createSession(false, Session.AUTOACK);
   tc.start();

   MessageListener proxy = es.getEventTrigger(MessageListener.class, false);
   while(!getReleased())
   {
    // block for up to 5 seconds.
    Message msg = sess.receiveMessage(5000);
    if(msg != null)
    {
     // fire an event when we get a message
     proxy.onMessage(msg);
    }
   }
   tc.close();
  }
  catch (JMSException ex)
  {
   // handle the exception here
   throw ex;
  }
  finally
  {
   if (tc != null)
   {
    try
    {
     tc.close();
    }
    catch (JMSExceptin ex1)
    {
```

```
     // handle exception
    }
   }
  }
 }

 // called when we want to stop the Work object.
 public synchronized void release()
 {
  released = true;
 }
}
```

As a result, any component that has access to the event source can add an event on demand, which allows components to subscribe to a topic in a more scalable way than by simply giving each client subscriber its own thread. The previous example is fully explored in the WebSphere Trader Sample. See the Samples Gallery for details.

Developing event listeners

Application components that listen for events can use the EventSource.addListener() method to register an event listener object (a type of asynchronous bean) with the event source to which the events will be published. An event source also can fire events in a type-safe manner using any interface.

About this task

Notifications between components within a single EAR file are handled by a special event source. See the topic, Using the application notification service.

1. Create an event listener object, which can be any type. For example, see the following interface code:

```
interface SampleEventGroup
{
  void finished(String message);
}

class myListener implements SampleEventGroup
{
  public void finished(String message)
  {
  // This will be called when we 'finish'.
  }
}
```

2. Register the event listener object with the event source. For example, see the following code:

```
InitialContext ic = ...;
EventSource es = (EventSource)ic.lookup("java:comp/websphere/ApplicationNotificationService");
myListener l = new myListener();
es.addListener(l);
```

   This enables the myListener.finished() method to be called whenever the event is fired. The following code example shows how this event might be fired:

```
InitialContext ic = ...;
EventSource es = (EventSource)ic.lookup("java:comp/websphere/ApplicationNotificationService");
myListener proxy = es.getEventTrigger(myListener.class);
// fire the 'event' by calling the method
// representing the event on the proxy
proxy.finished("done");
```

Using the application notification service

During the application lifetime, individual J2EE components (servlets or enterprise beans) within a single EAR file might need to signal each other. There is an event source in the java:comp namespace that is bound into all components within an EAR file that can be used for notification.

About this task

The JNDI name for this event source, in the java:comp namespace that is bound into all components within an EAR file, is:

```
java:comp/websphere/ApplicationNotificationService
```

Components within the same application can fire asynchronous events and register event listeners using this application notification service. Startup beans can be used to register these event listeners at application startup or they can be registered dynamically at run time.

To have your enterprise bean or servlet use the application notification service, write code similar to the following example:

```
InitialContext ic = new InitialContext();
EventSource appES = (EventSource)
  ic.lookup("java:comp/websphere/ApplicationNotificationService");
// now, the application can add a listener using the EventSource.addListener method.
// MyEventType is an interface.
MyEventType myListener = ...;
AppES.addListener(myListener);

// later another component can fire events as follows
InitialContext ic = new InitialContext();
EventSource appES = (EventSource)
ic.lookup("java:comp/websphere/ApplicationNotificationService");

// This highlights a constant string on the EventSource interface which
// specifies the 'java:comp/websphere/ApplicationNotificationService' string.
ic.lookup(appES.APPLICATION_NOTIFICATION_EVENT_SOURCE)
// now, the application can add a listener using the EventSource.addListener method.
MyEventType proxy = appES.getEventTrigger(MyEventType.class, false);
proxy.someEvent(someArguments);
```

Example

Example: Firing a listenerCountChanged event

You can fire a listenerCountChanged event that produces a proxy for the interface on which the method fires. Calling the method corresponding to the event on the proxy implements the EventSourceEvents interface. The same proxy can be used to send multiple events simultaneously.

The following code example demonstrates how to fire a listenerCountChanged event:

```
// imagine this snippet inside an EJB or servlet method.
// Make an inner class implementing the required event interfaces.
EventSourceEvents listener = new Object() implements EventSourceEvents.class
{
 void listenerCountChanged(EventSource es, int old, int newCount)
 {
  try
  {
     InitialContext ic = new InitialContext();
   // Here, the asynchronous bean can access an environment variable of
   // the component which created it.
    int i = (Integer)ic.lookup("java:comp/env/countValue").intValue());
   if(newCount == i)
   {
    // do something interesting
   }
   // call this event when the following code executes:
  }
  catch(NamingException e)
  {
  }
 }
```

```
 void listenerExceptionThrown( EventSource es, Object listener,
       String methodName, Throwable exception)
 {
 }
 void unexpectedException(EventSource es, Object runnable, Throwable exception)
 {
 }
}
// register it.
es.addListener(listener);

...

// now fire an event which the previous listener receives.
EventSourceEvents proxy = (EventSourceEvents)
   es.getEventTrigger(EventSourceEvents.class, false);

proxy.listenerCountChanged(es, 0, 1);

// now, fire another event, you can call any of the methods.
proxy.listenerCountChanged(es, 4, 5);
```

The output in this example is a proxy for the interface on which the method fires. Then, call the method corresponding to the event on the proxy. This action causes the same method with the same parameters to be called on any event listeners that implement the EventSourceEvents interface and that were previously registered with the EventSource ″es″. The same proxy can be used to send multiple events simultaneously.

The boolean parameter on the getEventTrigger() method is `sameTransaction`. When the `sameTransaction` parameter is `false`, a new transaction is started for each event listener invoked and these event listeners can be called in parallel to the caller. However, the event() method is blocked until all of the event listeners are notified. If the `sameTransaction` parameter is `true`, then the current transaction (if any) on the thread is used for all of the event listeners. The event listeners share the transaction of the method that fired the event. For that reason, all event listeners must run serially in an undetermined order. The order that listeners are called is undefined, and the order in which listeners are registered does not act as a guide for the order used at run time. The method on the proxy does not return until all of the event listeners are called, which means that this action is a synchronous operation.

The parameters that references and listeners pass do not interfere with the function of these references, unless you configure the method to do so. For example, event listeners can be used as collaborators and add data to a map, which was a parameter. Each event listener runs on its own transaction, independent of any transaction that is active on the thread. Extreme care must be taken when the sameTransaction parameter is `false` because the parameters can be accessed by multiple threads.

Developing asynchronous scopes

`Asynchronous scopes` are units of scoping that comprise a set of alarms, subsystem monitors, and child asynchronous scopes. You can create asynchronous scopes, starting with the parent.

About this task

Using asynchronous scopes can involve some or all of the following steps:

1. Create asynchronous scopes. Create the parent asynchronous scope object by using a unique parameter name that calls the AsynchScopeManager.createAsynchScope() method. You can store properties in an asynchronous scope object. This storage provides Java 2 Enterprise Edition (J2EE) applications with a way to store a non-serializable state that otherwise cannot be stored in a session bean. You also can create child asynchronous scopes, which is useful for scoping data beneath the parent.
2. Listen for alarm notifications

a. Create a listener object by implementing the AlarmListener interface. For more information, see the AlarmListener interface in the generated API documentation.
b. Supply this object to the AlarmManager.create() method, as the target for the alarm. The create() method takes the following parameters:
   Target for the alarm
   The target on which the fired() method is called when the alarm is fired.
   Context
   The context object for the alarm. This object is useful for supplying alarm-specific data to the listener and supports a single listener for multiple alarms.
   Interval
   The number of milliseconds before the alarm fires.

   After the specified interval, the alarm fires and the fired() method of the listener is called with the firing alarm as a parameter. The alarm object is returned. By calling methods on this object, you can cancel or reschedule the alarm.

3. Monitor remote systems.
   a. Implement a mechanism for detecting messages sent from the remote system. For example, publish and subscribe messaging.
   b. Create a subsystem manager object by calling the SubsystemMonitorManager.create() method with the following parameters:
      Name Each subsystem monitor must have a unique name.
      Heartbeat interval
      The expected interval, in milliseconds, between heartbeats.
      Missed heart beats until stale or suspect
      The number of heartbeats that can be missed before the subsystem is marked as stale.
      Missed heart beats until dead
      The number of heartbeats that can be missed before the system is marked as dead.
   c. Create an object that implements the SubsystemMonitorEvents interface. For more information, see the SubsystemMonitorEvents in the generated API documentation.
   d. Add an instance of this object to the subsystem monitor using the SubsystemMonitor.addListener() method.
   e. Whenever a heartbeat message arrives from the remote system, call the SubsystemMonitor ping() method.

   The subsystem monitor configures alarms to track the heartbeat status of the remote system. When the ping() method is called, the alarms are reset. If an alarm fires, the ping() method is not called; that is, the application did not receive a heartbeat from the monitored subsystem.

Example

Asynchronous scopes are useful in stateful server applications. An application can have a startup bean that creates an asynchronous scope on a named work manager. The application also might create subsystem monitors to monitor the health of any remote systems on which the application is dependent.

When a client attaches to the server, the application creates a child asynchronous scope that is owned by the application asynchronous scope for the client and named using the client ID. A subsystem monitor for monitoring the client might be created on the client asynchronous scope. If the client times out, a callback can clean up the client state on the server. Callbacks can be attached to the application subsystem monitors, on behalf of the client. When a remote system becomes unavailable, the client code in the server is notified and an event is sent to the client to warn that a critical remote system has failed. For example, the failure might be a data feed in an electronic trading application.

Asynchronous scopes

An asynchronous scope (AsynchScope object) is a unit of scoping provided for use with asynchronous beans.

Asynchronous scopes are collections of alarms, subsystem monitors, and child asynchronous scopes that enable a relationship to form. Each asynchronous scope uses a single work manager.

Each AsynchScope object owns and controls the life cycle of the following objects:

Child asynchronous scopes

Each AsynchScope object extends the AsynchScopeManager interface, which is a factory for AsynchScope objects. (For more information on the AsynchScopeManager interface, see the generated API documentation). Any asynchronous scope can therefore create named asynchronous scopes (children). Child asynchronous scopes can be useful for scoping data underneath the parent. All of the child asynchronous scopes must be uniquely named. These children are destroyed if the parent asynchronous scope is destroyed.

Alarms

Each asynchronous scope has an associated alarm manager. All of the alarms created by the alarm manager are automatically cancelled if the associated asynchronous scope is destroyed.

Subsystem monitors

Each asynchronous scope has a subsystem monitor manager, which manages a set of subsystem monitors associated with the asynchronous scope. When the asynchronous scope is destroyed, all of the associated subsystem monitors also are destroyed.

In summary, asynchronous scopes can be organized into an acyclic tree. The life cycle of each asynchronous scope is directly coupled to that of its parent asynchronous scope. Each asynchronous scope is associated with a set of alarms and subsystem monitors, and an optional set of child asynchronous scopes. These objects are cancelled and destroyed when the asynchronous scope is destroyed.

Asynchronous scope state

Each asynchronous scope has an associated map, in which applications can store their state in the form of name and value pairs.

Asynchronous scope events

Each asynchronous scope is also an event source. Applications can therefore register event listeners against the asynchronous scope. The event listeners can receive notification if, for example, the AsynchScope object is about to be destroyed.

Applications also can use this event source to fire events only to listeners of this asynchronous scope. For example, an AsynchScope object created for a client session might be used to fire asynchronous events to parties interested in that client.

Alarms

An alarm runs Java Platform, Enterprise Edition (Java EE) context-aware code at a given time interval. Alarm objects are fine-grained, nonpersistent, transient, and can fire at millisecond intervals.

Alarms are run using a thread pool associated with the work manager that owns the associated asynchronous scope. You must create a work manager instance to create an alarm. See the topic Configuring work managers for more information.

The AlarmManager.createAlarm() method takes an application-written object that implements the AlarmListener interface. For more information on the AlarmListener interface, see the generated API documentation. The fired method is called when the alarm expires. The createAlarm() method returns a non-serializable handle, which can be used to cancel or reset the alarm. All of the pending alarms are cancelled when its associated AsynchScope object is destroyed.

Note: The Java SE Development Kit 6 (JDK6)already has a timer mechanism, so why create a new one? The JDK 6 is a Java Platform, Standard Edition (Java SE) feature that knows nothing about the

Java EE environment. Timers fired by the Java SE feature do not run on a managed thread and are therefore unusable inside an application server. These timers also lack a Java EE context (that is, a java:comp value) and are not authenticated when they fire. The asynchronous scope alarms are fully supported by the product and have the same properties as any other asynchronous bean.

Alarm performance

The alarm subsystem is designed to handle a large number of alarms. However, do not expect alarms to process heavy loads when they are firing because this activity slows the processing of later alarms. If an alarm needs to process a heavy load, design a work object that is activated by a work manager. This procedure moves the heavy processing to a different thread and enables the alarm threads to process alarms unhampered. All of the alarms owned by asynchronous scopes that are owned by a single work manager share a common thread pool. The properties of this thread pool can be tuned at the work manager level using the administrative console.

Subsystem monitors

A subsystem monitor is an object that monitors the health of a remote system. It uses an event source to inform all registered listeners of the health of the system.

AdvancedJava Platform, Enterprise Edition (Java EE) applications often rely on remote, non-managed, non-Java EE systems. These remote systems can periodically send clients a message to indicate that they are working. A subsystem monitor is a set of alarms that tracks indicator messages or heart beats from a remote system.

An application creates a subsystem monitor by calling the SubsystemMonitorManager.create() method with the following parameters:

Name Each subsystem monitor must be uniquely named.

Heart beat interval
The time period, in milliseconds, between arriving heart beat messages.

Missed heart beats until stale or suspect
The number of heart beats that can be missed before the subsystem is marked as stale. This designation indicates that the subsystem might be having problems.

Missed heart beats until dead
The number of heart beats that can be missed before the system is considered down. The system then is marked as dead.

The subsystem monitor configures alarms to track the heart beat status. Whenever the ping() method is called, the alarms are reset. If an alarm fires, the ping() method has not been called; that is, the application did not receive a heart beat from the monitored subsystem. When the number of **Missed heart beats until stale** value has elapsed without a ping, a stale event is fired. Later, if the number of **Missed heart beats until dead** value elapses without a ping, a dead event is fired. If a ping is received after a stale or dead notification, a fresh event is sent, which indicates that the subsystem is alive again.

Make the **Missed heart beats until dead** value greater or equal to the **Missed heart beats until stale** value. If **Missed heart beats until stale** value equals the **Missed heart beats until dead** value, then a stale event is not published. Only a dead event is published.

You can register a listener that implements the SubsystemMonitorEvents interface for applications that require notification of events. For more information on the SybsystemMonitorEvents interface, see the generated API documentation.

Heart beat messages can be transmitted using a variety of mechanisms. The application must call the SubsystemMonitor ping() method whenever a heart beat message arrives from a remote system, but the method used to detect these messages is up to the application. For example, you might use a Java Message Service (JMS) publish or subscribe implementation or even a third-party Java messaging product that does not implement JMS.

Asynchronous scopes: Dynamic message bean scenario

Java Platform, Enterprise Edition (Java EE) now supports message-driven beans, but the beans are static. This scenario provides information about how to set up the environment to enable the dynamic message bean.

All of the message sources must be known in advance and bound at deployment time. This action is not always viable, especially in fluid messaging environments such as those found in brokerages. Some environments have publish-subscribe topic spaces that are continually changing and clients need servers that can subscribe on demand to an arbitrary topic.

An asynchronous bean application can create a work object that performs a blocking receive on a Java Message Service (JMS) topic and then publishes the message as an event on an application-defined event source. Clients requiring a subscription to that message can add an event listener to the event source. The event source can inform the work object when there are no listeners. Then, the event source can shut down and make the JMS and thread resources available. The work object registers a listener with its own event source. When the count is one again, the work object knows that it is the only listener and it is time to shut down the work object. The WebSphere Trader Sample (see your installed Samples Gallery) uses this pattern to dynamically subscribe to JMS topics at run time to gather stock prices. For more information, see an overview of the samples.

How does the server catch clients that disconnect or crash? It creates a subsystem monitor to watch the client and adds an event listener to catch dead events. When a dead event occurs, the server application can clean up the client server state. For example, the server application can remove the client event listener from the dynamic message bean, thereby allowing the server to subscribe to a dynamic topic only when it is needed.

Dynamic cache

Task overview: Using the dynamic cache service to improve performance

Caching the output of servlets, commands, and JavaServer Pages (JSP) improves application performance. WebSphere Application Server consolidates several caching activities including servlets, Web services, and WebSphere commands into one service called the *dynamic cache*. These caching activities work together to improve application performance, and share many configuration parameters that are set in the dynamic cache service of an application server. You can use the dynamic cache to improve the performance of servlet and JSP files by serving requests from an in-memory cache. Cache entries contain servlet output, the results of a servlet after it runs, and metadata.

About this task

The dynamic cache service works within an application server Java virtual machine (JVM), intercepting calls to cacheable objects. For example, it intercepts calls through a servlet service method or a command execute method, and either stores the output of the object to the cache or serves the content of the object from the dynamic cache.

1. Enable the dynamic cache service globally. To use the features associated with dynamic caching, you must enable the service in the administrative console. See Using the dynamic cache service for more information.
2. Configure the type of caching that you are using:
   - Configuring servlet caching.
   - Configuring portlet fragment caching.
   - Configuring Edge Side Include caching.
   - Configuring command caching.
   - “Example: Caching Web services” on page 1551.

- Configuring the Web services client cache.

3. Monitor the results of your configuration using the dynamic cache monitor. For more information, see Displaying cache information.
4. If you have any problems with your configuration, see the *Troubleshooting and support* PDF.

What to do next

To use the DistributedMap and DistributedObjectCache interfaces for the dynamic cache, see “Using the DistributedMap and DistributedObjectCache interfaces for the dynamic cache” on page 1561.

Disk cache infrastructure enhancements

Several performance enhancements are available for the dynamic cache service.

The dynamic cache service supports persisting objects to disk (specified by a file system location) so that objects that are evicted from the memory cache are not regenerated by the application server. Objects are written to disk when they are evicted from memory using a Least Recently Used (LRU) eviction algorithm. The objects in the memory cache may also be flushed to disk on normal server shutdown. Java objects that need to be offloaded to the disk should be serializable.

The disk offload function includes the following functions:

- An internal disk cache format for faster deletions and support for new options to limit disk cache size
- The disk cache garbage collector, which evicts objects out of the cache when a configured high threshold is reached
- Four new performance modes to tune your disk cache performance:
  - High performance/memory usage mode - keeps all metadata in system memory and provides the highest performance
  - Balanced performance/memory usage mode - provides optimal balance of performance and memory usage by keeping some metadata in system memory
  - Custom performance/memory usage mode - allows explicit configuration of the memory usage and customization of performance requirements
  - Low performance/memory usage mode - stores most of the metadata on disk for users who are very constrained on system memory

Limiting the disk cache. The dynamic cache service provides mechanisms to limit the use of the disk cache by specifying the size of the disk cache in gigabytes, in addition to the maximum number of entries that are persisted to the disk. The disk cache is considered full when either of these limits is reached and forms the basis for eviction of objects from the disk. If the cache subsystem cannot offload any more data to disk, due to either an out-of-disk space condition, insufficient space on disk, or an exception when writing data to disk as a result of a possibly corrupt disk, the disk offload capability is disabled to prevent data integrity problems. The event is logged and the disk cache subsystem is deleted. This prevents serving corrupt data from the cache on a restart. If the option to persist cache data is turned on, some information such as dependency and template information is flushed to disk on a server shutdown. If a disk full situation occurs during this shutdown process, any partially-persisted and un-persisted dependency or template data is removed from the cache. A side effect of this, to preserve integrity, is to invalidate the cached objects that are associated with the dependency or template data.

Disk cache size in GB. The disk cache size in GB option pertains primarily to the object data (which includes the cached object, its identifier, and metadata such as expiration time), template information and dependency information that are written to disk. The cache subsystem allocates separate storage and volumes (each of which can grow to 1 GB) for object data, templates and dependencies, as needed. When the total number of volumes on disk exceeds the specified cache size, any subsequent data that is written to disk is discarded until more space is made available by the disk cache garbage collector. To preserve data integrity, any information that is related to discarded objects is invalidated as well. The thresholds for garbage collection (described below) and the disk cache full state are associated with the

space available for object data. It is also possible that in certain, rare scenarios, as information is flushed to disk, critical system data needs to be written to disk, which may cause the total file system space required to exceed up to 5% of the specified maximum limit. It is recommended that there be at least 25% of actual file system space available for disk caching over and above the specified disk cache size in GB. It is also required that each cache instance has a unique disk offload location and it is recommended that each offload location be on a dedicated disk partition. The cache file system employs a logical file manager to manage storage allocation for cached objects, therefore the file system size or the size of the files in the cache directory may not be an accurate gauge of the available space for the cache subsystem. At the same time, because of the adjusted limit, the cache subsystem may encounter a cache full state prior to the approaching the specified maximum limit as measured in allocated file system space. The PMI counters provide a better picture of how full the cache is.

Related concepts

"Eviction policies using the disk cache garbage collector"
The disk cache garbage collector is responsible for evicting objects out of the disk cache, based on a specified eviction policy.

Related reference

"Dynamic cache PMI counter definitions" on page 1556
The dynamic cache statistics interface is defined as WSDynamicCacheStats under the `com\ibm\websphere\pmi\stat` package.

"Dynamic cache MBean statistics" on page 1554
The dynamic cache service provides an MBean interface to access cache statistics.

Eviction policies using the disk cache garbage collector

The disk cache garbage collector is responsible for evicting objects out of the disk cache, based on a specified eviction policy.

The garbage collector keeps a certain amount of space on disk available, which is governed by the configuration attribute that limits the amount of disk space that is used for caching objects. To enable the eviction policy, enable the Limit disk cache size in GB and/or Limit disk cache size in entries options in the administrative console.

The garbage collector is triggered when the disk space reaches a specified high threshold (a percentage of the Limit disk cache size in entries or in GB) and evicts objects, based on the eviction policy, from the disk in the background until the disk cache size reaches a specified low threshold (a percentage of the Limit disk cache size in entries or in GB). Eviction triggers when one or both of the high thresholds is reached for Limit disk cache size in GB and Limit disk cache size in entries. The supported policies are:

- None: This is the default policy. Objects are evicted only when they expire, or if they are invalidated.
- Random: The expired objects are removed first. If the disk size still has not reached the low threshold limit, objects are picked from the disk cache in random order and removed until the disk size reaches a low threshold limit.
- Size: The expired objects are removed first. If the disk size still has not reached the low threshold limit, then largest-sized objects are removed until the disk size reaches a low threshold limit.

Limit disk cache size in GB and High Threshold determines when to trigger eviction and when the disk cache is considered near full. It is computed as a function of the user-specified limit. If the specified limit is *10* GB (3 GB is the minimum), the cache subsystem initially creates three files that can grow to 1 GB in size for cache data, dependency ID information, and template information. Each time more space is needed to contain cache data, dependency ID information, or template information, a new file is created. Each of these files grow in 1 GB increments until the total number of files that are created is equal to disk cache in size in GB (in this case ten). Although the initial size of the new file may be much smaller than 1 GB, the dynamic cache service always rounds up to the next GB.

Eviction triggers when the cache data size reaches the high threshold and continues until the cache data size reaches the low threshold. Calculation of cache data size is dynamic. The following formula describes how to calculate the actual cache data size limit:

```
cache data size limit = disk cache size (in GB) - number of dependency files per GB - number of template files
```

When the cache data size limit is defined, the trigger point is calculated as follows:

```
eviction trigger point = cache data size limit * high threshold
size of evicted entries = cache data size * (high threshold - low threshold)
```

Consider the following scenarios:

- **Scenario 1**
  - Disk cache size in GB = 10 GB
  - High threshold = 90%
  - Low Threshold = 80%

  Initially, there is one file for dependency ID and template ID.

  ```
  cache data size limit = 10-(1+1) = 8 GB
  eviction trigger point = 8 * 90% = 7.2 GB
  size of evicted entries = 8 * (90% - 80%) = 0.8 GB
  ```

  In the above scenario, eviction starts when the data cache size reaches 7.2 GB and continues until the cache size is 6.4 GB (7.2 - 0.8).
- **Scenario 2**

  In scenario 1, if the dependency files grow to more than 1 GB, an additional dependency file generates. The eviction trigger point launches dynamically as follows:

  ```
  cache data size limit = 10 - (2+1) = 7GB
  eviction trigger point = 7 * 90% = 6.3GB
  size of evicted entries = 7 * (90% - 80%) = 0.7GB
  ```

  In the above scenario, eviction starts when the data cache size reaches 6.3 GB, and continues until the cache size in 5.6 GB (6.3 - 0.7).

Disk cache eviction for limit disk cache size in entries. Consider the following scenario:

- Disk cache size in entries = 100000
- High threshold = 90%
- Low threshold = 80%

```
eviction trigger point = 100000 * 90% = 90000
number of entries evicted = 100000 * (90% - 80%) = 10000
```

In this scenario, eviction starts when the number of cache entries reaches 90000 and 10000 entries are evicted from the cache.

Example: Caching Web services

This topic includes examples of building a set of cache policies and SOAP messages for a Web services application.

The following is a example of building a set of cache policies for a simple Web services application. The application in this example stores stock quotes and has operations to read, update the price of, and buy a given stock symbol.

Following are two SOAP message examples that the application can receive, with accompanying HTTP Request headers.

The first message sample contains a SOAP message for a GetQuote operation, requesting a quote for IBM. This is a read-only operation that gets its data from the back end, and is a good candidate for caching. In this example the SOAP message is cached and a timeout is placed on its entries to guarantee the quotes it returns are current.

Message example 1

```
POST /soap/servlet/soaprouter
HTTP/1.1
Host: www.myhost.com
Content-Type: text/xml; charset="utf-8"
SOAPAction: urn:stockquote-lookup
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:getQuote xmlns:m="urn:stockquote">
<symbol>IBM</symbol>
</m:getQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

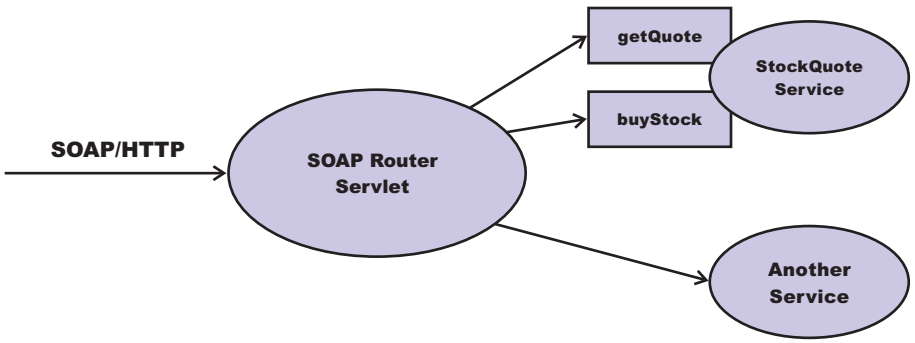
The SOAPAction HTTP header in the request is defined in the SOAP specification and is used by HTTP proxy servers to dispatch requests to particular HTTP servers. WebSphere Application Server dynamic cache can use this header in its cache policies to build IDs without having to parse the SOAP message.

Message example 2 illustrates a SOAP message for a BuyQuote operation. While message 1 is cacheable, this message is not, because it updates the back end database.

Message example 2

```
POST /soap/servlet/soaprouter
HTTP/1.1
Host: www.myhost.com
Content-Type: text/xml; charset="utf-8"
SOAPAction: urn:stockquote-update
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:buyStock xmlns:m="urn:stockquote">
<symbol>IBM</symbol>
</m:buyStock>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following graphic illustrates how to invoke methods with the SOAP messages. In Web services terms, especially Web Service Definition Language (WSDL), a service is a collection of operations such as getQuote and buyStock. A body element namespace (urn:stockquote in the example) defines a service, and the name of the first body element indicates the operation.



The following is an example of WSDL for the getQuote operation:

```
<?xml version="1.0"?>
<definitions name="StockQuoteService-interface"
targetNamespace="http://www.getquote.com/StockQuoteService-interface"
xmlns:tns="http://www.getquote.com/StockQuoteService-interface"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns=soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
<message name="SymbolRequest">
<part name="return" type="xsd:string"/>
</message>
<portType name="StockQuoteService">
<operation name="getQuote">
<input message="tns:SymbolRequest"/>
<output message="tns:QuoteResponse"/>
</operation>
</portType>
<binding name="StockQuoteServiceBinding"
type="tns:StockQuoteService">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="getQuote">
<soap:operation soapAction="urn:stockquote-lookup"/>
<input>
<soap:body use="encoded" namespace="urn:stockquote"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</input>
<output>
<soap:body use="encoded" namespace="urn:stockquotes"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</output>
</operation>
</binding>
</definition>
```

To build a set of cache policies for a Web services application, configure WebSphere Application Server dynamic cache to recognize cacheable service operation of the operation.

WebSphere Application Server inspects the HTTP request to determine whether or not an incoming message can be cached based on the cache policies defined for an application. In this example, buyStock and stock-update are not cached, but stockquote-lookup is cached. In the `cachespec.xml` file for this Web application, the cache policies need defining for these services so that the dynamic cache can handle both SOAPAction and service operation.

WebSphere Application Server uses the operation and the message body in Web services cache IDs, each of which has a component associated with them. Therefore, each Web services `<cache-id>` rule contains only two components. The first is for the operation. Because you can perform the stockquote-lookup operation by either using a SOAPAction header or a service operation in the body, you must define two different `<cache-id>` elements, one for each method. The second component is of type ″body″, and defines how WebSphere Application Server should incorporate the message body into the cache ID. You can use a hash of the body, although it is legal to use the literal incoming message in the ID.

The incoming HTTP request is analyzed by WebSphere Application Server to determine which of the `<cache-id>` rules match. Then, the rules are applied to form cache or invalidation IDs.

The following is sample code of a `cachespec.xml` file defining SOAPAction and servicesOperation rules:

```
<cache>
<cache-entry>
 <class>webservice</class>
 <name>/soap/servlet/soaprouter</name>
 <sharing-policy>not-shared</sharing-policy>
 <cache-id>
```

```
  <component id="" type="SOAPAction">
   <value>urn:stockquote-lookup</value>
  </component>
  <component id="Hash" type="SOAPEnvelope"/>
   <timeout>3600</timeout>
   <priority>1<priority>
  </component>
 </cache-id>
 <cache-id>
  <component id="" type="serviceOperation">
   <value>urn:stockquote:getQuote</value>
  </component>
  <component id="Hash" type="SOAPEnvelope"/>
   <timeout>3600</timeout>
   <priority>1</priority>
  </component>
 </cache-id>
</cache-entry>
</cache>
```

Dynamic cache MBean statistics

The dynamic cache service provides an MBean interface to access cache statistics.

Access cache statistics with the MBean interface, using JACL

- Obtain the MBean identifier with the **queryNames** command, for example:

  ```
  $AdminControl queryNames type=DynaCache,*  // Returns a list of the available dynamic cache MBeans
  ```

  Select your dynamic cache MBean and run the following command:

  ```
  set mbean <dynamic_cache_mbean>
  ```

- Retrieve the names of the available cache statistics:

  ```
  $AdminControl invoke $mbean getCacheStatisticNames
  ```

- Retrieve the names of the available cache instances:

  ```
  $AdminControl invoke $mbean getCacheInstanceNames
  ```

- Retrieve all of the available cache statistics for the base cache instance:

  ```
  $AdminControl invoke $mbean getAllCacheStatistics
  ```

- Retrieve all of the available cache statistics for the named cache instance:

  ```
  $AdminControl invoke $mbean getAllCacheStatistics "services/cache/servletInstance_4"
  ```

- Retrieve cache statistics that are specified by the names array for the base cache instance:

  ```
  $AdminControl invoke $mbean getCacheStatistics
   {"DiskCacheSizeInMB ObjectsReadFromDisk4000K RemoteObjectMisses"}
  ```

  Note: This command should all be entered on one line. It is broken here for printing purposes.

- Retrieve cache statistics that are specified by the names array for the named cache instance:

  ```
  $AdminControl invoke $mbean getCacheStatistics
   {services/cache/servletInstance_4 "ExplicitInvalidationsLocal CacheHits"}
  ```

  Note: This command should all be entered on one line. It is broken here for printing purposes.

- Retrieve all the cache IDs in memory for the named cache instance that matches the specified regular expression:

  ```
  $AdminControl invoke $mbean getCacheIDsInMemory {services/cache/servletInstance_4 \S}
  ```

- Retrieve all cache IDs on disk for the named cache instance that matches the specified regular expression:

  ```
  $AdminControl invoke $mbean getCacheIDsOnDisk {services/cache/servletInstance_4 \S}
  ```

- Retrieves the CacheEntry, which holds metadata information for the cache ID:

  ```
  $AdminControl invoke $mbean getCacheEntry {services/cache/servletInstance_4 cache_id_1}
  ```

- Invalidates all cache entries that match the pattern-mapped cache IDs in the named cache instance and all cache entries dependent upon the matched entries in the instance:

  ```
  $AdminControl invoke $mbean invalidateCacheIDs {services/cache/servletInstance_4 cache_id_1 true}
  ```

Example: Configuring the dynamic cache service

This example puts all of the steps together for configuring the dynamic cache service with the `cachespec.xml` file, showing the use of the cache ID generation rules, dependency IDs, and invalidation rules.

Suppose that a servlet manages a simple news site. This servlet uses the query parameter ″action″ to determine if the request views (query parameter ″view″) news or updates (query parameter ″update″) news (used by the administrator). Another query parameter ″category″ selects the news category. Suppose that this site supports an optional customized layout that is stored in the user's session using the attribute name ″layout″. Here are example URL requests to this servlet:

http://*yourhost/yourwebapp*/newscontroller?action=view&category=sports (Returns a news page for the sports category)

http://*yourhost/yourwebapp*/newscontroller?action=view&category=money (Returns a news page for the money category)

http://*yourhost/yourwebapp*/newscontroller?action=update&category=fashion (Allows the administrator to update news in the fashion category)

Here are the steps for configuring the dynamic cache service for this example with the `cachespec.xml` file:

1. Define the <cache-entry> elements that are necessary to identify the servlet. In this case, the URI for the servlet is ″newscontroller″, so this is the cache-entry <name> element. Because this example caches a servlet or JavaServer Pages (JSP) file, the cache entry class is ″servlet″.

   ```
   <cache-entry>
   <name> /newscontroller </name>
   <class>servlet  </class>
    </cache-entry>
   ```

2. Define cache ID generation rules. This servlet caches only when action=view, so one component of the cache ID is the parameter ″action″ when the value equals ″view″. The news category is also an essential part of the cache ID. The optional session attribute for the user's layout is included in the cache ID. The cache entry is now:

   ```
   <cache-entry>
    <name> /newscontroller </name>
    <class>servlet  </class>
     <cache-id>
     <component id="action" type="parameter">
      <value>view</value>
      <required>true</required>
     </component>
     <component id="category" type="parameter">
      <required>true</required>
     </component>
     <component id="layout" type="session">
      <required>false</required>
     </component>
    </cache-id>
   </cache-entry>
   ```

3. Define dependency ID rules. For this servlet, a dependency ID is added for the category. Later, when the category is invalidated due to an update event, all views of that news category are invalidated. Following is an example of the cache entry after adding the dependency ID:

   ```
   <cache-entry>
    <name>newscontroller </name>
    <class>servlet  </class>
     <cache-id>
   ```

```
  <component id="action" type="parameter">
   <value>view</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
  </component>
  <component id="layout" type="session">
   <required>false</required>
  </component>
 </cache-id>
 <dependency-id>category
  <component id="category" type="parameter">
   <required>true</required>
  </component>
 </dependency-id>
</cache-entry>
```

4. Define invalidation rules. Because a category dependency ID is already defined, define an invalidation rule to invalidate the category when action=update. To incorporate the conditional logic, add ″ignore-value″ components into the invalidation rule. These components do not add to the output of the invalidation ID, but only determine whether or not the invalidation ID creates and runs. The final cache-entry now looks like the following:

```
<cache-entry>
 <name>newscontroller </name>
 <class>servlet  </class>
  <cache-id>
  <component id="action" type="parameter">
   <value>view</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
  </component>
  <component id="layout" type="session">
   <required>false</required>
  </component>
 </cache-id>
 <dependency-id>category
  <component id="category" type="parameter">
   <required>true</required>
  </component>
 </dependency-id>
 <invalidation>category
  <component id="action" type="parameter" ignore-value="true">
   <value>update</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
     </component>
 </invalidation>
</cache-entry>
```

Dynamic cache PMI counter definitions

The dynamic cache statistics interface is defined as WSDynamicCacheStats under the `com\ibm\websphere\pmi\stat` package.

Dynamic cache statistics are structured as follows in the Performance Monitoring Infrastructure (PMI) tree:

```
__Dynamic Caching+
  |
  |__<Servlet: instance_1>
    |__Templates+
      |__<template_1>
      |__<template_2>
```

```
    |__Disk+
       |__<Disk Offload Enabled>
  |
  |__<Object: instance_2>
     |__Object Cache+
        |__<Counters>
+ indicates logical group
```

StatDescriptor locates and accesses particular statistics in the PMI tree. For example:

1. StatDescriptor to represent statistics for cache servlet: `instance_1 templates group template_1: new StatDescriptor (new String[] {WSDynamicCacheStats.NAME, "Servlet: instance1", WSDynamicCacheStats.TEMPLATE_GROUP, "template_1"});`
2. StatDescriptor to represent statistics for cache servlet: `instance_1 disk group Disk Offload Enabled: new StatDescriptor (new String[] {WSDynamicCacheStats.NAME, "Servlet: instance_1", WSDynamicCacheStats.DISK_GROUP, WSDynamicCacheStats.DISK_OFFLOAD_ENABLED});`
3. StatDescriptor to represent statistics for cache object: `instance2 object cache group Counters: new StatDescriptor (new String[] {WSDynamicCacheStats.NAME, "Object: instance_2", WSDynamicCacheStats.OBJECT_GROUP, WSDynamicCacheStats.OBJECT_COUNTERS});`

Note: Cache instance names are prepended with cache type ("Servlet: " or "Object: ").

Counter definitions for Servlet Cache

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. ObjectsOnDisk	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED`	The current number of cache entries on disk.	6.1
WSDynamicCacheStats. HitsOnDisk	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED`	The number of requests for cacheable objects that are served from disk.	6.1
WSDynamicCacheStats. ExplicitInvalidations FromDisk	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED`	The number of explicit invalidations resulting in the removal of entries from disk.	6.1
WSDynamicCacheStats. TimeoutInvalidations FromDisk	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED`	The number of disk timeouts.	6.1

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats PendingRemoval FromDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of pending entries that are to be removed from disk.	6.1
WSDynamicCacheStats. DependencyIdsOnDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of dependency ID that are on disk.	6.1
WSDynamicCacheStats. DependencyIdsBuffered ForDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of dependency IDs that are buffered for the disk.	6.1
WSDynamicCacheStats. DependencyIds OffloadedToDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of dependency IDs that are offloaded to disk.	6.1
WSDynamicCacheStats. DependencyIdBased InvalidationsFromDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of dependency ID-based invalidations.	6.1
WSDynamicCacheStats. TemplatesOnDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of templates that are on disk.	6.1
WSDynamicCacheStats. TemplatesBuffered ForDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of templates that are buffered for the disk.	6.1
WSDynamicCacheStats. TemplatesOffloaded ToDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of templates that are offloaded to disk.	6.1

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. TemplateBased InvalidationsFromDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of template-based invalidations.	6.1
WSDynamicCacheStats. GarbageCollector InvalidationsFromDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of garbage collector invalidations resulting in the removal of entries from disk cache due to high threshold has been reached.	6.1
WSDynamicCacheStats. OverflowInvalidations FromDisk	WSDynamicCacheStats.NAME - "Servlet: cache_instance_1 " - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of invalidations resulting in the removal of entries from disk due to exceeding the disk cache size or disk cache size in GB limit.	6.1

Counter definitions for Object Cache

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. ObjectsOnDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of cache entries on disk.	6.1
WSDynamicCacheStats. HitsOnDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of requests for cacheable objects that are served from disk.	6.1
WSDynamicCacheStats. ExplicitInvalidations FromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of explicit invalidations resulting in the removal of entries from disk.	6.1
WSDynamicCacheStats. TimeoutInvalidations FromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of disk timeouts.	6.1

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats PendingRemoval FromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of pending entries that are to be removed from disk.	6.1
WSDynamicCacheStats. DependencyIdsOnDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of dependency ID that are on disk.	6.1
WSDynamicCacheStats. DependencyIds BufferedForDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of dependency IDs that are buffered for the disk.	6.1
WSDynamicCacheStats. DependencyIds OffloadedToDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of dependency IDs that are offloaded to disk.	6.1
WSDynamicCacheStats. DependencyIdBased InvalidationsFromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats.DISK_ OFFLOAD_ENABLED	The number of dependency ID-based invalidations.	6.1
WSDynamicCacheStats. TemplatesOnDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of templates that are on disk.	6.1
WSDynamicCacheStats. TemplatesBuffered ForDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP / -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The current number of templates that are buffered for the disk.	6.1
WSDynamicCacheStats. TemplatesOffloaded ToDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of templates that are offloaded to disk.	6.1

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. TemplateBasedInvalidations FromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of template-based invalidations.	6.1
WSDynamicCacheStats. GarbageCollector InvalidationsFromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of garbage collector invalidations resulting in the removal of entries from disk cache due to high threshold has been reached.	6.1
WSDynamicCacheStats. OverflowInvalidations FromDisk	WSDynamicCacheStats.NAME - "Object: cache_instance_2" - WSDynamicCacheStats. DISK_GROUP -" WSDynamicCacheStats. DISK_OFFLOAD_ENABLED	The number of invalidations resulting in the removal of entries from disk due to exceeding the disk cache size or disk cache size in GB limit.	6.1

Using the DistributedMap and DistributedObjectCache interfaces for the dynamic cache

By using the DistributedMap or DistributedObjectCache interfaces, Java Platform, Enterprise Edition (Java EE) applications and system components can cache and share Java objects by storing a reference to the object in the cache.

About this task

The DistributedMap and DistributedObjectCache interfaces are simple interfaces for the dynamic cache. Using these interfaces, Java EE applications and system components can cache and share Java objects by storing a reference to the object in the cache. The default dynamic cache instance is created if the dynamic cache service is enabled in the administrative console. This default instance is bound to the global Java Naming and Directory Interface (JNDI) namespace using the name services/cache/distributedmap.

Multiple instances of the DistributedMap and DistributedObjectCache interfaces on the same Java virtual machine (JVM) enable applications to separately configure cache instances as needed. Each instance of the DistributedMap interface has its own properties.

Note: For more information about the DistributedMap and DistributedObjectCache interfaces, see the API documentation for the com.ibm.websphere.cache package. See Additional Application Programming Interfaces (APIs) for more information.

Note: If you are using custom object keys, you must place your classes in a shared library. You can define the shared library at cell, node, or server level. Then, in each server create a class loader and associate it with the shared library that you defined. See the *Setting up the application serving environment* PDF for more information.

Place JAR files in a shared library when you deploy the application in a cluster with replication enabled. Simply turning on replication does not require a shared library; however, if you are using application-specific Java objects, such as cache key or cache value, those Java classes are

required to be in the shared library. If those values are not in a shared library, you will get ClassNotFound exceptions when the data replication service (DRS) attempts to deserialize those objects on the receiving side.

In a clustered environment, the content you place in cache might be shared with other servers in the cluster. The content might also be offloaded to disk. If you intend to have the cached objects shared or offloaded to disk, you must make these particular objects serializable. If the objects you place in cache are non-serializable, you must specify that the sharing policy for these objects is not shared. The DistributedMap interface DistributedMap interface contains information about how to specify the sharing policy for a cached object. Specifying a sharing policy other than not shared for non-serializable objects can result in poor system performance.

There are four methods for configuring and using cache instances:

- Configuring the default object cache (method one below)
- Creating and configuring the custom object cache (method three below)
- Creating and configuring the custom object cache by using the cacheinstances.properties file (method four below)
- Using the resource reference (method five below)
- **Method 1 - Configure default cache instances**. The default servlet cache instance (JNDI name: `services/cache/basecache`) is created when the server starts , if servlet caching is enabled. The default object cache instance (JNDI name: `services/cache/distributedmap`) is always created when the server starts.
  1. In the administrative console, select **Servers > Application servers >** *<server_name>* **Container services > Dynamic cache services**.
  2. Configure other cache settings. Refer to the Dynamic cache service settings article for more information.
  3. Click **Apply** or **OK**.
  4. Restart WebSphere Application Server.

  You can use the following code to look up the cache instances:

```
 InitialContext ic = new InitialContext();
DistributedMap dm1 = (DistributedMap)ic.lookup("services/cache/instance_one");

DistributedMap dm2 = (DistributedMap)ic.lookup("services/cache/instance_two");

// or

InitialContext ic = new InitialContext();
DistributedObjectCache dm1 = (DistributedObjectCache)ic.lookup("services/cache/instance_one");

DistributedObjectCache dm2 = (DistributedObjectCache)ic.lookup("services/cache/instance_two");
```

- **Method 2 - Configure servlet cache instances**. A servlet cache instance is a location, in addition to the default servlet cache, where dynamic cache can store, distribute, and share data. By using servlet cache instances, your applications have greater flexibility and better tuning of the cache resources. The Java™ Naming and Directory Interface (JNDI) name that is specified for the cache instance is mapped to the name attribute in the `<cache instance>` tag in the `cachespec.xml` configuration file.
  1. In the administrative console, click **Resources > Cache instances > Servlet cache instances**.
  2. Enter the scope, as follows:
     - Specify CELL SCOPE to view and configure cache instances that are available to all servers within the cell.
     - Specify NODE SCOPE to view and configure cache instances that are available to all servers with the particular node.

– Specify SERVER SCOPE to view and configure cache instances that are available only on the specific server.

3. Enter the required display name for the resource in the name field.
4. Enter the JNDI name for the resource. Specify this name in the attribute field in the `<cache-instance>` tag in the `cachespec.xml` configuration file. This tag finds the particular cache instance in which to store cache entries.
5. Configure other cache settings. Refer the Dynamic cache service settings article for more information.
6. Click **Apply** or **OK**.
7. **Optional:**If you want to set up additional custom properties for this instance, click **Resources > Cache instances > Servlet cache instances** *<servlet_cache_instance_name>* **> Custom properties > New**.
8. **Optional:** Enter the name of the custom property in the Name field. Refer to the Dynamic cache custom properties article for more information.

   Note: Use the custom property with scope indicated **Per cache instnace** only. For example, enter `createCacheAtServerStartup` in the Name field.
9. Enter a valid value for the property in the Value field.
10. Save the property and restart WebSphere Application Server.

-
- **Method 3 - Configure object cache instances**. An object cache instance is a location, in addition to the default object cache, where dynamic cache can store, distribute, and share data for Java™ Platform, Enterprise Edition (Java EE) applications. Use cache instances to give applications better flexibility and tuning of the cache resources. Use the DistributedObjectCache programming interface to access the cache instances. For more information about the DistributedObjectCache application programming interface, see the API documentation.

  Note: Method three is an extension to method one or method two, listed above. First use either method one or method two.

  Create and configure the object cache instance, as follows:

  1. In the administrative console, click **Resources > Cache instances > Object cache intances**.
  2. Enter the scope:
     – Specify CELL SCOPE to view and configure cache instances that are available to all servers within the cell.
     – Specfy NODE SCOPE to view and configure cache instances that area available to all servers with the particular node.
     – Specify SERVER SCOPE to view and configure cache instances that are available only on the specific server.
  3. Enter the required display name for the resource in the name field.
  4. Enter the JNDI name for the resource. Use this name when looking up a reference to this cache instance. The results return a DistributedMap object.
  5. Configure other cache settings. See the Dynamic cache service setting article for more information.
  6. Click **Apply** or **OK**.
  7. **Optional:** If you want to set up additional custom properties for this instance, click **Resources > Cache instances > Object cache instances** *<servlet_cache_instance_name>* **Custom properties > New**.
  8. **Optional:** Enter the name of the custom property in the Name field.

     Note: Use the custom property with scope indicated **Per cache instance** only. For example, enter `createCacheAtServerStartup` in the Name field.

9. Enter a valid value for the property in the Value field.
10. Save the property and restart WebSphere Application Server.

If you defined two object cache instances in the administrative console with JNDI names of **services/cache/instance_one** and **services/cache/instance_two**, you can use the following code to look up the cache instances:

```
 InitialContext ic = new InitialContext();
DistributedMap dm1 = (DistributedMap)ic.lookup("services/cache/instance_one");

DistributedMap dm2 = (DistributedMap)ic.lookup("services/cache/instance_two");

// or

InitialContext ic = new InitialContext();
DistributedObjectCache dm1 = (DistributedObjectCache)ic.lookup("services/cache/instance_one");

DistributedObjectCache dm2 = (DistributedObjectCache)ic.lookup("services/cache/instance_two");
```

- **Method 4 - Configure cache instances using the cacheinstances.properties file**. You can create cache instances using the cacheinstances.properties file and package the file in your Enterprise Archive (EAR) file. Use the table information in the cacheinstances.properties file article as a reference of the names, values, and explanations.

  The first line defines the cache instance name. The subsequent lines define custom properties. The formats are as follows:

  ```
  cache.instance.x=InstanceName
  cache.instance.x.customPropertyName=customPropertyValue
  ```

  where:

  – *x* is the instance name number, which starts with `0`.
  – *customPropertyName* is the custom property name. Refer to the Dynamic cache custom properties article for more information.

    Note: Use the custom property with scope indicated per cache instance only.
  – *customPropertyValue* is the possible custom property value.

  Examples

  The following is an example of how you can create additional cache instances using the cacheinstances.properties. file:

  ```
  cache.instance.0=/services/cache/instance_one
  cache.instance.0.cacheSize=1000
  cache.instance.0.enableDiskOffload=true
  cache.instance.0.diskOffloadLocation=${app_server_root}/diskOffload
  cache.instance.0.flushToDiskOnStop=true
  cache.instance.0.useListenerContext=true
  cache.instance.0.enableCacheReplication=false
  cache.instance.0.disableDependencyId=false
  cache.instance.0.htodCleanupFrequency=60
  cache.instance.1=/services/cache/instance_two
  cache.instance.1.cacheSize=1500
  cache.instance.1.enableDiskOffload=false
  cache.instance.1.flushToDiskOnStop=false
  cache.instance.1.useListenerContext=false
  cache.instance.1.enableCacheReplication=true
  cache.instance.1.replicationDomain=DynaCacheCluster
  cache.instance.1.disableDependencyId=true
  ```

  The preceding example creates two cache instances named instance_one and instance_two. Cache instance_one has a cache entry size of `1,000` and instance_two has a cache entry size of `1,500`. Disk offload is enabled in instance_one and disabled in instance_two. Use listener context is enabled in instance_one and disabled in instance_two. Flush to disk on stop is enabled in instance_one and

disabled in instance_two. Cache replication is enabled in instance_two and disabled in instance_one. The name of the data replication domain for instance_two is DynaCacheCluster. Dependency ID support is disabled in instance_two.

Place the cacheinstances.properties file in either your application server or application class path. For example, you can use your application WAR file, WEB-INF\classes directory or server_root\classes directory. The first entry in the properties file (cache.instance.0) specifies the JNDI name for the cache instance in the global namespace.

You can use the following code to look up the cache instance:

```
  InitialContext ic = new InitialContext();
     DistributedMap dm1 =
(DistributedMap)ic.lookup("services/cache/instance_one");
     DistributedMap dm2 =
(DistributedMap)ic.lookup("services/cache/instance_two");

// or

InitialContext ic = new InitialContext();
DistributedObjectCache dm1 =  (DistributedObjectCache)ic.lookup("services/cache/instance_one");

DistributedObjectCache dm2 = (DistributedObjectCache)ic.lookup("services/cache/instance_two");
```

- **Method 5: Resource reference**.

  Note: This method is an extension to method three and method four, listed above. First use either method three or method four.

  Define a resource-ref in your module deployment descriptor (web.xml and ibm-web-bnd.xmi files) and look up the cache using the `java:comp` namespace.

  Resource-ref example:

  File: web.xml

```
<resource-ref id="ResourceRef_1">
  <res-ref-name>dmap/LayoutCache</res-ref-name>
  <res-type>com.ibm.websphere.cache.DistributedMap</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-ref id="ResourceRef_2">
  <res-ref-name>dmap/UserCache</res-ref-name>
  <res-type>com.ibm.websphere.cache.DistributedMap</res-type>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

  File: ibm-web-bnd.xmi

```
<?xml version="1.0" encoding="UTF-8"?>
<webappbnd:WebAppBinding xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:webappbnd="webappbnd.xmi"
xmlns:webapplication="webapplication.xmi" xmlns:commonbnd="commonbnd.xmi"
xmlns:common="common.xmi"
xmi:id="WebApp_ID_Bnd" virtualHostName="default_host">
  <webapp href="WEB-INF/web.xml#WebApp_ID"/>
  <resRefBindings xmi:id="ResourceRefBinding_1"
jndiName="services/cache/instance_one">
  <bindingResourceRef href="WEB-INF/web.xml#ResourceRef_1"/>
  </resRefBindings>
  <resRefBindings xmi:id="ResourceRefBinding_2"
jndiName="services/cache/instance_two">
  <bindingResourceRef href="WEB-INF/web.xml#ResourceRef_2"/>
  </resRefBindings>
  </webappbnd:WebAppBinding>
```

  The following example shows how to look up the resource-ref:

```
InitialContext ic = new InitialContext();
  DistributedMap dm1a =(DistributedMap)ic.lookup("java:comp/env/dmap/LayoutCache");
  DistributedMap dm2a =(DistributedMap)ic.lookup("java:comp/env/dmap/UserCache");
```

```
// or
  DistributedObjectCache dm1a =(DistributedObjectCache)ic.lookup("java:comp/env/dmap/LayoutCache");
  DistributedObjectCache dm2a =(DistributedObjectCache)ic.lookup("java:comp/env/dmap/UserCache");
```

The previous resource-ref example maps `java:comp/env/dmap/LayoutCache` to `/services/cache/instance_one` and `java:comp/env/dmap/UserCache` to `/services/cache/instance_two`. In the examples, DistributedMap dm1 and dm1a are the same object. DistributedMap dm2 and dm2a are the same object.

- **Method 6: Java virtual machine cache settings**. You can set the custom properties globally to affect all cache instances. This overwrites the settings in method 1, method 2 and method 3, but not method 4 (cacheinstances.properties). Configure the cache instance globally, as follows:
  1. In the administrative console, click **Servers > Application servers > server_name > Java and process management > Process definition > Java virtual machine > Custom properties > New**.
  2. Enter the name of the custom property in the Name field. Refer to the Dynamic cache custom properties article for more information. After you find the custom property name, add the `com.ibm.ws.cache.CacheConfig` prefix to the front of custom property name. For example, if the custom property name is `createCacheAtServerStartup`, enter `com.ibm.ws.cache.CacheConfig.createCacheAtServerStartup` in the Name field.
  3. Enter a valid value for the property in the Value field.
  4. Save the property and restart WebSphere Application Server.

Object cache instance settings

An object cache instance is a location, in addition to the default shared dynamic cache, where any Java 2 Platform, Enterprise Edition (J2EE) application can store, distribute, and share data. This gives applications greater flexibility and better tuning of the cache resources. Use the DistributedMap programming interface to access this cache instance. See the API documentation for more information.

To view this administrative console page, click **Resources > Cache instances > Object cache instances > *cache_instance_name***.

Name:

Specifies the required display name for the resource.

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name for the resource. Use this name when looking up a reference to this cache instance. The results return a `DistributedMap` object.

Description:

Specifies a description for the resource. This field is optional.

Category:

Specifies a category string to classify or group the resource. This field is optional.

Cache size:

Specifies a positive integer for the maximum number of entries the cache holds. The cache size is usually in the thousands.

Default	2000
Range	100 - 200,000

Default priority:

Specifies the default priority for servlets that can be cached. This value determines how long an entry stays in a full cache.

The recommended value is one. The range is one through 255.

Enable disk offload:

Specifies if disk offloading is enabled.

If you have disk offload disabled, when a new entry is created while the cache is full, the priorities are configured for each entry and the least recently used algorithm are used to remove the entry from the cache in memory. If you enable disk offload, the entry that would be removed from the cache is copied to the local file system. The location of the file is specified by the disk offload location.

Default	false

Offload location:

Specifies the directory that is used for disk offload.

If disk offload location is not specified, the default location, `${WAS_TEMP_DIR}/`*`node`*`/`*`server name`*`/_dynacache/`*`cache JNDI name`* will be used. If disk offload location is specified, the node, server name, and cache instance name are appended. For example, `${USER_INSTALL_ROOT}/diskoffload` generates the location as `${USER_INSTALL_ROOT}/diskoffload/`*`node`*`/`*`server name`*`/`*`cache JNDI name`*. This value is ignored if disk offload is not enabled.

The default value of the ${WAS_TEMP_DIR} property is `${USER_INSTALL_ROOT}/temp`. If you change the value of the ${WAS_TEMP_DIR} property after starting WebSphere Application Server, but do not move the disk cache contents to the new location:

- The Application Server creates a new disk cache file at the new disk offload location.
- If the Flush to disk setting is enabled, all the disk cache content at the old location is lost when you restart the Application Server

Flush to disk:

Specifies if in-memory cached objects are saved to disk when the server is stopped. This value is ignored if `Enable Disk Offload` is not selected.

Default	false

Limit disk cache size in GB:

Specifies a value for the maximum disk cache size in GB. When you select this option, you can specify a positive integer value. Leaving this option blank indicates an unlimited size. This setting applies only if enable disk offload is specified for the cache.

Value	0 to MAXINT. A value of 0 indicates unlimited size.

Limit disk cache size in entries:

Specifies a value for the maximum disk cache size in number of entries. When you select this option, you can specify a positive integer value. Leaving this option blank indicates an unlimited size. This setting applies only if enable disk offload is specified for the cache.

Value	0 to MAXINT. A value of 0 indicates unlimited size.

Limit disk cache entry size:

Specifies a value for the maximum size of an individual cache entry in MB. Any cache entry larger than this, when evicted from memory, will not be offloaded to disk. When you select this option, you can specify a positive integer value. Leaving this option blank indicates an unlimited size. This setting applies only if enable disk offload is specified for the cache.

Value	0 to MAXINT. A value of 0 indicates unlimited size.

Performance settings:

Specifies the level of performance that is required by the disk cache. This setting applies only if **enableDiskOffload** is specified for the cache. Performance levels determine how memory resources should be used on background activity such as cache cleanup, expiration, garbage collection, and so on. This setting applies only if enable disk offload is specified for the cache.

High performance and high memory usage	Indicates that all metadata will be kept in memory.
Balanced performance and balanced memory usage	Indicates some metadata will be kept in memory. This is the default performance setting and will provide an optimal balance of performance and memory usage for most users.
Low performance and low memory usage	Indicates that limited metadata will be kept in memory.
Custom performance	Indicates that the administrator will explicitly configure the memory settings that will be used to support the above background activity. The administrator sets these values using the **DiskCacheCustomPerformanceSettings** object.

Disk cache cleanup frequency:

Specifies a value for the disk cache cleanup frequency, in minutes. If this value is set to 0, the cleanup runs only at midnight. This setting applies only when the Disk Offload Performance Level is low, balanced, or custom. The high performance level does not require disk cleanup, and this value is ignored.

Value	0 to 1440

Maximum buffer for cache identifiers per metaentry:

Specifies a value for the maximum number of cache identifiers that are stored for an individual dependency ID or template in the disk cache metadata in memory. If this limit is exceeded the information is offloaded to the disk. This setting applies only when the disk offload performance level is custom.

Value	100 to MAXINT

Maximum buffer for dependency identifiers:

Specifies a value for the maximum number of dependency identifier buckets in the disk cache metadata in memory. If this limit is exceeded the information is offloaded to the disk. This setting applies only when the disk cache performance level is custom.

Value	100 to MAXINT

Maximum buffer for templates:

Specifies a value for the maximum number of template buckets that are in the disk cache metadata in memory. If this limit is exceeded the information is offloaded to the disk. This setting applies only when the disk cache performance level is custom.

Value	10 to MAXINT

Eviction policy algorithm:

Specifies the eviction algorithm that the disk cache will use to evict entries once the high threshold is reached. This setting applies only if enable disk offload is specified for the cache.

None	No eviction policy, so the disk cache can grow until it reaches its limit at which time the dynamic cache service stops writing to disk
Random	When the disk size reaches a high threshold limit, the disk cache garbage collector wakes up and randomly picks entries on the disk and evicts them until the size reaches a low threshold limit.
Size	When the disk size reaches a high threshold limit, the disk cache garbge collector wakes up and picks the largest entries on the disk and evicts them until the disk size reaches a low threshold limit.

High threshold:

Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the disk cache size in GB or entries. The disk cache garbage collector is awaken when the disk size exceeds high threshold limit. The lower value limits disk cache size in GB and disk cache size in entries. This setting does not apply when the disk cache eviction policy is set to none.

Values	1 to 100

Low threshold:

Specifies when the eviction policy ends. The threshold is expressed in terms of the percentage of the disk cache size in GB or entries. The lower value limits disk cache size in GB and disk cache size in entries. The disk cache garbage collector, when awaken, evicts entries until the disk size reaches the low threshold limit. This setting does not apply when the disk cache eviction policy is set to none.

Values	1 to 100

Use listener context:

Set this value to true to have invalidation events sent to registered invalidation listeners using the Java 2 Platform, Enterprise Edition (J2EE) context of the listener. If you want to use listener J2EE context for callback, set this value to **true**. If you want to use the caller thread context for callback, set this to **false**.

Dependency ID support:

Specifies that the dynamic cache service, supports cache entry dependency IDs. Disable this option if you do not need to use dependency IDs. Dependency IDs specify additional cache group identifiers that associate multiple cache entries to the same group identifier in your cache policy.

This option might not be available for cache instances that were created with a previous version of WebSphere Application Server.

Default	true

Enable cache replication:

Use cache replication to enable sharing of cache IDs, cache entries, and cache invalidations with other servers in the same replication domain.

This option might be unavailable for cache instances created with a previous version of WebSphere Application Server.

Full group replication domain:

Specifies a replication domain from which your data is replicated.

Specifies a replication domain from which your data is replicated. Choose from any replication domains that have been defined. If there are no replication domains listed, you must create one during cluster creation or manually in the administrative console by clicking **Environment > Internal replication domains > New**. The replication domain you choose to use with the dynamic cache service must be using a Full group replica. Do not share replication domains between replication consumers. Dynamic cache should use a different replication domain from session manager or stateful session beans.

Replication type:

Specifies the global sharing policy for this cache instance.

The following settings are available:

- **Both push and pull** sends the cache ID of newly updated content to other servers in the replication domain. Then, if one of the other servers requests the content, and that server has the ID of the cache entry for the previously updated content, it will retrieve the content from the publishing server. If a request is made for an ID which has not been previously published, the server assumes it does not exist in the cluster and creates a new entry.
- **Pull only** shares cache entries for this object between application servers on demand. If an application server gets a cache miss for this object, it queries the cooperating application servers to see if they have the object. If no application server has a cached copy of the object, the original application server runs the request and generates the object. These entries cannot store non-serializable data. This mode of sharing is not recommended.
- **Push only** sends the cache ID and cache content of new content to all other servers in the replication domain.
- The sharing policy of **Not Shared** results in the cache ID and cache content not being shared with other servers in the replication domain.

The default setting for a an environment without clustering is **Not Shared**. When enabling replication, the default value is **Not Shared**.

Push frequency:

Specifies the time, in seconds, to wait before pushing new or modified cache entries to other servers.

A value of `0` (zero) sends the cache entries immediately. Setting this property to a value greater than `0` (zero) results in a ″batch″ push of all cache entries that are created or modified during the time period. The default is `1` (one).

Object cache instance collection

Use this page to configure and manage object cache instances, which in addition to the default shared dynamic cache, can store, distribute, and share data for Java 2 Platform, Enterprise Edition (J2EE) applications. Use cache instances to give applications better flexibility and tuning of the cache resources.

To view this administrative console page, click **Resources > Cache instances > Object cache instances**.

Use the DistributedObjectCache programming interface to access the cache instances. For more information about the DistributedObjectCache application programming interface, see the API documentation.

Scope: Specify CELL SCOPE to view and configure cache instances that are available to all servers within the cell. Specify NODE SCOPE to view and configure cache instances that are available to all servers with the particular node. Specify SERVER SCOPE to view and configure cache instances that are available only on the specific server.

Name:

Specifies the required display name for the resource.

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name for the resource. Use this name when looking up a reference to this cache instance. The results return a `DistributedMap` object.

Cache size:

Specifies a positive integer for the maximum number of entries the cache holds. The cache size is usually in the thousands. The default is 2000.

The minimum value is 100, with no set maximum value.

cacheinstances.properties file

Use the information in this document as a reference of the names, values, and explanations that you can use in the cacheinstances.properties file.

The following list provides the property names, associated values, and explanations for the cacheinstance.properties file.

Property name - *x* is the instance number, which starts with `0`	**Version**	**Scope**	**Possible value**	**Description**

Property name - *x* is the instance number, which starts with `0`	Version	Scope	Possible value	Description
Cache core properties				
`cache.instance.x`	5.1.x and later	Per cache instance	any string (no default set)	Specifies cache instance name or JNDI name.
`cache.instance.x. cacheSize`	5.1.x and later	Per cache instance	> 0 (default=2000)	Specifies the maximum number of entries that are held in memory cache.
`cache.instance.x. disableDependencyId`	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies that the dynamic cache service supports cache entry dependency IDs. Disable this option if you do not need to use dependency IDs. Dependency IDs specify additional cache group identifiers that associate multiple cache entries to the same group identifier in your cache policy.
`cache.instance.x. disableTemplates Support`	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether template support feature is enabled.
`cache.instance.x. useListenerContext`	5.1.x and later	Per cache instance	True or false (default=false)	Set this value to true to have invalidation events sent to registered invalidation listeners, using the Java Platform, Enterprise Edition (Java EE) context of the listener. If you want to use listener Java EE context for callback, set this value to true. If you want to use the caller thread context for callback, set this value to false.
`cache.instance.x. enableNioSupport`	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether DistributedMap or DistributedNioMap is used.
`cache.instance.x. memoryCacheSizeInMB`	7.0	Per cache instance	> 0 (default: -1 limit does not exist)	Specifies a value for the maximum memory cache size in megabytes (MB)

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x.memoryCacheHighThreshold	7.0	Per cache instance	> 0 % (default=95)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the memory cache size in MB. The higher value is used when limit memory cache size in MB is specified.
cache.instance.x.memoryCacheLowThreshold	7.0	Per cache instance	> 0 % (default=80)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the memory cache size in MB. The lower value is used when limit memory cache size in MB is specified.
cache.instance.x.createCacheAtServerStartup	7.0	Per cache instance	True or false (default=false)	Specifies whether the configured cache instance is created during the server startup. This is useful when cache replication feature is used. However, the time for server startup will take long.
Cache servlet/JavaServer Pages (JSP) caching properties				

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. cascadeCachespec Properties	6.0.2.19, 6.1.0.9 and later	Per cache instance	True or false (default=false)	A configurable change in the behavior of the cache so that the child pages and fragments inherit the cache specification properties of their parent pages and fragments. If the request for a fragment does not match a defined cache policy, the fragment will inherit the save-attributes and the store-cookies properties from its parent fragment. Enable this cascade of save-attributes and store-cookies properties by setting the value to true.
cache.instance.x. disableStoreCookies	6.0.2.9, 6.1.x and later	Per cache instance	″none″, ″ALL″,″All″, cache instance name, comma delineated list of cookie names, (default=″none″)	Specifies whether disable store cookies is NONE or ALL. Stores cookies as part of the response by default unless configured otherwise on a per request basis in cachespec.xml file. There is a risk of sharing cookies between users, which violates security.
cache.instance.x. enableServlet Support	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether the cache instance is servlet cache or object cache.
Cache disk offload properties				
cache.instance.x. enableDiskOffload	5.1.x and later	Per cache instance	True or false (default=false)	Specifies whether disk offload is enabled.
cache.instance.x. diskOffload Location	5.1.x and later	Per cache instance	String – For example: . $(app_server_root) /diskOffload	Specifies the location on the disk to save cache entries when disk offload is enabled.

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. diskCacheSize	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	>= 0 (0=limit does not exist)	Specifies a value for the maximum disk cache size in number of entries.
cache.instance.x. diskCacheSizeInGB	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	0 or > 2 in GB (0=limit does not exist)	Specifies a value for the maximum disk cache size in gigabytes (GB).
cache.instance.x. diskCacheEntrySize InMB	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	>= 0 in MB (0=limit does not exist)	Specifies a value for the maximum size of an individual cache entry in megabytes (MB). Any cache entry that is larger than this, when evicted from memory, will not be offloaded to disk.
cache.instance.x. flushToDiskOnStop	5.1.x and later	Per cache instance	True or false (default = false)	Specifies if in-memory cached objects are saved to disk when the server stops.
cache.instance.x. diskCache PerformanceLevel	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	0=low 1=balance 2=custom 3=high (default=1)	Specifies the performance level to tune the performance of the disk cache.
cache.instance.x. htodCleanup Frequency	5.1.1.2 and later	Per cache instance	0 <= x <= 1440 in minutes (0=cleanup at midnight)	Specifies a value for the disk cache cleanup frequency, in minutes. If this value is set to 0, the cleanup runs only at midnight. This setting applies only when the Disk Offload Performance Level is low, balanced, or custom. The high performance level does not require disk cleanup, and this value is ignored.

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x.htodDelayOffloadDepIdBuckets	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 (default=1000)	Specifies a value for the maximum number of dependency identifier buckets in the disk cache metadata in memory. If this limit is exceeded, the information is offloaded to the disk. This setting applies only when the disk cache performance level is custom.
cache.instance.x.htodDelayOffloadTemplateBuckets	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 (default=100)	Specifies a value for the maximum number of template buckets that are in the disk cache metadata in memory. If this limit is exceeded, the information is offloaded to the disk. This setting applies only when the disk cache performance level is custom.
cache.instance.xhtodDelayOffloadEntriesLimit	5.1.1.2 and later	Per cache instance	> 0 (default=1000)	Specifies a value for the maximum number of cache identifiers that are stored for an individual dependency ID or template in the disk cache metadata in memory. If this limit is exceeded, the information is offloaded to the disk. This setting applies only when the disk offload performance level is custom.
cache.instance.x.diskCacheEvictionPolicy	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	0=disable 1=random 2:size (default=0)	Specifies the eviction algorithm that the disk cache will use to evict entries once the high threshold is reached.

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. diskCacheHigh Threshold	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 % (default=80)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the disk cache size in GB or entries. The high value is used when limit disk cache size in GB and limit disk cache size in entries are specified.
cache.instance.x. diskCacheLow Threshold	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 % (default=70)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the disk cache size in GB or entries. The lower value is used when limit disk cache size in GB and limit disk cache size in entries are specified.
Cache replication properties				
cache.instance.x. enableCache Replication	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether cache replication is enabled. Use cache replication to have cache entries copied to multiple application servers configured in the same replication domain.
cache.instance.x. replicationType	5.1.x and later	Per cache instance	1 (Not shared, 2 (Push), 4 (Push and pull)	Specifies the global sharing policy for this application server.
cache.instance.x. replicationDomain	6.0.2.x and later	Per cache instance	String – For example: DynamicCacheDomain	Specifies a replication domain from which your data is replicated.

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. useServer ClassLoader	5.1.1.9, 6.0.2.9, 6.1.x and later	Per cache instance	True or false (default=false)	Specifies whether using server class loader is enabled. Setting this value to true, deserializes the InvalidationEvent using system classloader first and then using application classloader, if that fails. This improves performance.
cache.instance.x. cacheEntryWindow	5.1.1.13, 6.0.2.17, 6.1.0.7 and later	Per cache instance	> 0 (default=50)	Specifies a limit on the total number of cache entries that are sent by the data replication service (DRS) in terms of number of entries.
cache.instance.x. cachePercentage Window	5.1.1.13, 6.0.2.17, 6.1.0.7 and later	Per cache instance	> 0 % (default=2)	Specifies a limit on the number of cache entries that are sent by DRS in terms of the percentage of total cache in memory.
cache.instance.x. cacheInvalidate EntryWindow	5.1.1.14, 6.0.2.19, 6.1.0.7 and later	Per cache instance	> 0 (default=50)	Specifies a limit on the total number of invalidation events that are sent by DRS in terms of number of entries.
cache.instance.x. cacheInvalidate PercentWindow	5.1.1.14, 6.0.2.19, 6.1.0.7 and later	Per cache instance	> 0 % (default=2)	Specifies a limit on the number of invalidation events that are sent by DRS in terms of the percentage of total cache in memory.
cache.instance.x. filterTimeOut Invalidation	6.0.2.13, 6.1.x and later	Per cache instance	True or false (default=false)	Specifies whether sending invalidations that are based on timeout eviction is enabled.
cache.instance.x. filterLRU Invalidation	6.0.2.13, 6.1.x and later	Per cache instance	True or false (default=false)	Specifies whether sending invalidations that are based on LRU eviction is enabled.

Property name - *x* is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. ignoreValueIn InvalidationEvent	5.1.1.13, 6.0.2.17, 6.1.x or later	Per cache instance	True or false (default=false)	Specifies whether the cache value of Invalidation event is ignored. If it is true, the cache value of Invalidation event is set to NULL when the code is returned to the caller.

Invalidation listeners

Invalidation listener mechanism uses Java events for alerting applications when contents are removed from the cache.

Applications implement the InvalidationListener interface (defined in the com.ibm.websphere.cache package) and register it to the cache using the DistributedMap interface. Listeners receive InvalidationEvents (defined in the com.ibm.websphere.cache package) when entries from the cache are removed, due to an explicit user invalidation, timeout, least recently used (LRU) eviction, cache clear, or disk timeout. Applications can immediately recalculate the invalidated data and prime the cache before the next user request.

Enable listener support in DistributedMap before registering listeners. DistributedMap can also be configured to use the invalidation listener Java Platform, Enterprise Edition (Java EE) context from registration time during callbacks. Setting the value of the custom property useListenerContext to true enables the invalidation listener Java EE context for callbacks. See Cache instance settings for more information.

The following example shows how to set up an invalidation listener:

```
dmap.enableListener(true);  // Enable cache invalidation listener.
InvalidationListener listener = new MyListenerImpl();  //Create invalidation listener object.
dmap.addInvalidationListener(listener);  //Add invalidation listener.
      :
      :
      :
dmap.removeInvalidationListener(listener);  //Remove the invalidation listener.
//This increases performance.
dmap.enableListener(false);  // Disable cache invalidation listener.
//This increases performance.
```

For more information about invalidation listeners, see Additional Application Programming Interfaces (APIs) for the com.ibm.websphere.cache package.

Dynamic query

Using EJB query

The Enterprise JavaBeans (EJB) query language is used to specify a query over container-managed entity beans. The language is similar to structured query language (SQL). An EJB query is independent of the bean's mapping to a persistent store.

About this task

An EJB query can be used in three situations:
- To define a finder method of an EJB entity bean.
- To define a select method of an EJB entity bean.
- To dynamically specify a query using the executeQuery method dynamic API.

Finder and select queries are specified in the bean's deployment descriptor using the `<ejb-ql>` tag; they are compiled into SQL during deployment. Dynamic queries are included within the application code itself.

The product's EJB query language is compliant with the EJB QL defined in Sun's EJB 2.1 and EJB 3.0 specifications and has additional capabilities as listed in the topic Comparison of EJB specification and WebSphere Query Language.

- Before using EJB query, familiarize yourself with query language concepts, starting with the topic, EJB Query Language.
- Define an EJB query in one of the following ways:
  - **Rational Application Developer.** When defining an entity bean, specify the `<ejb-ql>` tag for the finder or select method. For more information about using Rational Application Developer see the assembly tool information center at http://wilson.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/index.jsp?topic=/com.ibm.jee5.doc/topics/cejb3.html
  - **Dynamic query service.** Add the executeQuery method to your application.

Example

See the topic Example: EJB queries.

EJB query language

EJB query language enables you to write queries based on entity beans without knowing the underlying relational schema.

An EJB query is a string that contains the following elements:
- a SELECT clause that specifies the enterprise beans or values to return;
- a FROM clause that names the bean collections;
- an optional WHERE clause that contains search predicates over the collections;
- an optional GROUP BY and HAVING clause (see Aggregation functions);
- an optional ORDER BY clause that specifies the ordering of the result collection.

Collections of entity beans are identified in EJB queries through the use of their abstract schema name in the query FROM clause.

The elements of EJB query language are discussed in more detail in the following related topics.

Example: Queries with EJB:

Here is an example Enterprise JavaBeans (EJB) schema, followed by a set of example queries.

Table 46. DeptBean schema

Entity bean name (EJB name)	DeptEJB (not used in query)
Abstract schema name	DeptBean
Implementation class	com.acme.hr.deptBean (not used in query)
Persistent attributes (cmp fields)	• deptno - Integer (key) • name - String • budget - BigDecimal
Relationships	• emps - 1:Many with EmpEJB • mgr - Many:1 with EmpEJB

Table 47. EmpBean schema

Entity bean name (EJB name)	EmpEJB (not used in query)
Abstract schema name	EmpBean
Implementation class	com.acme.hr.empBean (not used in query)
Persistent attributes (cmp fields)	• empid - Integer (key) • name - String • salary - BigDecimal • bonus - BigDecimal • hireDate - java.sql.Date • birthDate - java.util.Calendar • address - com.acme.hr.Address
Relationships	• dept - Many:1 with DeptEJB • manages - 1:Many with DeptEJB

Address is a serializable object used as cmp field in EmpBean. The definition of address is as follows:

```
   public class com.acme.hr.Address  extends Object implements Serializable {
public String street;
public String state;
public String city;
public Integer zip;
     public double distance (String start_location) { ... } ;
     public  String format ( ) { ... } ;
}
```

The following query returns all departments:

```
SELECT OBJECT(d) FROM DeptBean d
```

The following query returns departments whose name begins with the letters ″Web″. Sort the result by name:

```
SELECT OBJECT(d) FROM DeptBean d WHERE  d.name LIKE  'Web%' ORDER BY d.name
```

The keywords SELECT and FROM are shown in uppercase in the examples but are case insensitive. If a name used in a query is a reserved word, the name must be enclosed in double quotes to be used in the query. You can find a list of reserved words in “EJB query: Reserved words” on page 1603. Identifiers enclosed in double quotes are case sensitive. This example shows how to use a cmp field that is a reserved word:

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d."select" > 5
```

The following query returns all employees who are managed by Bob. This example shows how to navigate relationships using a path expression:

```
SELECT OBJECT (e) FROM EmpBean e WHERE e.dept.mgr.name='Bob'
```

A query can contain a parameter which refers to the corresponding value of the finder or select method. Query parameters are numbered starting with 1:

```
SELECT OBJECT (e) FROM EmpBean e WHERE e.dept.mgr.name= ?1
```

This query shows navigation of a multivalued relationship and returns all departments that have an employee that earns at least 50000 but not more than 90000:

```
SELECT OBJECT(d) FROM DeptBean d,  IN (d.emps) AS e
WHERE e.salary BETWEEN 50000 and 90000
```

There is a join operation implied in this query between each department object and its related collection of employees. If a department has no employees, the department does not appear in the result. If a department has more than one employee that earns more than 50000, that department appears multiple times in the result.

The following query eliminates the duplicate departments:

```
SELECT DISTINCT OBJECT(d) from DeptBean d,  IN (d.emps) AS e  WHERE e.salary > 50000
```

Find employees whose bonus is more than 40% of their salary:

```
SELECT OBJECT(e) FROM EmpBean e where e.bonus > 0.40 * e.salary
```

Find departments where the sum of salary and bonus of employees in the department exceeds the department budget:

```
SELECT OBJECT(d) FROM DeptBean d where d.budget <
( SELECT SUM(e.salary+e.bonus) FROM IN(d.emps) AS e )
```

A query can contain DB2 style date-time arithmetic expressions if you use java.sql.* datatypes as CMP fields and your datastore is DB2. Find all employees who have worked at least 20 years as of January 1st, 2000:

```
SELECT OBJECT(e) FROM EmpBean e where year(  '2000-01-01' - e.hireDate ) >= 20
```

If the datastore is not DB2 or if you prefer to use java.util.Calendar as the CMP field, then you can use the java millsecond value in queries. The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate <  631180800232
```

Find departments with no employees:

```
SELECT OBJECT(d) from DeptBean d where d.emps IS EMPTY
```

Find all employees whose earn more than Bob:

```
SELECT OBJECT(e) FROM EmpBean e, EmpBean b
WHERE b.name = 'Bob' AND e.salary + e.bonus > b.salary + b.bonus
```

Find the employee with the largest bonus:

```
SELECT OBJECT(e) from EmpBean e  WHERE e.bonus =
(SELECT MAX(e1.bonus) from EmpBean e1)
```

The above queries all return EJB objects. A finder method query must always return an EJB Object for the home. A select method query can in addition return CMP fields or other EJB Objects not belonging to the home.

The following would be valid select method queries for EmpBean. Return the manager for each department:

```
SELECT  d.mgr FROM DeptBean d
```

Return department 42 manager's name:

```
SELECT  d.mgr.name FROM DeptBean d WHERE  d.deptno = 42
```

Return the names of employees in department 42:

```
SELECT e.name FROM EmpBean e WHERE  e.dept.deptno=42
```

Another way to write the same query is:

```
SELECT e.name from DeptBean d, IN (d.emps) AS e WHERE d.deptno=42
```

Finder and select queries allow only a single CMP field or EJBObject in the SELECT clause. A select query can return aggregate values in Enterprise JavaBeans 2.1 using SUM, MIN, MAX, AVG and COUNT.

```
SELECT max(e.salary) FROM EmpBean e WHERE e.dept.deptno=42
```

The dynamic query API allows multiple expressions in the SELECT clause. The following query would be a valid dynamic query, but not a valid select or finder query:

```
SELECT  e.name, e.salary+e.bonus as total_pay , object(e), e.dept.mgr
FROM  EmpBean e
ORDER BY 2
```

The following dynamic query returns the number of employees in each department:

```
SELECT e.dept.deptno as department_number , count(*) as employee_count
FROM  EmpBean e
GROUP BY  by e.dept.deptno
ORDER BY 1
```

The dynamic query API allows queries that contain bean or value object methods:

```
SELECT object(e), e.address.format( )
FROM EmpBean e EmpBean e
```

FROM clause:

The FROM clause specifies the collections of objects to which the query is to be applied. Each collection is specified either by an abstract schema name (ASN) or by a path expression identifying a relationship. An identification variable is defined for each collection.

Conceptually, the semantics of the query is to form a temporary collection of tuples, **R**, with elements consisting of all possible combinations of objects from the collections. This collection is subject to the constraints imposed by any path relationships and by the JOIN operation. The JOIN can be either an *inner* or *outer* join.

The identification variables are bound to elements of the tuple. After forming the temporary collection, the search conditions of the WHERE clause are applied to R, and yield a new temporary collection, **R1**. The ORDER BY, GROUP BY, HAVING, and SELECT clauses are applied to R1 to yield the final result.

```
from_clause::=FROM identification_variable_declaration [, {identification_variable_declaration |
                          collection_member_declaration } ]*

identification_variable_declaration::= range_variable_declaration [join]*

join :=  [ { LEFT [OUTER] | INNER  }]  JOIN  {collection_valued_path_expression | single_valued_path_expression}
 [AS] identifier
```

Examples: Joining collections

`DeptBean` contains records 10, 20, and 30. `EmpBean` contains records 1, 2, and 3 that are related to department 10, and records 4 and 5 that are related to department 20. Department 30 has no employees.

```
SELECT d FROM  DeptBean AS d, EmpBean AS e
WHERE d.name = e.name
```

The comma syntax performs an inner join resulting in all possible combinations. In this example, R would consist of 15 tuples (3 departments x 5 employees). If any collection is empty, then R is also empty. The keyword *AS* is optional.

This example shows that a collection can be joined with itself.

```
SELECT d FROM DeptBean AS d, DeptBean AS d1
```

R would consist of 9 tuples (3 departments x 3 departments).

Examples: Relationship joins

A collection can be a relationship based on a previously declared identifier as in

```
SELECT e FROM DeptBean AS d , IN (d.emps) AS e
```

R would contain 5 tuples. Department 30 would not appear in R because it contains no employees. Department 10 would appear in 3 tuples and department 20 would appear in 2 tuples. IN can only refer to multi-valued relationships. The following is not valid

```
SELECT m FROM EmpBean e,  IN( e.dept.mgr) as m   INVALID
```

When joining with a relationship the alternate syntax INNER JOIN (keyword INNER is optional) can also be used, as shown here.

```
SELECT e FROM DeptBean AS d INNER JOIN d.emps AS e
```

An ASN declaration (**d** in the above query) can be followed by one or more join clauses. The relationship following the JOIN keyword must be related (directly or indirectly) to the ASN declaration. Unlike the case with the IN clause, relationships used in a join clause can be single- or multi-valued. This query has the same semantics as the query

```
SELECT e FROM DeptBean AS d , IN (d.emps) AS e
```

You can use multiple joins together.

```
SELECT m FROM EmpBean e JOIN e.dept d JOIN d.mgr m
```

This is equivalent to

```
SELECT m FROM EmpBean e JOIN e.dept.mgr m
```

Examples: OUTER JOIN

An OUTER JOIN results in a temporary collection that contains combinations of the *left* and *right* operands, subject to the relationship constraints and such that the left operand always appears in R. In the example an outer join results in a temporary collection R that contains department 30, even though the collection **d.emps** is empty. The tuple contains Department 30 along with a NULL value. References to **e** in the query yields a null value.

```
SELECT e FROM DeptBean AS d LEFT OUTER JOIN d.emps AS e
```

The keyword OUTER is optional, as shown here..

```
SELECT e FROM DeptBean AS d LEFT JOIN d.emps AS e
```

You can also use combinations of INNER and OUTER JOIN.

```
SELECT m FROM EmpBean e JOIN e.dept d LEFT JOIN d.mgr m
```

Inheritance in EJB query:

If an Enterprise JavaBeans (EJB) inheritance hierarchy has been defined for an abstract schema, using the abstract schema name in a query statement implies the collection of objects for that abstract schema as well as all subtypes.

Example: Inheritance

Suppose that bean `ManagerBean` is defined as a subtype of `EmpBean` and `ExecutiveBean` is a subtype of `ManagerBean` in an EJB inheritance hierarchy. The following query returns employees as well as managers and executives:

```
SELECT OBJECT(e) FROM EmpBean e
```

Path expressions:

A path expression is an identification variable followed by the navigation operator (.) and a container managed persistence (CMP) or relationship name.

A path expression that leads to a cmr field can be further navigated if the cmr field is single-valued. If the path expression leads to a multi-valued relationship, then the path expression is terminal and cannot be further navigated. If the path expression leads to a CMP field whose type is a value object, it is possible to navigate to attributes of the value object.

Example: Value object

Assume that `address` is a CMP field for `EmpBean`, which is a value object.

```
SELECT  object(e)  FROM EmpBean e
WHERE  e.address.distance('San Jose') < 10  and e.address.zip = 95037
```

It is best to use the composer pattern to map value object attributes to relational columns if you intend to search on value attributes. If you store value objects in serialized format, then each value object must be retrieved from the database and deserialized. Value object methods can only be done in dynamic queries.

A path expression can also navigate to a bean method. The method must be defined on either the remote or local bean interface. Methods can only be used in dynamic queries. You cannot mix both remote and local methods in a single query statement.

If the query contains remote methods, the dynamic query must be executed using the query remote interface. Using the query remote interface causes the query service to activate beans and create instances of the remote bean interface

Likewise, a query statement with local bean methods must be executed with the query local interface. This causes the query service to activate beans and local interface instances.

Do not use get methods to access CMP and cmr fields of a bean.

If a method has overloaded definitions, the overloaded methods must have different number of parameters.

Methods must have non-void return types and method arguments and return types must be either primitive types byte, short, int, long, float, double, boolean, char or wrapper types from the following list:

Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Date

If any input argument to a method is NULL, it is assumed the method returns a NULL value and the method is not invoked.

A collection valued path expression can be used in the FROM clause as a collection member declaration, and with the IS EMPTY, MEMBER OF, and EXISTS predicates in the WHERE clause.

`FROM EmpBean e WHERE e.dept.mgr.name='Bob'`	OK
`FROM EmpBean e WHERE e.dept.emps.name='BOB'`	INVALID -- cannot navigate through emps because it is multivalued
`FROM EmpBean e,  IN (e.dept.emps) e1` `WHERE e1.name='BOB'`	OK
`FROM EmpBean e WHERE e.dept.emps IS EMPTY`	OK

WHERE clause:

The WHERE clause lists search conditions for items to add to a result set.

The WHERE clause contains search conditions composed of the following:
- literal values
- input parameters
- expressions
- basic predicates
- quantified predicates
- BETWEEN predicate
- IN predicate
- LIKE predicate
- NULL predicate
- EMPTY collection predicate
- MEMBER OF predicate
- EXISTS predicate
- IS OF TYPE predicate

If the search condition evaluates to TRUE, the tuple is added to the result set.

Literals:

Literals can be considered constants that do not change in value.

A string literal is enclosed in single quotes. A single quote that occurs within a string literal is represented by two single quotes. For example: 'Tom''s'. A string literal cannot exceed the maximum length that is supported by the underlying persistent datastore.

A numeric literal can be any of the following:
- an exact value such as 57, -957, +66
- any value supported by Java long
- a decimal literal such as 57.5, -47.02
- an approximate numeric value such as 7E3, -57.4E-2

A decimal or approximate numeric value must be in the range supported by the underlying persistent datastore.

A boolean literal can be the keyword TRUE or FALSE and is case insensitive.

Input parameters:

Input parameters are designated by the question mark followed by a number; for example: `?2`. Input parameters are numbered starting at 1 and correspond to the arguments of the finder or select method; therefore, a query must not contain an input parameter that exceeds the number of input arguments.

An input parameter can be a primitive type of byte, short, int, long, float, double, boolean, char or wrapper types of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Char, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, an EJBObject, or a binary data string in the form of Java byte[].

An input parameter must not have a NULL value. To search for the occurrence of a NULL value the NULL predicate should be used.

Expressions:

An expression specifies a value.

Conditional expressions can consist of comparison operators and logical operators (AND, OR, NOT).

Arithmetic expressions can be used in comparison expressions and can be composed of arithmetic operations and functions, path expressions that evaluate to a numeric value and numeric literals and numeric input parameters.

String expressions can be used in comparison expressions and can be composed of string functions, path expressions that evaluate to a string value and string literals and string input parameters. A CMP field of type char is handled as if it were a string of length 1.

Binary expressions can be used in comparison expressions and can be composed of path expressions that evaluate to the Java byte[] type as well as input parameters of type byte[].

Boolean expressions can be used with = and <> comparison and can be composed of path expressions that evaluate to a boolean value and TRUE and FALSE keywords and boolean input parameters.

Reference expressions can be used with = and <> comparison and can be composed of path expressions that evaluate to a cmr field, an identification variable and an input parameter whose type is an EJB reference

Four different expression types are supported for working with date-time types. For portability the java.util.Calendar type should be used. DB2 style date, time and timestamp expressions are supported if the datastore is DB2 and the CMP field is of type java.util.Date, java.sql.Date, java.sql.Time or java.sql.Timestamp. If you use DB2 UDB, you might obtain a syntax error when using the java.sql.Timestamp.ojbect. You must use the syntax TIMESTAMP 'yyyy-mm-dd hh:mm:ss.nnnn'.

A Calendar type can be compared to another Calendar type, an exact numeric literal or input parameter of type long whose value is the standard Java long millisecond value.

The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate <  631180800232
```

Date expressions can be used in comparison expressions and can be composed of operators + - , date duration expressions and date functions, path expressions that evaluate to a date value, string representation of a date and date input parameters.

Time expressions can be used in comparison expressions and can be composed of operators + - , time duration expressions and time functions, path expressions that evaluate to a time value, string representation of time and time input parameters.

Timestamp expressions can be used in comparison expressions and can be composed of operators + - , timestamp duration expressions and timestamp functions, path expressions that evaluate to a timestamp value, string representation of a timestamp and timestamp input parameters.

Standard bracketing ( ) for ordering expression evaluation is supported.

The operators and their precedence order from highest to lowest are:
- Navigation operator (.)
- Arithmetic operators in precedence order:
  - + - unary
  - * / multiply, divide
  - + - add, subtract
- Comparison operators: =, >, <, >=, <=, <>(not equal)
- Logical operator NOT
- Logical operator AND
- Logical operator OR

Null value semantics:

The following describe the semantics of NULL values.
- Comparison or arithmetic operations with an unknown (NULL) value yield an unknown value
- In a Java 2 platform, Enterprise Edition (J2EE) version 1.3 application, a path expression uses an outer-join semantic where a NULL field or cmr value evaluates to NULL. In J2EE version 1.4, the path expression uses an inner-join semantic.
- The IS NULL and IS NOT NULL operators can be applied to path expressions and return TRUE or FALSE. Boolean operators AND, OR and NOT use three valued logic.

AND	**True**	**False**	**Unknown**
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

OR	**True**	**False**	**Unknown**
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

	NOT
True	False
False	True
Unknown	Unknown

Example: Null value semantics

```
select object(e) from EmpBean where e.salary > 10  and e.dept.budget > 100
```

If salary is NULL the evaluation of `e.salary > 10` returns unknown and the employee object is not returned. If the cmr field dept or budget is NULL evaluation of `e.dept.budget > 100` returns unknown and the employee object is not returned.

```
select object(e) from EmpBean where e.dept.budget is null
```

In J2EE 1.3 if dept or budget is NULL evaluation of `e.dept.budget` is null returns TRUE and the employee object is returned. In J2EE 1.4 the employee object is returned only if budget is NULL.

```
select object(e) from EmpBean e ,  in (e.dept.emps) e1 where e1.salary > 10
```

If dept is NULL, then the multivalued path expression `e.dept.emps` results in an empty collection (not a collection that contains a NULL value). An employee with a null dept value will not be returned.

```
select object(e) from EmpBean e where  e.dept.emps is empty
```

If dept is NULL the evaluation of the predicate in unknown and the employee object is not returned.

```
select object(e) from EmpBean e , EmpBean e1  where e member of  e1.dept.emps
```

If dept is NULL evaluation of the member of predicate returns unknown and the employee is not returned.

Date time arithmetic and comparisons:

DATE, TIME and TIMESTAMP values can be compared with another value of the same type. Comparisons are chronological. Date time values can also be incremented, decremented, and subtracted.

If the datastore is DB2, then DB2 string representation of DATE, TIME and TIMESTAMP types can also be used. A string representation of a date or time can use ISO, USA, EUR or JIS format. A string representation of a timestamp uses ISO format.

Format	Date format	Date examples	Time format	Time examples
ISO	yyyy-mm-dd	1987-02-24 1987-2-24	hh.mm.ss	13.50.00 13.50
USA	mm/dd/yyyy	2/24/1987	hh:mm AM or PM	1:50 pm 02:10 AM
EUR	dd.mm.yyyy	24.02.1987 24.2.1987	hh.mm.ss	13.50.00 13.55
JIS	yyyy-mm-dd	1987-02-24	hh:mm:ss	13:50 13:50:05

Example 1: Date time arithmetic comparisons

```
e.hiredate > '1990-02-24'
```

The timestamp of February 24th, 1990 1:50 pm can be represented as follows:

```
'1990-02-24-13.50.00.000000'  or
'1990-02-24-13.50.00'
```

If the datastore is DB2, DB2 decimal durations can be used in expressions and comparisons. A date duration is a decimal(8,0) number that represents the difference between two dates in the format YYYYMMDD. A time duration is a decimal(6,0) number that represents the difference between two time values as HHMMSS. A timestamp duration is a decimal(20,6) number representing the differences between two timestamp values as YYYYMMDDHHMMSS.ZZZZZZ (ZZZZZZ is the number of microseconds and is to the right of the decimal point) .

Two date values (or time values or timestamp values) can be subtracted to yield a duration. If the second operand is greater than the first the duration is a negative decimal number. A duration can be added or subtracted from a datetime value to yield a new datetime value.

Example 2: Date time arithmetic comparisons

`DATE('3/15/2000') - '12/31/1999'` results in a decimal number 215 which is a duration of 0 years, 2 months and 15 days.

Durations are really decimal numbers and can be used in arithmetic expressions and comparisons.

`( DATE('3/15/2000') - '12/31/1999' ) + 14 > 215` evaluates to TRUE.

`DATE('12/31/1999') + DECIMAL(215,8,0)` results in a date value 3/15/2000.

`TIME('11:02:26') - '00:32:56'` results in a decimal number 102930 which is a time duration of 10 hours, 29 minutes and 30 seconds.

`TIME('00:32:56') + DECIMAL(102930,6,0)` results in a time value of 11:02:26.

`TIME('00:00:59') + DECIMAL(240000,6,0)` results in a time value of 00:00:59.

`e.hiredate + DECIMAL(500,8,0) > '2000-10-01'` means compare the hiredate plus 5 months to the date 10/01/2000.

Basic predicates:

A basic predicate compares two values.

Basic predicates can be of two forms, for example:

```
expression-1  comparison-operator  expression-2
expression-3  comparison-operator ( subselect )
```

The subselect must not return more than one value and the subselect cannot return a type of an Enterprise JavaBean (EJB) reference. Boolean types and reference types only support = and <> comparisons.

Example: Basic predicates

```
d.name='Java Development'
e.salary > 20000
e.salary > ( select avg(e.salary) from EmpBean e)
```

Quantified predicates:

A quantified predicate compares a value with a set of values produced by a subselect.

Use the syntax:

```
expression   comparison-operator   SOME | ANY | ALL    ( subselect )
```

The expression must not evaluate to a reference type.

When SOME or ANY is specified the result of the predicate is as follows:
- TRUE if the comparison is true for at least one value returned by the subselect.
- FALSE if the subselect is empty or if the comparison is false for every value returned by the subselect.
- UNKNOWN if the comparison is not true for all of the values returned by the subselect and at least one of the comparisons is unknown because of a null value.

When ALL is specified the result of the predicate is as follows:
- TRUE if the subselect returns empty or if the comparison is true to every value returned by the subselect.
- FALSE if the comparison is false for at least one value returned by the subselect.
- UNKNOWN if the comparison is not false for all values returned by the subselect and at least one comparison is unknown because of a null value.

BETWEEN predicate:

The BETWEEN predicate determines whether a given value lies between two other given values.

The syntax for the predicate is:

```
expression  [NOT]  BETWEEN expression-2  AND expression-3
```

The expression must not evaluate to a boolean or reference type.

Example: BETWEEN predicate

```
e.salary BETWEEN 50000 AND 60000
```

is equivalent to:

```
e.salary >= 50000  AND e.salary <= 60000
e.name NOT BETWEEN 'A' AND 'B'
```

is equivalent to:

```
e.name < 'A'  OR e.name > 'B'
```

IN predicate:

The IN predicate compares a value to a set of values.

It can have one of two forms:

```
expression [NOT] IN  ( subselect )
expression [NOT] IN  ( value1, value2,  .... )
```

ValueN can either be a literal value or an input parameter. The expression cannot evaluate to a reference type.

Example: IN predicate

```
e.salary IN ( 10000, 15000 )
```

is equivalent to

```
( e.salary = 10000  OR  e.salary = 15000 )
e.salary IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary = ANY   ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
e.salary NOT IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary <> ALL    ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

LIKE predicate:

The LIKE predicate searches a string value for a certain pattern.

The syntax for this predicate is:

```
string-expression   [NOT]  LIKE  pattern    [ ESCAPE escape-character ]
```

The pattern value is a string literal or parameter marker of type string in which the underscore (_) stands for any single character and percent (%) stands for any sequence of characters (including empty sequence). Any other character stands for itself. The escape character can be used to search for character _ and %. The escape character can be specified as a string literal or an input parameter.

If the string-expression is null, then the result is unknown.

If both string-expression and pattern are empty, then the result is true.

Example: LIKE predicate

- '' LIKE '' is true
- '' LIKE '%' is true
- e.name LIKE '12%3' is true for '123' '12993' and false for '1234'
- e.name LIKE 's_me' is true for 'some' and 'same', false for 'soome'
- e.name LIKE '/_foo' escape '/' is true for '_foo', false for 'afoo'
- e.name LIKE '//_foo' escape '/' is true for '/afoo' and for '/bfoo'
- e.name LIKE '///_foo' escape '/' is true for '/_foo' but false for '/afoo'

NULL predicate:

The NULL predicate tests for null values.

Use the syntax:

```
single-valued-path-expression IS [NOT] NULL
```

Example: NULL predicate

```
e.name IS NULL
e.dept.name IS NOT NULL
e.dept IS NOT NULL
```

EMPTY collection predicate:

You can use the EMPTY collection predicate to test if a multivalued relationship has no members.

Use the following syntax:

```
collection-valued-path-expression  IS [NOT]  EMPTY
```

Example: Empty collection predicate

To find all departments with no employees:

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d.emps IS EMPTY
```

MEMBER OF predicate:

This expression tests whether the object reference specified by the single valued path expression or input parameter is a member of the designated collection.

If the collection valued path expression designates an empty collection the value of the MEMBER OF expression is FALSE.

```
{ single-valued-path-expression | input_parameter } [ NOT ] MEMBER [ OF ] collection-valued-path-expression
```

Example: MEMBER OF predicate

Find employees that are not members of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e , DeptBean d
WHERE e NOT MEMBER OF d.emps AND d.deptno = ?1
```

Find employees whose manager is a member of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e, DeptBean d
WHERE e.dept.mgr MEMBER  OF d.emps  and d.deptno=?1
```

EXISTS predicate:

The exists predicate tests for the presence or absence of a condition specified by a subselect.

Use the syntax:

```
EXISTS ( subselect )
EXISTS collection-valued-path-expression
```

The result of EXISTS is true if the subselect returns at least one value or the path expression evaluates to a nonempty collection, otherwise the result is false.

To negate an EXISTS predicate, precede it with the logical operator NOT.

Example: EXISTS predicate

Return departments that have at least one employee earning more than 1000000:

```
SELECT  OBJECT(d) FROM  DeptBean d
WHERE EXISTS ( SELECT  1  FROM IN (d.emps) e WHERE  e.salary > 1000000 )
```

Return departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d
WHERE NOT EXISTS  ( SELECT 1 FROM IN (d.emps) e)
```

The above query can also be written as follows:

```
SELECT OBJECT(d) FROM DeptBean d WHERE NOT EXISTS d.emps
```

IS OF TYPE predicate:

The IS OF TYPE predicate is used to test the type of an Enterprise JavaBeans (EJB) reference. It is similar in function to the Java instance of operator.

IS OF TYPE is used when several abstract beans have been grouped into an EJB inheritance hierarchy. The type names specified in the predicate are the bean abstract names. The ONLY option can be used to specify that the reference must be exactly this type and not a subtype.

```
identification-variable IS OF TYPE ( [ONLY] type-1, [ONLY] type-2, ..... )
```

Example: IS OF TYPE predicate

Suppose that bean `ManagerBean` is defined as a subtype of `EmpBean` and `ExecutiveBean` is a subtype of `ManagerBean` in an EJB inheritance hierarchy.

The following query returns employees as well as managers and executives:

```
SELECT  OBJECT(e) FROM EmpBean e
```

If you are interested in objects which are employees and not managers and not executives:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e IS OF TYPE( ONLY EmpBean )
```

If you are interested in object which are managers or executives:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e IS OF TYPE( ManagerBean)
```

The above query is equivalent to the following query:

```
SELECT  OBJECT(e) FROM ManagerBean e
```

If you are interested in managers only and not executives:

```
SELECT  OBJECT(e) FROM  EmpBean e WHERE e IS OF TYPE( ONLY ManagerBean)
```

or:

```
SELECT  OBJECT(e) FROM  ManagerBean e
WHERE  e IS OF TYPE (ONLY ManagerBean)
```

Scalar functions:

An Enterprise JavaBeans (EJB) query contains scalar functions for doing type conversions, string manipulation, and for manipulating date-time values.

The list of scalar functions is documented in the topic EJB query: Scalar functions.

Example: Scalar functions

Find employees hired in 1999:

```
SELECT OBJECT(e) FROM EmpBean e where YEAR(e.hireDate) = 1999
```

The only scalar functions that are guaranteed to be portable across backend datastore vendors are the following:

- ABS
- MOD
- SQRT
- CONCAT
- LENGTH
- LOCATE
- SUBSTRING
- UCASE
- LCASE

The other scalar functions should be used only when DB2 is the backend datastore.

EJB query: Scalar functions:

Enterprise JavaBeans (EJB) query contains scalar built-in functions for doing type conversions, string manipulation, and for manipulating date-time values.

EJB query scalar built-in functions are listed below:

Numeric functions

```
ABS ( < any numeric datatype > ) -> < any numeric datatype >
MOD ( <int>, <int> ) -> int
SQRT ( < any numeric datatype > ) -> Double
```

Type conversion functions

```
CHAR ( < any  numeric datatype > ) ->  string
CHAR ( <  string  > ) ->  string
CHAR ( < any datetime  datatype >  [, Keyword k ]) ->  string
```

Datetime datatype is converted to its string representation in a format specified by the keyword k. The valid keywords values are ISO, USA, EUR or JIS. If k is not specified the default is ISO.

```
BIGINT ( < any numeric datatype > ) -> Long
BIGINT ( < string > ) -> Long
```

The function in the second line of the following code converts the argument to an integer n by truncation, and returns the date that is n-1 days after January 1, 0001:

```
DATE ( < date string > ) -> Date
DATE (  < any numeric datatype>) -> Date
```

The following function returns date portion of a timestamp:

```
DATE( timestamp ) -> Date
DATE ( < timestamp-string > ) -> Date
```

The following function converts number to decimal with optional precision p and scale s.

```
DECIMAL ( < any numeric datatype > [, p [ ,s ] ] ) -> Decimal
```

The following function converts string to decimal with optional precision p and scale s.

```
DECIMAL ( < string > [ , p [ , s ] ] ) -> Decimal
DOUBLE ( < any numeric datatype > ) -> Double
DOUBLE ( < string > ) -> Double
FLOAT ( < any numeric datatype > ) -> Double
FLOAT ( < string > ) -> Double
```

Float is a synonym for DOUBLE.

```
INTEGER ( < any numeric datatype > ) -> Integer
INTEGER ( < string > ) -> Integer

REAL ( < any numeric datatype > ) -> Float

SMALLINT ( < any numeric datatype ) -> Short
SMALLINT ( < string > ) -> Short

TIME ( < time > ) -> Time
TIME ( < time-string  > ) -> Time
TIME ( < timestamp > ) -> Time
TIME ( < timestamp-string  > ) -> Time

TIMESTAMP ( < timestamp > ) -> Timestamp
TIMESTAMP ( < timestamp-string > ) -> Timestamp
```

String functions

```
CONCAT ( <string>, <string>  ) -> String
```

The following function returns a character string representing absolute value of the argument not including its sign or decimal point. For example, `digits( -42.35)` is ″4235″.

```
DIGITS ( Decimal d  ) -> String
```

The following function returns the length of the argument in bytes. If the argument is a numeric or datetime type, it returns the length of internal representation.

```
LENGTH ( < string >  ) -> Integer
```

The following function returns a copy of the argument string where all upper case characters have been converted to lower case.

```
LCASE ( < string > ) -> String
```

The following function returns the starting position of the first occurrence of argument 1 inside argument 2 with optional start position. If not found, it returns 0.

```
LOCATE ( String s1 , String s2  [, Integer start ] ) -> Integer
```

The following function returns a substring of s beginning at character m and containing n characters. If n is omitted, the substring contains the remainder of string s. The result string is padded with blanks if needed to make a string of length n.

```
SUBSTRING ( String s ,  Integer m [ , Integer n ] ) -> String
```

The following function returns a copy of the argument string where all lower case characters have been converted to upper case.

```
UCASE ( < string > ) -> String
```

Date - time functions

The following function returns the day portion of its argument. For a duration, the return value can be -99 to 99.

```
DAY (  Date ) ->  Integer
DAY ( < date-string > ) ->  Integer
DAY ( < date-duration > ) -> Integer
DAY ( Timestamp  ) ->  Integer
DAY ( < timestamp-string > ) ->  Integer
DAY ( < timestamp-duration > ) -> Integer
```

The following function returns one more than number of days from January 1, 0001 to its argument.

```
DAYS ( Date  ) ->  Integer
DAYS ( < Date-string > ) ->  Integer
DAYS ( Timestamp  ) ->  Integer
DAYS ( < timestamp-string > ) ->  Integer
```

The following function returns the hour part of its argument. For a duration, the return value can be -99 to 99.

```
HOUR ( Time ) -> Integer
HOUR ( < time-string > ) -> Integer
HOUR ( < time-duration > ) ->  Integer
HOUR ( Timestamp ) -> Integer
HOUR ( < timestamp-string > ) -> Integer
HOUR ( < timestamp-duration >  ) -> Integer
```

The following function returns the microsecond part of its argument.

```
MICROSECOND ( Timestamp ) -> Integer
MICROSECOND ( < timestamp-string > ) -> Integer
MICROSECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the minute part of its argument. For a duration, the return value can be -99 to 99.

```
MINUTE ( Time ) -> Integer
MINUTE ( < time-string > ) -> Integer
MINUTE ( < time-duration > ) ->  Integer
MINUTE ( Timestamp ) -> Integer
MINUTE ( < timestamp-string > ) -> Integer
MINUTE ( < timestamp-duration >  ) -> Integer
```

The following function returns the month portion of its argument. For a duration, the return value can be -99 to 99.

```
MONTH (  Date ) ->  Integer
MONTH ( < date-string > ) ->  Integer
MONTH ( < date-duration > ) -> Integer
MONTH ( Timestamp  ) ->  Integer
MONTH ( < timestamp-string > ) ->  Integer
MONTH ( < timestamp-duration > ) -> Integer
```

The following function returns the second part of its argument. For a duration, the return value can be -99 to 99.

```
SECOND ( Time ) -> Integer
SECOND ( < time-string > ) -> Integer
SECOND ( < time-duration > ) ->  Integer
SECOND ( Timestamp ) -> Integer
SECOND ( < timestamp-string > ) -> Integer
SECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the year portion of its argument. For a duration, the return value can be -9999 to 9999.

```
YEAR (  Date ) ->  Integer
YEAR ( < date-string > ) ->  Integer
YEAR ( < date-duration > ) -> Integer
YEAR ( Timestamp  ) ->  Integer
YEAR ( < timestamp-string > ) ->  Integer
YEAR ( < timestamp-duration > ) -> Integer
```

Aggregation functions:

Aggregation functions operate on a set of values to return a single scalar value. You can use these functions in the select and subselect methods.

The following example illustrates an aggregation:

```
SELECT  SUM (e.salary) FROM EmpBean e WHERE e.dept.deptno =20
```

This aggregation computes the total salary for department 20.

The aggregation functions are AVG, COUNT, MAX, MIN, and SUM. The syntax of an aggregation function is illustrated in the following example:

```
aggregation-function  (    [ ALL |  DISTINCT ]  expression )
```

or:

```
COUNT( [ ALL |  DISTINCT ] identification-variable )
```

or:

```
COUNT( * )
```

The DISTINCT option eliminates duplicate values before applying the function. ALL is the default option and does not eliminate duplicates. Null values are ignored in computing the aggregate function except in the cases of COUNT(*) and COUNT(identification-variable), which return a count of all the elements in the set.

If your datastore is Informix, you must limit the expression argument to a single valued path expression when using the COUNT function or the DISTINCT forms of the functions SUM, AVG, MIN, and MAX.

Defining return type

For a select method using an aggregation function, you can define the return type as a primitive type or a wrapper type. The return type must be compatible with the return type from the datastore. The MAX and MIN functions can apply to any numeric, string or datetime datatype and return the corresponding datatype. The SUM and AVG functions take a numeric type as input, and return the same numeric type that is used in the datastore. The COUNT function can take any datatype, and returns an integer.

When applied to an empty set, the SUM, AVG, MAX, and MIN functions can return a null value. The COUNT function returns zero (0) when it is applied to an empty set. Use wrapper types if the return value might be NULL; otherwise, the container displays an ObjectNotFound exception.

Using GROUP BY and HAVING

The set of values that is used for the aggregate function is determined by the collection that results from the FROM and WHERE clause of the query. You can divide the set into groups and apply the aggregation function to each group. To perform this action, use a GROUP BY clause in the query. The GROUP BY clause defines grouping members, which comprise a list of path expressions. Each path expression specifies a field that is a primitive type of byte, short, int, long, float, double, boolean, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time or java.sql.Timestamp.

The following example illustrates the use of the GROUP BY clause in a query that computes the average salary for each department:

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

In division of a set into groups, a NULL value is considered equal to another NULL value.

Just as the WHERE clause filters tuples (that is, records of the return collection values) from the FROM clause, the groups can be filtered using a HAVING clause that tests group properties involving aggregate functions or grouping members:

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e
GROUP BY e.dept.deptno
HAVING  COUNT(*) > 3  AND  e.dept.deptno > 5
```

This query returns the average salary for departments that have more than three employees and the department number is greater than five.

It is possible to use a HAVING clause without a GROUP BY clause, in which case the entire set is treated as a single group, to which the HAVING clause is applied.

SELECT clause:

The SELECT clause consists of either a single identification variable that is defined in the FROM clause, or a single valued path expression that evaluates to an object reference or container managed persistence (CMP) value. You can use the DISTINCT keyword to eliminate duplicate references.

For finder and select queries, the syntax of the SELECT clause is illustrated in the following example:

```
SELECT  [  ALL | DISTINCT  ]
 { single-valued-path-expression  |  aggregation expression  |  OBJECT ( identification-variable )  }
```

For a query that defines a finder method, the query must return an object type consistent with the home that is associated with the finder method. For example, a finder method for a department home can not return employee objects.

Example: SELECT clause

Find all employees that earn more than John:

```
SELECT OBJECT(e) FROM EmpBean ej, EmpBean e
WHERE  ej.name = 'John'  and e.salary > ej.salary
```

Find all departments that have one or more employees who earn less than 20000:

```
SELECT DISTINCT  e.dept  FROM EmpBean e where e.salary < 20000
```

A select method query can have a path expression that evaluates to an arbitrary value:

```
SELECT  e.dept.name  FROM EmpBean e where e.salary < 2000
```

The previous query returns a collection of name values for those departments having employees earning less than 20000.

A select method query can return an aggregate value:

```
SELECT  avg(e.salary)  FROM EmpBean e
```

Example: Valid dynamic queries

For dynamic queries the syntax is as follows:

```
SELECT { ALL | DISTINCT }  [ selection , ]*  selection
selection  ::= { expression   |  scalar-subselect [[AS] id ] }
```

A scalar-subselect is a subselect that returns a single value.

The following are examples of dynamic queries:

```
SELECT e.name, e.salary+e.bonus as total_pay from EmpBean e
SELECT  SUM( e.salary+e.bonus) from EmpBean e where e.dept.deptno = ?1
```

ORDER BY clause:

The ORDER BY clause specifies an ordering of the objects in the result collection

Use the syntax:

```
ORDER BY  [ order_element ,]* order_element
order_element ::= { path-expression | integer } [ ASC | DESC ]
```

The path expression must specify a single valued field that is a primitive type of byte, short, int, long, float, double, char or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Character, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp.

ASC specifies ascending order and is the default. DESC specifies descending order.

Integer refers to a selection expression in the SELECT clause.

Example: ORDER BY clause

Return department objects in decreasing deptno order:

```
SELECT  OBJECT(d) FROM  DeptBean d ORDER BY  d.deptno DESC
```

Return employee objects sorted by department number and name:

```
SELECT  OBJECT(e) FROM EmpBean e ORDER BY e.dept.deptno ASC,  e.name DESC
```

UNION operation:

The UNION clause specifies a combination of the output of two subqueries. The two queries must return the same number of elements and compatible types.

For the purposes of UNION, all Enterprise JavaBeans (EJB) types in the same inheritance hierarchy are considered compatible. UNION requires that equality be defined for the element types.

```
query_expression := query_term [UNION [ALL] query_term]*

query_term :=  {select_clause_dynamic  from_clause  [where_clause]
 [group_by_clause] [having_clause] } | (query_expression) }
```

You cannot use dependent value objects with UNION.

UNION ALL combines all results together in a single collection.

UNION combines results but eliminates duplicates.

If ORDER BY is used together with UNION, the ORDER BY must refer to selection expression using integer numbers.

Examples: UNION operation

This example returns a collection of all employee objects of type EmpBean and all manager objects of type ManagerBean where ManagerBean is a subtype of EmpBean.

```
select e from EmpBean e union all select m from DeptBean d, in(d.mgr) m
```

This example shows a query that is not valid, because EmpBean and DeptBean are not compatible.

```
select e from EmpBean e union all select d from DeptBean d
```

Subqueries:

A subquery can be used in quantified predicates, the EXISTS predicate, or the IN predicate. A subquery should only specify a single element in the SELECT clause.

When a path expression appears in a subquery, the identification variable of the path expression must be defined either in the subquery, in one of the containing subqueries, or in the outer query. A scalar subquery is a subquery that returns one value. A scalar subquery can be used in a basic predicate and in the SELECT clause of a dynamic query.

Example: Subqueries

```
SELECT  OBJECT(e) FROM  EmpBean e
WHERE e.salary > ( SELECT  AVG(e1.salary) FROM  EmpBean e1)
```

The above query returns employees who earn more than average salary of all employees.

```
SELECT  OBJECT(e) FROM EmpBean e WHERE  e.salary >
( SELECT AVG(e1.salary) FROM  IN  (e.dept.emps) e1  )
```

The above query returns employees who earn more than average salary of their department.

```
SELECT  OBJECT(e) FROM EmpBean e WHERE e.salary =
( SELECT MAX(e1.salary) FROM IN (e.dept.emps) e1  )
```

The above query returns employees who earn the most in their department.

```
 SELECT OBJECT(e) FROM EmpBean e
WHERE e.salary > ( SELECT AVG(e.salary) FROM EmpBean e1
WHERE YEAR(e1.hireDate) =  YEAR(e.hireDate)  )
```

The above query returns employees who earn more than the average of employees hired in same year.

EJB query language limitations and restrictions:

When using the Enterprise JavaBeans (EJB) query language on the product, deviations can be seen in comparison to standard EJB query language. The limitations and restrictions you must be aware of are listed in the following section.

This topic outlines current known limitations and restrictions.

- EJB query language (QL) queries involving enterprise beans with keys made up of relationships to other enterprise beans appear as not valid and cause errors at deployment time. This is a known problem.
- The IBM EJB QL support extends the EJB 2.0 specification in various ways, including relaxing some restrictions, adding support for more DB2 functions, and so on. If portability across various vendor databases or EJB deployment tools is a concern, then care should be taken to write all EJB QL queries strictly according to Chapter 11 in the EJB 2.0 specification.
- Pre-loading across m:n relationships results in the generation of inaccurate structured query language (SQL). This is a known limitation that may be addressed in the future.
- Pre-loading across self referencing relationships causes inaccurate SQL to be generated.
- Avoid relationships between parent and children enterprise beans within the same inheritance hierarchy that are not well-defined.
- EJB Query Language validation for EJB 2.0 JAR files currently runs as a part of the EJB-RDB Mapping validation. If a mapping document (Map.mapxmi file) does not exist in the project, the EJB queries are not validated.

EJB query compatibility issues with SQL:

Because an Enterprise JavaBeans (EJB) query is compiled into structured query language (SQL), you must be aware of compatibility issues between the Java language and SQL.

The two languages differ along the following points that can be critical to correct EJB query formulation:

- The comparison semantics of SQL strings do not exactly match those of the Java language. For example: 'A' (the letter A) and'A ' (the letter A plus a blank space) are considered equal in SQL, but not in the Java language.
- Comparisons and collating order depend on the underlying database. For example, if you are using DB2 with an EBCDIC code page, the collating order is not the same as doing the sort in a Java program. Some databases sort the NULL value low while others sort the NULL value high.
- An arithmetic overflow causes an exception in SQL, but not in the Java language.

- SQL databases have differing minimum and maximum ranges for floating point values, which can differ from floating point value ranges in the Java language. Values near the range limits of Java Double may fail to translate into SQL.
- Java methods do not translate into SQL; therefore standard EJB queries cannot include Java methods.

  Note: Only with the dynamic EJB query service can you use functions that do not translate into SQL. Such functions include Java methods and converters or composers that are used in mapping enterprise beans to relational databases (RDBs). A standard finder or select query that uses any of these functions fails at deployment time with the message ″Cannot push down query″. (You can resolve this problem by changing either the query or the mapping.) The dynamic query run time, however, processes the query by performing the operation involving the function in the application server.

Database restrictions for EJB query:

The Enterprise JavaBeans (EJB) query functions must adhere to certain restrictions for databases.

General database restriction

- All of the enterprise beans involved in a given query must map to the same data source. The EJB query does not support cross-data source join operations.
- It is possible that a structured query language (SQL) statement generated by the WebSphere Application Server deployment code generation utility for an *ejbSelect* Enterprise JavaBeans query language query returns rows in a result set that consist of null values in all columns.

  During run time persistence manager saves the set received as a result from this query. When your application retrieves the primary key of the result bean, persistence manager calls the extractor. The extractor is a method that is an EJB deploy generated class. This method returns a value of **0** for any null column entries. This value is passed back to the EJB container to forward to the application. The EJB container invokes the bean instance with the PK value of **0**. This could create a problem, as the end user cannot determine if this bean instance has a *null PK* or a *PK value of 0*.

  To avoid this, use the *IS NOT NULL* clause in the finder query to eliminate such null values from the result set.

Specific database restrictions

Different database products place different restrictions on elements that can be included in EJB query statements. Following is a list of those restrictions; check with your database administrator to see if any apply in your environment:

- Certain functions are used in queries that run against DB2 only, because these functions are not supported by other databases. These functions include date and time arithmetic expressions, certain scalar functions (those *not* listed as portable across vendors), and implied scalar functions when used for mapping certain CMP fields. For example, consider mapping an int numeric type to a decimal (5,2) type field. When deployed against a database other than DB2, a finder or select query that contains a CMP field with this particular mapping fails, producing a `Cannot push down query` error message.
- A CMP of type String, when mapped to a character large object (CLOB) in the database, cannot be used in comparison operations because the database does not support CLOB comparisons.
- Databases can impose limits on the length of string values that are used either as literals or input parameters with comparison operators. These limits can hinder query performance. For example: For DB2 on the z/OS platform, the search ″name = ?1″ can fail if the value of ?1 at run time is greater than 255 in length.
- Mapping a numeric CMP type to a column that contains a dissimilar type can cause unexpected results. For example, consider the case of mapping the int numeric type to a column of type decimal (5,2). This scenario does not preserve an exact decimal value (for example, the value 12.25) over the course of transfer from the database to the enterprise bean CMP field, and back again to the database. This

mapping causes replacement of the initial value with a whole number (in this case, 12). Consequently, you want to avoid using the CMP field in comparison operations when the CMP field uses a mapping of this nature.

- Some databases do not support a datatype that corresponds to the semantics of java.sql.Time. For example: If a CMP field of type java.sql.Time is mapped to an Oracle DATE column, comparisons on time might not produce the expected result because the year-month-day portion of the column value is truncated in the mapping.
- Some databases treat a zero length string value ('') as a null value; this approach can affect the query results. For the sake of portability, avoid the use of zero length string values.
- Some databases perform division between two integer values using integer arithmetic rules, while others use non-integer rules. This discrepancy might not be desirable in environments that use both kinds of databases. For the sake of portability, avoid the division of integer values in an EJB query.

Rules for data type manipulation in EJB query:

When using an Enterprise JavaBean (EJB) query to work with data types, certain rules must be followed.

Rules for container managed persistence (CMP) field type

You can use a CMP field of any type in a SELECT clause. You must, however, use fields of only the following types in search conditions and in grouping or ordering operations:

- Primitive types: byte, short, int, long, float, double, boolean, char
- Object types: Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.util.Date
- JDBC types: java.sql.Date, java.sql.Time, java.sql.Timestamp
- Binary string: byte[]

Converters and basic types

If ALL of the following conditions occur:

- a CMP field of one of the basic types listed previously is mapped to an SQL column using a converter
- the CMP field appears in the left hand side of a basic predicate
- the right hand side of the predicate is a literal or input parameter

then the toData() method of the converter is used to compute the SQL search value.

For example, given a converter that maps the integer value 10 to the string value ″Ten,″ the following EJB query:

```
e.cmp = 10
```

is translated into the following SQL query:

```
column = 'Ten'
```

If you include a more complicated predicate, such as in the following example:

```
e.cmp * 10 >  e.salary
```

in a finder or select query, you receive the `Cannot push down query` error message. Use the dynamic EJB query service for such multi-function queries; the dynamic query run time processes the predicate in the application server.

Overall, converters preserve equality, collating sequence, and NULL values. If a converter does not meet these requirements, avoid using it for CMP field comparison operations.

User types, converters, and composers

A user type cannot be used in a comparison operation or expression. You can, however, use subfields of the user type in a path expression. For example, consider the CMP addr field with the type `com.exam.Address`, and street, city, and state subfields. The following syntax for a query on this CMP field is not valid:

```
e.addr =  ?1
```

However, a query that designates one of the subfields is valid:

```
e.addr.street = ?1
```

A CMP field can be mapped to an SQL column using Java serialization. Using the CMP field in predicates or expressions for deployment queries usually results in the `Cannot push down query` error message. The dynamic query run time processes the expression by reading and deserializing all instances of the user type in the application server.

However, this expensive process sacrifices performance. You can maintain performance by using a composer in a deployment EJB query. In the previous example, if you want to map the addr field to a binary type, you use a composer to map each subfield to a binary column in the database.

EJB query: Reserved words:

The following words are reserved in WebSphere Application Server Enterprise JavaBeans (EJB) queries.

all, as, distinct, empty, false, from, group, having, in, is, like, select, true, union, where

Avoid using identifiers that start with underscore (for example, _integer) as these are also reserved.

EJB query: BNF syntax:

The Backus-Naur Form (BNF) is one of the most commonly used notations for specifying the syntax of programming languages or command sets. This article lists the syntax for Enterprise JavaBeans (EJB) query language.

```
EJB QL ::= [select_clause] from_clause [where_clause] [order_by_clause]

DYNAMIC EJB QL := query_expression  [order_by_clause]

query_expression := query_term [UNION [ALL] query_term]*

query_term :=  {select_clause_dynamic  from_clause  [where_clause]
 [group_by_clause] [having_clause] } | (query_expression) }  [order_by_clause]


from_clause::=FROM identification_variable_declaration
               [, {identification_variable_declaration | collection_member_declaration } ]*

identification_variable_declaration::=collection_member_declaration |
       range_variable_declaration [join]*

join :=  [ { LEFT [OUTER] | INNER  }]  JOIN  {collection_valued_path_expression | single_valued_path_expression}
 [AS] identifier

collection_member_declaration::=
       IN ( collection_valued_path_expression ) [AS] identifier

range_variable_declaration::=abstract_schema_name [AS] identifier

single_valued_path_expression ::=
       {single_valued_navigation | identification_variable}. ( cmp_field |
 method  |  cmp_field.value_object_attribute |  cmp_field.value_object_method )
          | single_valued_navigation
```

```
single_valued_navigation::=
       identification_variable.[  single_valued_cmr_field.  ]*
            single_valued_cmr_field

collection_valued_path_expression ::=
       identification_variable.[  single_valued_cmr_field.  ]*
           collection_valued_cmr_field

select_clause::= SELECT { ALL | DISTINCT }  {single_valued_path_expression |
                         identification_variable |  OBJECT ( identification_variable) |
        aggregate_functions }

select_clause_dynamic ::= SELECT { ALL | DISTINCT }  [ selection , ]*  selection

selection  ::= { expression  |  subselect }  [[AS] id ]

order_by_clause::= ORDER BY [ {single_valued_path_expression | integer}  [ASC|DESC],]*
      {single_valued_path_expression | integer}[ASC|DESC]

where_clause::= WHERE conditional_expression

conditional_expression ::= conditional_term |
                           conditional_expression OR conditional_term
conditional_term ::= conditional_factor |
                     conditional_term AND conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary::=simple_cond_expression | (conditional_expression)

simple_cond_expression ::= comparison_expression | between_expression |
       like_expression | in_expression | null_comparison_expression |
       empty_collection_comparison_expression | quantified_expression |
       exists_expression | is_of_type_expression  |  collection_member_expression

between_expression ::= expression [NOT] BETWEEN expression AND expression

in_expression ::= single_valued_path_expression [NOT] IN
           { (subselect) |  ( [ atom ,]* atom )  }

atom = { string-literal | numeric-constant | input-parameter }

like_expression ::= expression [NOT] LIKE
                {string_literal | input_parameter}
                [ESCAPE {string_literal | input_parameter}]

null_comparison_expression ::=
       single_valued_path_expression IS [ NOT ] NULL

empty_collection_comparison_expression ::=
       collection_valued_path_expression IS [NOT] EMPTY

collection_member_expression  ::=
       { single_valued_path_expression |  input_paramter }  [ NOT ] MEMBER [ OF ]
         collection_valued_path_expression

quantified_expression ::=
       expression comparison_operator  {SOME | ANY | ALL} (subselect)

exists_expression ::= EXISTS {collection_valued_path_expression |  (subselect)}

subselect ::= SELECT [{ ALL | DISTINCT }]  expression  from_clause [where_clause]
       [group_by_clause] [having_clause]

group_by_clause::= GROUP BY [single_valued_path_expression,]*
                    single_valued_path_expression

having_clause ::= HAVING conditional_expression
```

```
is_of_type_expression ::= identifier  IS OF TYPE
            ([[ONLY] abstract_schema_name,]* [ONLY] abstract_schema_name)

comparison_expression ::=  expression   comparison_operator { expression |  ( subquery ) }

comparison_operator ::=    = | > | >= | < | <= | <>

method ::=  method_name( [[expression , ]* expression ] )

expression ::= term |   expression {+|-} term

term ::=  factor |  term {*|/} factor

factor ::= {+|-} primary

primary ::= single_valued_path_expression | literal |
       ( expression ) |  input_parameter | functions | aggregate_functions

aggregate_functions :=
       AVG([ALL|DISTINCT] expression) |
       COUNT({[ALL|DISTINCT] expression | * | identification_variable }) |
       MAX([ALL|DISTINCT] expression) |
       MIN([ALL|DISTINCT] expression) |
       SUM([ALL|DISTINCT] expression) |

functions ::=
        ABS(expression) |
        BIGINT(expression) |
        CHAR({expression [,{ISO|USA|EUR|JIS}] )  |
        CONCAT (expression , expression ) |
        DATE(expression) |
        DAY({expression ) |
        DAYS( expression) |
        DECIMAL( expression [,integer[,integer]])
        DIGITS( expression) |
        DOUBLE( expression ) |
        FLOAT( expression) |
        HOUR ( expression ) |
        INTEGER( expression ) |
        LCASE ( expression) |
        LENGTH(expression) |
        LOCATE( expression, expression [, expression] ) |
        MICROSECOND( expression ) |
        MINUTE ( expression ) |
        MOD ( expression , expression ) |
        MONTH( expression ) |
        REAL( expression) |
        SECOND( expression ) |
        SMALLINT( expression )  |
        SQRT (  expression) |
        SUBSTRING( expression, expression[, expression]) |
        TIME( expression ) |
        TIMESTAMP( expression ) |
        UCASE ( expression) |
        YEAR( expression )

xrel := XREL identification_variable . { single_valued_cmr_field | collection_valued_cmr_field }
            [ , identification_variable . { single_valued_cmr_field | collection_valued_cmr_field } ]*
```

EJB specification and WebSphere query language comparison:

WebSphere Application Server extends the Enterprise JavaBeans (EJB) query language with elements of its own.

WebSphere Application Server supports the following extensions to the EJB query language.

Item	
Delimited identifiers	
Dependent Value object attributes used in path expressions	
EJB Inheritance	
EXISTS predicate	
Java methods: EJB bean methods or value object methods	dynamic query only
Multiple element select clauses	dynamic query only
SQL Date/time expressions	
Subqueries, group by, and having clauses	

Using the dynamic query service

There are times in the development process when you might prefer to use the dynamic query service rather than the regular Enterprise JavaBean (EJB) query service (which can be referred to as *deployment query*). During testing, for instance, the dynamic query service can be used at application run time, so you do not have to re-deploy your application.

About this task

Following are common reasons for using the dynamic query service rather than the regular EJB query service:

- You need to programmatically define a query at application run time, rather than at deployment.
- You need to return multiple CMP or CMR fields from a query. (Deployment queries allow only a single element to be specified in the SELECT clause.) For more information, see the Example: EJB queries article.
- You want to return a computed expression in the query.
- You want to use value object methods or bean methods in the query statement. For more information, see Path expressions.
- You want to interactively test an EJB query during development, but do not want to repeatedly deploy your application each time you update a finder or select query.

The dynamic query API is a stateless session bean; using it is similar to using any other J2EE EJB application bean. It is included in the com.ibm.websphere.ejbquery in the API package.

The dynamic query bean has both a remote and a local interface. If you want to return remote EJB references from the query, or if the query statement contains remote methods, you must use the query remote interface:

```
remote interface = com.ibm.websphere.ejbquery.Query
remote home interface = com.ibm.websphere.ejbquery.QueryHome
```

If you want to return local EJB references from the query, or if the query statement contains local methods, you must use the query local interface:

```
local interface = com.ibm.websphere.ejbquery.QueryLocal
local home interface = com.ibm.websphere.ejbquery.QueryLocalHome
```

Because it uses less application server memory, the local interface ensures better overall EJB performance than the remote.

1. Verify that the query.ear application file is installed on the application server on which your application is to run, if that server is different from the default application server created during installation of the product.

   The `query.ear` file is located in the *app_server_root* directory, where `<WAS_HOME>` is the location of the WebSphere Application Server. The product installation program installs the `query.ear` file on the default application server using a JNDI name of

   ```
   com/ibm/websphere/ejbquery/Query
   ```

   (You or the system administrator can change this name.)
2. Set up authorization for the methods executeQuery(), prepareQuery(), and executePlan() in the remote and local dynamic query interfaces to control access to sensitive data. (This step is necessary only if your application requires security.)

   Because you cannot control which ASN names, CMP fields, or CMR fields can be used in a dynamic EJB query, you or your system administrator must place restrictions on use of the methods. If, for example, a user is permitted to run the executeQuery method, he or she can run any valid dynamic query. In a production environment, you certainly want to restrict access to the remote query interface methods.
3. Write the dynamic query as part of your application client code. You can consult the following examples as query models; they illustrate which import statements to use, and so on:
   - Remote interface dynamic query example
   - Local interface dynamic query example
4. If the CMP you want to query is on a different module, you should:
   a. do a remote lookup on query.ear
   b. map the query.ear file to the server that the queried CMP bean is installed on.
5. Compile and run your client program with the file **qryclient.jar** in the classpath.

Example: Using the remote interface for Dynamic query

When you run a dynamic Enterprise JavaBeans (EJB) query using the remote interface, you are calling the executeQuery method on the Query interface. The executeQuery method has a transaction attribute of REQUIRED for this interface; therefore you do not need to explicitly establish a transaction context for the query to run.

Begin with the following import statements:

```
import com.ibm.websphere.ejbquery.QueryHome;
import com.ibm.websphere.ejbquery.Query;
import com.ibm.websphere.ejbquery.QueryIterator;
import com.ibm.websphere.ejbquery.IQueryTuple;
import com.ibm.websphere.ejbquery.QueryException;
```

Next, write your query statement in the form of a string, as in the following example that retrieves the names and ejb-references for underpaid employees:

```
String query =
"select e.name as name , object(e) as emp from EmpBean e where e.salary < 50000";
```

Create a Query object by obtaining a reference from the QueryHome class. (This class defines the executeQuery method.) Note that for the sake of simplicity, the following example uses the dynamic query JNDI name for the Query object:

```
InitialContext ic =  new InitialContext();

Object obj =  ic.lookup("com/ibm/websphere/ejbquery/Query");

QueryHome  qh =
 ( QueryHome) javax.rmi.PortableRemoteObject.narrow( obj, QueryHome.class );
Query qb = qh.create();
```

You then must specify a maximum size for the query result set, which is defined in the QueryIterator object, which is included in the Class QueryIterator. This class is included in the You then must specify a maximum size for the query result set, which is defined in the QueryIterator object, which is included in the QueryIterator API package. This example sets the maximum size of the result set to 99:

```
QueryIterator it = qb.executeQuery(query, null, null ,0, 99 );
```

The iterator contains a collection of IQueryTuple objects, which are records of the return collection values. Corresponding to the criteria of our example query statement, each tuple in this scenario contains one value of *name* and one value of *object(e)*. To display the contents of this query result, use the following code:

```
while (it.hasNext() ) {
 IQueryTuple tuple = (IQueryTuple) it.next();
 System.out.print( it.getFieldName(1) );
 String s = (String) tuple.getObject(1);
 System.out.println( s);
 System.out.println( it.getFieldName(2) );
 Emp e = ( Emp) javax.rmi.PortableRemoteObject.narrow( tuple.getObject(2), Emp.class );
 System.out.println( e.getPrimaryKey().toString());
}
```

The output from the program might look something like the following:

```
name Bob
emp 1001
name Dave
emp 298003
...
```

Finally, catch and process any exceptions. An exception might occur because of a syntax error in the query statement or a run-time processing error. The following example catches and processes these exceptions:

```
} catch (QueryException qe) {
    System.out.println("Query Exception "+ qe.getMessage() );
}
```

Handling large result collections for the remote interface query

If you intend your query to return a large collection, you have the option of programming it to return results in multiple smaller, more manageable quantities. Use the skipRow and maxRow parameters on the remote executeQuery method to retrieve the answer in chunks. For example:

```
int skipRow=0;
int maxRow=100;
QueryIterator it = null;
do {
 it = qb.executeQuery(query, null, null ,skipRow, maxRow );
 while (it.hasNext() ) {
 // display result
 skipRow = skipRow + maxRow;
}
} while ( ! it.isComplete() ) ;
```

Example: Using the local interface for Dynamic query

When you run a dynamic Enterprise JavaBeans (EJB) query using the local interface, you are calling the executeQuery method on the QueryLocal interface. This interface does not initiate a transaction for the method; therefore you must explicitly establish a transaction context for the query to run.

Note: To establish a transaction context, the following example calls the begin() and commit() methods. An alternative to using these methods is simply embedding your query code within an EJB method that runs within a transaction context.

Begin your query code with the following import statements:

```
import com.ibm.websphere.ejbquery.QueryLocalHome;
import com.ibm.websphere.ejbquery.QueryLocal;
import com.ibm.websphere.ejbquery.QueryLocalIterator;
import com.ibm.websphere.ejbquery.IQueryTuple;
import com.ibm.websphere.ejbquery.QueryException;
```

Next, write your query statement in the form of a string, as in the following example that retrieves the names and ejb-references for underpaid employees:

```
String query =
"select e.name, object(e) from EmpBean e where e.salary < 50000 ";
```

Create a QueryLocal object by obtaining a reference from the QueryLocalHome class. (This class defines the executeQuery method.) Note that in the following example, `ejb/query` is used as a local EJB reference pointing to the dynamic query JNDI name (`com/ibm/websphere/ejbquery/Query`):

```
InitialContext ic =  new InitialContext();
   QueryLocalHome  qh =  ( LocalQueryHome) ic.lookup( "java:comp/env/ejb/query" );
QueryLocal qb = qh.create();
```

The last portion of code initiates a transaction, calls the executeQuery method, and displays the query results. The QueryLocalIterator class is instantiated because it defines the query result set. This class is included in the Class QueryIterator API package. Keep in mind that the iterator loses validity at the end of the transaction; you must use the iterator in the same transaction scope as the executeQuery call.

```
userTransaction.begin();
QueryLocalIterator it = qb.executeQuery(query, null, null);
while (it.hasNext() ) {
 IQueryTuple tuple = (IQueryTuple) it.next();
 System.out.print( it.getFieldName(1) );
 String s = (String) tuple.getObject(1);
 System.out.println( s);
 System.out.println( it.getFieldName(2) );
 EmpLocal e = ( EmpLocal ) tuple.getObject(2);
 System.out.println( e.getPrimaryKey().toString());
}
userTransaction.commit();
```

In most situations, the QueryLocalIterator object is *demand-driven*. That is, it causes data to be returned incrementally: for each record retrieval from the database, the next() method must be called on the iterator. (Situations can exist in which the iterator is not demand-driven. For more information, consult the ″Local query interfaces″ subsection of the Dynamic query performance considerations topic.)

Because the full query result set materializes incrementally in the application server memory, you can easily control its size. During a test run, for example, you may decide that return of only a few tuples of the query result is necessary. In that case you should use a call of the close() method on the QueryLocalIterator object to close the query loop. Doing so frees SQL resources that the iterator uses. Otherwise, these resources are not freed until the full result set accumulates in memory, or the transaction ends.

Dynamic query performance considerations

While using a dynamic query can be convenient, there are times when it can have an impact on your application performance.

General performance considerations

Use of the following elements in your dynamic query can diminish application performance somewhat:

- Datatype converters and Java methods

Why: In general, query operations and predicates are translated into SQL so that the database server can perform them. If your query includes datatype converters (for EJB to RDB mapping, for example) or Java methods, however, the associated predicates and operations of your query must be performed in the memory of the application server.

- EJB methods and criteria that call for the return of EJB references

  Why: Queries that incorporate these elements trigger full activation of EJBs in the memory of the application server. (Returning a list of CMP fields from a query does not cause an EJB to be activated.)

When assessing application performance, you should also be aware that dynamic queries share connections with the persistence manager. Consequently, an application that includes a mixture of finder methods, CMR navigation, and dynamic queries relies on a single shared connection between the persistence manager and the dynamic query service to perform these tasks.

Limiting the return collection size

- **Remote interface queries**: The QueryIterator class of the remote interface mandates that all of your query results materialize in application server memory over the course of one method call. The SQL cursor(s) used to run the EJB query are closed upon completion of that call. Because this requirement poses a high risk for creating bottlenecks within the database server, you need to limit the size of any potentially large result collections.
- **Local interface queries**: In most situations, the QueryLocalIterator object behaves as a wrapper around an SQL cursor. It is *demand-driven*; it causes data to be returned incrementally. For each record retrieval from the database, the next() method must be called on the iterator.

  Use of certain operations in local interface queries, however, overrides the demand-driven behavior. In these cases, the query results fully materialize in memory just as do the result collections of remote interface queries. An example of such a case is:

  ```
  select  e.myBusinessMethod( ) from EmpBean e
   where e.salary < 50000 order by 1 desc
  ```

  This query requires performance of an EJB method to produce the final result collection. Consequently, the full dataset from the database must be returned in one collection to application server memory, where the EJB method can be run on the dataset in its entirety. For that reason, local interface query operations that invoke EJB methods are generally not demand-driven. You cannot control the return collection size for such queries.

  Because they *are* demand-driven, all other local interface queries allow you to control the size of return collections. You can use a call of the close() method on the QueryLocalIterator object to close the query loop after the desired number of return values has been fetched from the datastore. Otherwise, the SQL cursor(s) used to run the EJB query are not closed until the full result set accumulates in memory, or the transaction ends.

Access intent implications for dynamic query

WebSphere Application Server gives you the option to set access intent policies for your entity enterprise beans as a way of managing their transfer of data with the underlying data store. An access intent policy controls the isolation level used on the data source connection, as well as the database locks used during data retrieval. By manipulating these elements, you can maximize the efficiency of your application's data flow.

To learn more, begin with the topics "Access intent policies" on page 193 and "Concurrency control" on page 194.

When formulating dynamic queries, keep in mind the following considerations concerning their interaction with access intent policies:

- A dynamic query uses the first ASN name in the FROM clause to determine access intent.
- The collection increment attribute of an access intent policy is not used in processing a dynamic query.
- When performed on entity beans that have a pessimistic-Update access intent policy, your dynamic queries must return updateable collections. Therefore you need to formulate your query statements to

return only collections of entity beans, *not* collections of CMP fields. For example, the statement `select object(c) from Customer` is valid for a dynamic query performed under the constraint of a pessimistic-Update policy. The statement `select c.name from Customer c`, however, is not a valid dynamic query under this constraint.

- Using pessimistic-Update policy places restrictions on the types of query expressions. The restrictions depend on the back end database type and release. Refer to the topic “Access intent -- isolation levels and update locks” on page 992 for details.

Dynamic query API: prepareQuery() and executePlan() methods

Use these methods to more efficiently allocate the overhead associated with dynamic query. They are equivalent in function to the prepareStatement() and executeQuery() methods of the JDBC API.

To perform a dynamic Enterprise JavaBeans (EJB) query, the application server must parse the query string into structured query language (SQL) at run time. You can, of course, eliminate run-time overhead by choosing to perform a standard EJB query instead of a dynamic query. Sometimes referred to as *deployment queries*, standard queries are parsed and built at deployment, then performed by a finder or select method.

Another option is to write code that redistributes dynamic query overhead for better application performance. Begin by calling the prepareQuery() method in place of the executeQuery() method. The prepareQuery() method parses and translates your query, and returns a string called a *query plan*. The plan contains the SQL statement produced by parsing and translation, as well as other information needed by the dynamic query API. Save this string in your application and call the executePlan() method with the string to run your query. (You also might want to use the prepareQuery() method simply to see the SQL translation product; just call the method and display the return value.)

Pass the parameters of your query as an array of type Object on the prepareQuery() and the executePlan() method calls. Ensure that you pass appropriate data types, because the application server validates your query according to parameter type (rather than actual values) when it processes the prepareQuery() method call.

Example code

Note: In the example code that follows, the first executePlan() method call substitutes `parms[0]` for `?1`. Hence the first query performed is functionally equivalent to the following query statement:

```
select e.name as name, object(e) as emp from EmpBean e where e.salary < 50000
```

The second call runs a query that is functionally equivalent to this statement:

```
select e.name as name, object(e) as emp from EmpBean e where e.salary < 60000
```

The example:

```
String query =
"select e.name as name , object(e) as emp from EmpBean e where e.salary < ?1";
QueryIterator it = null;
Integer[] parms = new Integer[1];
parms[0] = new Integer(0);
```

In the call to `prepareQuery()`, pass any Integer value. Doing so defines `?1` as an Integer type, as in the following:

```
String queryPlan= qb.prepareQuery(query, parms, null );

   parms[0] = new Integer(50000);
```

Next you run the query with a real value of `Integer(50000)` for `?1`:

```
select e.name as name, object(e) as emp from EmpBean e where e.salary < 50000it =
 qb.executePlan( queryPlan, parms, 0, 99);

parms[0] = new Integer(60000);
```

Run the query again with a different value of `Integer(60000)` for `?1`:

```
it = qb.executePlan( queryPlan, parms, 0, 99);
```

Related tasks

"Using the dynamic query service" on page 1606
There are times in the development process when you might prefer to use the dynamic query service rather than the regular Enterprise JavaBean (EJB) query service (which can be referred to as *deployment query*). During testing, for instance, the dynamic query service can be used at application run time, so you do not have to re-deploy your application.

Dynamic and deployment EJB query services comparison

You can use the dynamic query service to build and execute queries against entity beans constructed dynamically at run time, rather than defining them at deployment time. By using dynamic query you gain the flexibility of queries defined at run time and utilize the power of Enterprise JavaBeans (EJB)-Query Language (QL). Apart from supporting all of the capabilities of an EJB-QL query, dynamic query adds functionality not available to standard static query. Two examples are the ability to select multiple data fields directly from the bean itself (static queries currently only allow one) and executing business methods directly in the query.

You can effectively create more efficient and less resource intensive applications with dynamic query. For example, two data fields are required from the results of a query. Because a standard EJB-QL query can only select one data field, it is necessary to select the entire EJB object and extract the needed data from the returned results through data access methods, possibly traversing Container Managed Relationships (CMR) boundaries in the process. However, when using dynamic query, you can get both pieces of data directly from the query without additional CMR traversal or accessor methods. This principle is the key to evaluating whether or not dynamic query can be used for performance gain. You should review the amount of data that must be retrieved, in addition to the amount of business logic needed to retrieve it, for example, CMR traversal or accessor methods.

Using parameters in the query rather than literal values is another performance consideration. Under most circumstances, it is better to define conditional values as parameters in the query and then pass those parameters through the appropriate mechanisms. By using this method, you have a greater chance of matching a cached query plan, and you eliminate the need to parse and build the plan from scratch. For example, ″SELECT Object(o) FROM schemaname AS o WHERE o.fieldname LIKE foo″, is more appropriately expressed as ″SELECT Object(o) FROM schemaname AS o WHERE o.fieldname LIKE ?1″ with the value *foo* passed as a parameter to the executeQuery method. The result is that any subsequent execution of a dynamic query structure that is the same, except for different string literal conditions, is registered as a plan cache hit (which delivers better ″observed″ performance).

When used as a direct replacement for an equivalent static query, dynamic query is approximately 25% slower than the static variation. This slowdown is due to the need for parsing and building a plan for the query, in addition to executing it. In the static variation, these costs are paid at deploy time. Despite this, the added functionality gained through the use of dynamic query, specifically the ability to select multiple data fields in a single query even across CMRs, creates opportunities to utilize dynamic query for the sake of performance improvement.

Internationalization

Task overview: Globalizing applications

An application that can present information to users according to regional cultural conventions is said to be *globalized*: The application can be configured to interact with users from different localities in culturally

appropriate ways. In a globalized application, a user in one region sees error messages, output, and interface elements in the requested language. Date and time formats, as well as currencies, are presented appropriately for users in the specified region. A user in another region sees output in the conventional language or format for that region. Globalization consists of two phases: *internationalization* (enabling an application component to use regional conventions) and *localization* (implementing a specific regional convention). This product supports globalization through the use of its localizable-text API and internationalization service.

- Make sure the server runtime environment is properly configured.

  For more information about supported locales and character encodings, see Working with locales and character encodings.
- Implement message catalogs in your application by using the localizable-text API.

  This product supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

  For more information about the localizable-text API, see “Task overview: Internationalizing interface strings (localizable-text API)” on page 1616.
- Implement more extensive locale support by using the internationalization service.

  With the internationalization service, you can manage the distribution of the internationalization information, or *internationalization context*, that is necessary to perform localizations within Java Platform, Enterprise Edition (Java EE) application components. Supported application components also include Web service client environments and Web service-enabled enterprise beans.

  For more information about the internationalization service, see “Task overview: Internationalizing application components (internationalization service)” on page 1627.

Globalization

An application that can present information to users according to regional cultural conventions is said to be *globalized*: The application can be configured to interact with users from different localities in culturally appropriate ways. In a globalized application, a user in one region sees error messages, output, and interface elements in the requested language. Date and time formats, as well as currencies, are presented appropriately for users in the specified region. A user in another region sees output in the conventional language or format for that region. Globalization consists of two phases: *internationalization* (enabling an application component to use regional conventions) and *localization* (implementing a specific regional convention).

Historically, the creation of globalized applications has been restricted to large corporations writing complex systems. However, given the rise in distributed computing and in the use of the World Wide Web, application developers are pressured to globalize a much wider variety of applications. This trend requires making globalization techniques much more accessible to application developers.

Internationalization of an application is driven by two variables, the time zone and the locale. The *time zone* indicates how to compute the local time as an offset from a standard time like Greenwich Mean Time. The *locale* is a collection of information about language, currency, and the conventions for presenting information like dates. A time zone can cover many locales, and a single locale can span time zones. With both time zone and locale, the date, time, currency, and language for users in a specific region can be determined.

By convention, a given locale is specified with a pair of codes (for language and region) that are governed by different standards. The ISO-639 standard governs the language code; the ISO-3166 standard governs the regional code. In notation, the two codes are typically joined by an underscore (_) character, for example, en_US for English in the United States. In Java code, locales are set and retrieved by means of the java.util.Locale class.

A first step: Localization of interface strings

In an application that is not globalized, the user interface is unalterably written into the application code. Internationalizing a user interface adds a layer of abstraction into the design of an application. The additional layer of abstraction enables you to localize the application for each locale that must be supported by the application.

In a localized application, the locale determines the message catalog from which the application retrieves message strings. Instead of printing an error message, the application represents the error message with some language-neutral information; in the simplest case, each error condition corresponds to a key. To print a usable error message, the application looks up the key in a *message catalog*. Each message catalog is a list of keys with associated strings. Different message catalogs provide strings for the different languages that are supported. The application looks up the key in the appropriate catalog, retrieves the corresponding error message in the requested language, and prints the string for the user.

Localization of text can be used for far more than translating error messages. For example, by using keys to represent each element in a graphical user interface (GUI) and by providing the appropriate message catalogs, the GUI (buttons, menus, and so on) can support multiple languages. Extending support to additional languages requires that you provide message catalogs for those languages; in many cases, the application needs no further modification.

The localizable-text package is a set of Java classes and interfaces that can be used to localize the strings in distributed applications easily. Language-specific string catalogs can be stored centrally so that they can be maintained efficiently.

Globalization challenges in distributed applications

With the advent of Internet-based business computational models, applications increasingly consist of clients and servers that operate in different geographical regions. These differences introduce the following challenges to the task of designing a solid client-server infrastructure:

Clients and servers can run on computers that have different endian architectures or code sets

Clients and servers can reside in computers that have different endian architectures: A client can reside in a little-endian CPU, while the server code runs in a big-endian one. A client might want to call a business method on a server running in a code set different from that of the client.

A client-server infrastructure must define precise endian and code-set tracking and conversion rules. The Java platform has nearly eliminated these problems in a unique way by relying on its Java virtual machine (JVM), which encodes all of the string data in UCS-2 format and externalizes everything in big-endian format. The JVM uses a set of platform-specific programs for interfacing with the native platform. These programs perform any necessary code set conversions between UCS-2 and the native code set of a platform.

Clients and servers can run on computers with different locale settings

Client and server processes can use different locale settings. For example, a Spanish client might call a business method upon an object that resides on an American English server. Some business methods are locale-sensitive in nature; for example, given a business method that returns a sorted list of strings, the Spanish client expects that list to be sorted according to the Spanish collating sequence, not in the English collating sequence of the server. Because data retrieval and sorting procedures run on the server, the locale of the client must be available to perform a legitimate sort.

A similar consideration applies in instances where the server has to return strings containing date, time, currency, exception messages, and so on, that are formatted according to the cultural expectations of the client.

Clients and servers can reside in different time zones

Client and server processes can run in different time zones. To date, all internationalization literature and resources concentrate mainly on code set and locale-related issues. They have generally ignored the time zone issue, even though business methods can be sensitive to time zone as well as to locale.

For example, suppose that a vendor makes the claim that orders received before 2:00 PM are processed by 5:00 PM the same day. The times given, of course, are in the time zone of the server that is processing the order. It is important to know the time zone of the client to give customers in other time zones the correct times for same-day processing.

Other time zone-sensitive operations include time stamping messages logged to a server, and accessing file or database resources. The concept of Daylight Savings Time further complicates the time zone issue.

Java Platform, Enterprise Edition (Java EE) provides support for application components that run on computers with differing endian architecture and code sets. It does not provide dedicated support for application components that run on computers with different locales or time zones.

The conventional method for solving locale and time zone mismatches across remote application components is to pass one or more extra parameters on all business methods needed to convey the client-side locale or time zone to the server. Although simple, this technique has the following limitations when used in Enterprise JavaBeans (EJB) applications:

- It is intrusive because it requires that one or more parameters be added to all bean methods in the call chain to locale-sensitive or time zone-sensitive methods.
- It is inherently error-prone.
- It is impracticable within applications that do not support modification, such as legacy applications.

The internationalization service addresses the challenges posed by locale and time zone mismatch without incurring the limitations of conventional techniques. The service systematically manages the distribution of internationalization contexts across the various components of EJB applications, including client applications, enterprise beans, and servlets. For more information, see “Task overview: Internationalizing application components (internationalization service)” on page 1627.

Language versions offered by this product

This product is offered in several languages, as enabled by the operating platform on which the product is installed.

For the z/OS platform, the following language versions are available:

- English
- Japanese

Globalization: Resources for learning

Use links in this topic to find relevant supplemental information about globalization. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and IBM Redbooks publications that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:

- “Programming instructions and examples” on page 1616
- “Programming specifications” on page 1616

Programming instructions and examples

- Java internationalization tutorial

  An online tutorial that explains how to use the Java SDK Internationalization API.
- Globalize your On Demand Business

  IBM's portal site for delivering globalized applications.

Programming specifications

- Java 2 Platform Standard Edition 5.0 Development Kit Documentation: Internationalization

  The Java internationalization documentation from Sun Microsystems, including a list of supported locales and encodings. For other versions of the Java platform, click the ″Internationalization Home Page″ link on that page.
- Java Specification Request 150, Internationalization Service for J2EE

  The specification of the Java internationalization service that was developed through the Java Community Process.
- W3C, Internationalization Core Working Group

  The W3C's Internationalization Core Working Group responsible for investigating the internationalization of Web services, in particular, the dependence of Web services on language, culture, region, and locale-related contexts.
- Making the WWW truly World Wide

  The W3C effort to make World Wide Web technology work with the many writing systems, languages, and cultural conventions of the global community:

Task overview: Internationalizing interface strings (localizable-text API)

This topic summarizes the steps involved in implementing message catalogs through the localizable-text API.

About this task

This product supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

1. Identify localizable text in your application.
2. Create the message catalogs that are necessary for the locales to be supported by your application.
3. In your application code, compose the language-specific strings for output.
4. Using an assembly tool, assemble your application code as one or more application components.
5. Prepare the localizable-text package for deployment with your localized application. In this step, you create a deployment Java archive (JAR) file.
6. Assemble the application modules and the deployment JAR file into a Java Platform, Enterprise Edition (Java EE) application.
7. Deploy and manage the application.

Results

Your application is deployed with localized text.

Identifying localizable text

The first step in localizing strings in an application component is identifying the best candidates for translation.

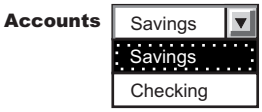
1. Determine which elements of the application need translating. Good candidates for localization include the following:
   - Graphical user interfaces: window titles, menus and menu items, buttons, on-screen instructions
   - Prompts in command-line interfaces

- Application output: messages and logs

2. Assign a unique key to each element for use in message catalogs for the application. The key provides a language-neutral link between the application and language-specific strings in the message catalogs. Establishing a naming convention for keys before creating the catalogs can make writing code with these keys much more intuitive for interface programmers.

Example

Suppose you are localizing the GUI for a banking system, and the first window contains a pull-down list to use for selecting a type of account.



The labels for the list and the account types in the list are good choices for localization. Three elements require keys: the list and two items in the list.

What to do next

Create message catalogs for the language-specific strings.

Creating message catalogs

Perform this task to begin the localization of strings in an application component.

Before you begin

Identify strings that need to be localized.

About this task

You can create a catalog as either a java.util.ResourceBundle subclass or a Java properties file. The properties-file approach is more common, because properties files can be prepared by people without programming experience and swapped without modifying the application code.

1. For each string that is identified for localization, add a line to the message catalog that lists the string key and value in the current language. In a properties file, each line has the following structure:

   ```
   key = string associated with the key
   ```

2. Save the catalog, giving it a locale-specific name. To enable resolution to a specific properties file, the Java API specifies naming conventions for the properties files in a resource bundle as *bundleName_localeID*.properties. Give the set of message catalogs a collective name, for example, BankingResources. For information about locale IDs that are recognized by the Java APIs, see ″Resources for learning.″

Example

The following English catalog (BankingResources_en.properties) supports the labels for the list and its two list items:

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
```

Do not create compound strings by concatenation (for example, combining the values of savingsString and accountString to form `Savings Accounts` in English. Success depends upon the grammar of the original language (in this case, English) and is not likely to extend to other languages.

The corresponding German catalog (BankingResources_de.properties) supports the labels as follows:

```
accountString = Konten
savingsString = Sparkonto
checkingString = Girokonto
```

What to do next

Write code to compose the language-specific strings.

Composing language-specific strings

Perform this task to complete the localization of strings in an application component.

Before you begin

Create message catalogs for the language-specific strings.

1. In application code, create a LocalizableTextFormatter instance, passing in required localization values.
2. Set other localization values as needed for more complex situations.
3. Generate a properly formatted, language-specific string.

What to do next

When the application is finished, deploy your application. For more information, see “Preparing the localizable-text package for deployment” on page 1625.

Localization API support

The com.ibm.websphere.i18n.localizabletext package contains classes and interfaces for localizing text.

This package makes extensive use of the internationalization features of the standard Java APIs from Sun Microsystems, including the following classes:

- java.util.Locale
- java.util.TimeZone
- java.util.ResourceBundle
- java.text.MessageFormat

For more information about the standard Java APIs, see “Globalization: Resources for learning” on page 1615.

The localizable-text package wraps the Java support and extends it for efficient and simple use in a distributed environment. The primary class used by application programmers is LocalizableTextFormatter. Instances of this class are usually created in server programs, but client programs can also create them. Formatter instances are created for specific resource-bundle names and keys. Client programs that receive a LocalizableTextFormatter instance call its format method. This method uses the locale of the client application to retrieve the appropriate resource bundle and compose a locale-specific message based on the key.

For example, suppose that a distributed application supports both French and English locales; the server is using an English locale and the client, a French locale. The server creates two resource bundles, one each for English and French. When the client makes a request that triggers a message, the server creates a LocalizableTextFormatter instance that contains the name of the resource bundle and the key for the message and passes the instance back to the client.

When the client receives the LocalizableTextFormatter instance, it calls the format method of the object. By using the locale and name of the resource bundle, the format method determines the name of the resource bundle that supports the French locale and retrieves the message that corresponds to the key from the French resource bundle. Formatting of the message is transparent to the client.

In this simple example, the resource bundles reside centrally with the server. They do not have to exist with the client. Part of what the localizable-text package provides is the infrastructure to support centralized catalogs. This implementation uses an enterprise bean (a stateless session bean provided with the localizable-text package) to access the message catalogs. When the client calls the format method on the LocalizableTextFormatter instance, the following events occur:

1. The client application sets the time-zone and locale values in the LocalizableTextFormatter instance, either by passing them explicitly or through default values.
2. A LocalizableTextFormatterEJBFinder call is made to retrieve a reference to the formatter bean.
3. Information from the LocalizableTextFormatter instance, including the time zone and locale of the client, is sent to the formatting bean.
4. The formatting bean uses the name of the resource bundle, the message key, the time zone, and the locale to compose a language-specific message.
5. The formatter bean returns the formatted message to the client.
6. The formatted message is inserted into the LocalizableTextFormatter instance and returned by the format method.

A call to the format method requires at most one remote call, to contact the formatter bean. As an alternative, the LocalizableTextFormatter instance can cache formatted messages, eliminating the remote call for subsequent uses. In addition, you can set a fallback string so that the application can return a readable string even if it cannot access the appropriate message catalog.

The resource bundles can be stored locally. The localizable-text package provides a static variable that indicates whether the bundles are stored locally (LocalizableConfiguration.LOCAL) or remotely (LocalizableConfiguration.REMOTE). However, the setting of this variable applies to all applications running within the same Java virtual machine.

LocalizableTextFormatter class

The LocalizableTextFormatter class, found in the com.ibm.websphere.i18n.localizabletext package, is the primary programming interface for using the localizable-text package. Instances of this class contain the information needed to create language-specific strings from keys and resource bundles.

The LocalizableTextFormatter class extends the java.lang.Object class and implements the following interfaces:

- java.io.Serializable
- com.ibm.websphere.i18n.localizabletext.LocalizableText
- com.ibm.websphere.i18n.localizabletext.LocalizableTextL
- com.ibm.websphere.i18n.localizabletext.LocalizableTextTZ
- com.ibm.websphere.i18n.localizabletext.LocalizableTextLTZ

Creation and initialization of class instances

The LocalizableTextFormatter class supports the following constructors:

- LocalizableTextFormatter()
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)

The LocalizableTextFormatter instance must have certain values, such as a resource-bundle name, a key, and the name of the formatting application. If you do not pass these values in by using the second constructor listed previously, you can set them separately by making the following calls:

- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)

You can use a fourth method, setArguments(Object[] args), to set optional localization values after construction. See “Processing of application-specific values” on page 1621 at the end of this topic. For a usage example, see “Composing complex strings” on page 1623.

API for formatting text

The formatting methods in the LocalizableTextFormatter class generate a string from a set of message keys and resource bundles, based on some combination of locale and time-zone values. Each method corresponds to one of the four localizable-text interfaces implemented. The following list indicates the interface in which each formatting method is defined:

- LocalizableText.format()
- LocalizableTextL.format(java.util.Locale locale)
- LocalizableTextTZ.format(java.util.TimeZone timeZone)
- LocalizableTextLTZ.format(java.util.Locale locale, java.util.TimeZone timeZone)

The format method with no arguments uses the locale and time-zone values set as defaults for the Java virtual machine. All four methods issue LocalizableException objects as needed.

Location of message catalogs and the appName value

Applications written with the localizable-text package can access message catalogs locally or remotely. In a distributed environment, the use of remote, centrally located message catalogs is appropriate. All clients can use the same catalogs, and maintenance of the catalogs is simplified. Local formatting is useful in test situations and appropriate under some circumstances. To support either local or remote formatting, a LocalizableTextFormatter instance must indicate the name of the formatting application.

For example, when an application formats a message by using remote catalogs, the message is actually formatted by an enterprise bean on the server. Although the localizable-text package contains the code to automate the lookup of the formatter bean and to issue a call to it, the application needs to know the name of the formatter bean. Several methods in the LocalizableTextFormatter class use a value described as *appName*, which refers to the name of the formatting application. It is not necessarily the name of the application in which the value is set.

Caching of messages

LocalizableTextFormatter instances can optionally cache formatted messages so that they do not require reformatting when needed again. By default, caching is not enabled, but you can use a `LocalizableTextFormatter.setCacheSetting(true)` call to enable caching. When caching is enabled and the format method is called, the method determines whether the message is already formatted. If so, the cached message is returned. If the message is not found in the cache, the message is formatted and returned to the caller, and a copy of the message is cached for future use.

If caching is disabled after messages are cached, those messages remain in the cache until the cache is cleared by a call to the LocalizableTextformatter.clearCache method. You can clear the cache at any time; the cache is automatically cleared when any of the following methods is called:

- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)
- setArguments(Object[] args)

API for providing fallback information

Under some circumstances, it can be impossible to format a message. The localizable-text package implements a fallback strategy, making it possible to get some information even if a message cannot be formatted correctly into the requested language. The LocalizableTextFormatter instance can optionally

store fallback values for a message string, the time zone, and the locale. These values can be ignored unless the LocalizableTextFormatter instance issues an exception. To set fallback values, call the following methods as appropriate:
- setFallBackString(String message)
- setFallBackLocale(Locale locale)
- setFallBackTimeZone(TimeZone timeZone)

For a usage example, see “Generating localized text” on page 1624.

Processing of application-specific values

The localizable-text package provides native support for localization based on time zone and locale, but you can construct messages on the basis of other values as well. If you need to consider variables other than locale and time zone in formatting localized text, write your own formatter class.

Your formatter class can extend the LocalizableTextFormatter class or independently implement some or all of the same localizable-text interfaces. As a minimum, your class must implement the java.io.Serializable interface and at least one of the localizable-text interfaces and its corresponding format method. If your class implements more than one localizable-text interface and format method, the order of evaluation of the interfaces is as follows:
1. LocalizableTextLTZ
2. LocalizableTextL
3. LocalizableTextTZ
4. LocalizableText

As an example, the localizable-text package provides a class that reports the time and date (LocalizableTextDateTimeArgument). In that class, date and time formatting is localized in accordance with three values: locale, time zone, and style.

Creating a formatter instance

Perform this task to set localization values for strings in an application component.

About this task

Server programs typically create LocalizableTextFormatter instances that are sent to clients as the result of some operation; clients format the objects at the appropriate time. Less typically, client programs create LocalizableTextFormatter objects locally.

1. If needed for your application, write your own formatter class. For more information about implementation, see “LocalizableTextFormatter class” on page 1619.
2. In application code, call the appropriate constructor for the formatter class and set required localization values. Some localization values, such as resource bundle name, key and formatting application, must be set, either through a constructor or soon after construction. Other localization values can be set only as needed. For more information about the API, see the related reference.

Example

The following code creates a LocalizableTextFormatter instance by using the default constructor and then sets the required localization values:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType) {
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ltf.setPatternKey("accountNumber");
```

```
   ltf.setResourceBundleName("BankingSample.BankingResources");
   ltf.setApplicationName("BankingSample");
   ...
}
```

The line of code in boldface exploits default behavior of the Java platform. By default, the Java platform looks first for a subclass of java.util.ResourceBundle called BankingResources. When none is found, the Java platform looks for a valid properties file of the same name. In this continuing example, a properties file is found.

The application that is requesting a localized message can specify the locale and time zone for message formatting, or the application can use the default values set for the Java virtual machine.

For example, a GUI can enable users to select the language in which to display the interface. A default value must be set initially so that the GUI can be created properly when the application first starts, but users can then change the locale for the GUI to suit their needs. The following code shows how to change the locale used by an application based on the selection of a menu item:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
...
import java.util.Locale;

public void actionPerformed(ActionEvent event) {
   String action = event.getActionCommand();
   ...
   if (action.equals("en_us")) {
      applicationLocale = new Locale("en", "US");
      ...
   }
   if (action.equals("de_de")) {
      applicationLocale = new Locale("de", "DE");
      ...
   }
   if (action.equals("fr_fr")) {
      applicationLocale = new Locale("fr", "FR");
      ...
   }
   ...
}
```

For more information, see ″Generating localized text.″

What to do next

Set optional localization values.

Setting optional localization values

In addition to setting localization values that are required by the LocalizableTextFormatter interface, you can set a number of optional values in application code, either through the constructor or by calling any of several methods for that purpose.

About this task

With optional values, you can do the following actions:

- Compose complex strings from variable substrings
- Customize the formatting of strings, considering variables other than time zone and locale

1. In application code, add the optional values into an array of type Object.

   ```
   Object[] arg = {new String(getAccountNumber())};
   ```

2. Pass the array into a LocalizableTextFormatter instance. You can pass the array through the appropriate constructor or call the setArguments(Object[]) method. For a usage example, see "Composing complex strings."

   Because the array is passed by value rather than by reference, any updates to the array variable after this point are not reflected in the LocalizableTextFormatter instance unless it is reset by calling the setArguments(Object[]) method.

What to do next

Write code to generate the localized text.

Composing complex strings:

Perform this task to insert variable substrings into a localized string.

Before you begin

Identify strings that need to be localized.

About this task

The localized-text package supports the substitution of variable substrings into a localized string that is retrieved from the message catalog by key.

1. In the message catalog, specify the location of the substitution in the string to be retrieved. Variable components are designated by braces (for example, `{0}`).
2. In application code, create a LocalizableTextFormatter instance, passing in an array that contains the variable value. If the variable substring must be localized, you can create a nested LocalizableTextFormatter instance and pass the instance in instead of a value.
3. Generate a localized string. When a format method is called on a formatter instance, the formatter takes each element of the array passed in the previous step and substitutes it for the placeholder with the matching index in the string that is retrieved from the message catalog. For example, the value at index 0 in the array replaces the `{0}` variable in the retrieved string.

Example

The following line from an English message catalog shows a string with a single substitution:

```
successfulTransaction = The operation on account {0} was successful.
```

The same key in message catalogs for other languages has a translation of this string with the variable at the appropriate location for each language.

The following code shows the creation of a single-element argument array and the creation and use of a LocalizableTextFormatter instance:

```
public void updateAccount(String transactionType) {
   ...
   Object[] arg = {new String(this.accountNumber)};
   ...
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter ("BankingResources",
                                    "successfulTransaction",
                                    "BankingSample",
                                    arg);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

Nesting formatter instances for localized substrings:

The ability to substitute variable substrings into the strings retrieved from message catalogs adds a level of flexibility to the localizable-text package, but this capability is of limited use unless the variable value can be localized. You can localize this value by nesting LocalizableTextFormatter instances.

Before you begin

Identify strings that need to be localized.

1. In the message catalog, add entries that correspond to potential values for the variable substring.
2. In application code, create a LocalizableTextFormatter instance for the variable substring, setting required localization values.
3. Create a LocalizableTextFormatter instance for the primary string, passing in an array that contains the formatter instance for the variable substring.

Example

The following line from an English message catalog shows a string entry with two substitutions and entries to support the localizable variable at index 0 (the second variable in the string, the account number, does not need to be localized):

```
successfulTransaction = The {0} operation on account {1} was successful.
depositOpString = deposit
withdrawOpString = withdrawal
```

The following code shows the creation of the nested formatter instance and its insertion (with the account number variable) into the primary formatter instance:

```
public void updateAccount(String transactionType) {
   ...
   // Successful deposit
   LocalizableTextFormatter opLTF =
      new LocalizableTextFormatter("BankingResources",
                                   "depositOpString",
                "BankingSample");
   Object[] args = {opLTF, new String(this.accountNumber)};
   ...
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter ("BankingResources",
                                    "successfulTransaction",
                                    "BankingSample",
                                    args);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

Generating localized text

Perform this task to specify the runtime formatting of localized text in an application component.

Before you begin

Create a formatter instance and set the localization values as needed.

1. If needed, customize the formatting behavior.
2. In application code, call the appropriate format method.

Example

You can provide fallback behavior for use if the appropriate message catalog is not available at formatting time.

The following code generates a localized string. If the formatting fails, the application retrieves and uses a fallback string instead of the localized string:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType){
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ...
   ltf.setFallBackString("Enter account number: ");
   try {
      msg = new Label(ltf.format(this.applicationLocale), Label.CENTER);
   }
   catch (LocalizableException le) {
      msg = new Label(ltf.getFallBackString(), Label.CENTER);
   }
   ...
}
```

What to do next

When the application is finished, deploy your application. For more information, see “Preparing the localizable-text package for deployment.”

Customizing the behavior of a formatting method:

Perform this task to change the runtime formatting of localized strings in an application component.

About this task

You can customize formatting behavior by passing your own formatter classes into a LocalizableTextFormatter instance through an array of optional values. This action enables you to consider variables other than locale and time zone when formatting localized text.

1. Write your own formatter class. For more information about implementation, see ″LocalizableTextFormatter class.″
2. In application code, create an instance of your formatter class as appropriate and pass it with any other optional localization values into an instance of LocalizableTextFormatter. When the LocalizableTextFormatter instance reads the instance that has been passed in, it attempts to call the format() method on the passed-in instance. The string returned is then processed with any other elements in the array.

Example

The localizable-text package provides an example of a user-defined class, called LocalizableTextDateTimeArgument. This class enables date and time information to be selectively formatted according to the style values defined in the java.text.DateFormat interface as well as the constants that are defined within the LocalizableTextDateTimeArgument class.

Preparing the localizable-text package for deployment

The LocalizableTextEJBDeploy tool is used to create a deployment Java Archive (JAR) file for the localizable text service. You must deploy the enterprise bean in each enterprise application that requires support for localized text.

Before you begin

Write code to compose the language-specific strings.

1. Make sure that the LocalizableTextEJBDeploy tool is included in the class path.

   Note: In versions 6.0.x and earlier, the LocalizableTextEJBDeploy tool used to reside in the file *app_server_root*/lib/ltext.jar. It now resides in the file *app_server_root*/plugins/com.ibm.ws.runtime_1.0.0.jar.
2. Set up a working directory for the LocalizableTextEJBDeploy tool to use. You need to pass this location to the tool through a command-line interface.
3. Run the LocalizableTextEJBDeploy tool. You might be asked if you want to regenerate deployment code for the LocalizableText bean. Do not redeploy the bean; if you do, an incorrect Java Naming and Directory Interface (JNDI) name will be generated.

   To deploy the bean on multiple hosts and servers, run the tool for each host and server combination. This action generates a unique JNDI name for each deployment. After the tool is run, a deployment JAR file is located in the working directory that you specified.

What to do next

Using an assembly tool, assemble the deployment JAR file in an enterprise application with other application components.

As part of preparing for deployment, perform the following:
- Add the resource bundles for your application to the Enterprise Archive (EAR) file as files.
- Add the location of the EAR file to the server class path for the server so that the resource bundles can be located on the virtual host and server.

The same deployment JAR file can be included in several enterprise applications.

LocalizableTextEJBDeploy command

This topic describes the command-line syntax for the LocalizableTextEJBDeploy tool.

Note: In versions 6.0.x and earlier, the LocalizableTextEJBDeploy tool used to reside in the file *app_server_root*/lib/ltext.jar. It now resides in the file *app_server_root*/plugins/com.ibm.ws.runtime_1.0.0.jar.

```
LocalizableTextEJBDeploy
   -a applicationName
   -h virtualHostName
   -i installationDirectory
   -s serverName
   -w workingDirectory
```

Parameters

The required parameters, which can be specified in any order, follow:

applicationName
The name of the formatting session bean. This name is used in LocalizableTextFormatter instances to specify where the actual formatting occurs. If the name cannot be resolved at run time, the format method issues an exception.

virtualHostName
The name of the virtual host on which the formatting session bean is deployed. This value is case-sensitive on all operating platforms.

installationDirectory
The location at which the application server product is installed.

serverName
The name of the application server. If this argument is not specified, the default server name for the product is used.

workingDirectory
A location for the tool to use temporarily.

Task overview: Internationalizing application components (internationalization service)

This topic summarizes the steps involved in using the internationalization service.

About this task

With the internationalization service, you can manage the distribution of the internationalization information, or *internationalization context*, that is necessary to perform localizations within Java Platform, Enterprise Edition (Java EE) application components. Supported application components also include Web service client environments and Web service-enabled enterprise beans.

1. Use the internationalization context API within application components to obtain or manage internationalization context.

   Servlet and enterprise bean business methods can use internationalization context to perform locale- and time zone-sensitive localizations. Enterprise JavaBeans (EJB) client applications, and server components that are configured to manage internationalization context must use the internationalization context API to set the context elements scoped to their invocations.

   You use the internationalization context API within Web service-enabled Java EE client programs and stateless session beans in the same manner that you would use conventional Java EE application components, with one exception. Internationalization context propagated over Web service requests contains a time zone ID, whereas conventional Remote Method Invocation/ Internet Inter-ORB Protocol (RMI/IIOP) requests propagate complete time zone information, including the raw offset, Daylight Savings Time information, and so on.

2. Assemble internationalized applications.

   The internationalization type specifies the internationalization policy that applies to a servlet or an enterprise bean and, in particular, indicates whether the application component or its hosting Java EE container manages internationalization context. Container internationalization attributes can be specified for container-managed servlet and enterprise bean business methods. These attributes tailor a policy by indicating which context the container scopes to an invocation. Configuring internationalization policies declaratively prescribes, by means of the application deployment descriptor, the distribution and management of context throughout an application.

   As you edit the deployment descriptor for assembly, you can also set the internationalization type and configure any container internationalization attributes for the servlets and enterprise beans in your application.

   You configure internationalization type and container internationalization attributes for Web service-enabled stateless session beans in the same manner as you do for conventional beans.

3. Manage the internationalization service.

   Use the administrative console to enable the service on all application servers.

   By default, the service is enabled within Java EE client environments but is disabled on application servers. You must enable the service on all application servers hosting your servlets and enterprise beans to use internationalization context.

4. Troubleshoot the internationalization service as needed.

   Use the administrative console to enable the trace service to log internationalization service messages when debugging your applications.

   The trace strings for the internationalization service follow; use both:

   ```
   com.ibm.ws.i18n.context.*=all=enabled:com.ibm.websphere.i18n.context.*=all=enabled
   ```

Internationalization service

In a distributed client-server environment, application processes can run on different machines, configured for different locales, corresponding to different cultural conventions; they can also be located across geographical boundaries. The internationalization service can help manage your application in a globally distributed environment.

For an understanding of how differences in locale impact application development, read “Globalization” on page 1613.

Java Platform, Enterprise Edition (Java EE) provides support for application components that run on computers with differing endian architecture and code sets. It does not provide dedicated support for application components that run on computers with different locales or time zones.

The internationalization service addresses the challenges posed by locale and time zone mismatch without incurring the limitations of conventional techniques. The service systematically manages the distribution of internationalization contexts across the various components of EJB applications, including client applications, enterprise beans, and servlets.

The service works by associating an internationalization context with every service request within an application. When a client-side component calls a business method, the internationalization service interposes by obtaining the internationalization context associated with the current client-side process and by attaching that context to the outgoing request. On the server side, the internationalization service again interposes by detaching the context from the incoming request and associating it with the server-side process on which the business method will run, effectively scoping the context to the business method. For HTTP requests, the caller context is constructed from the HTTP attributes and default values. The service propagates internationalization context on subsequent business method invocations in the same manner, which distributes the context of the originating request over the entire chain of business method invocations.

This basic operation of scoping and propagation is defined precisely by *internationalization context management policies*. Internationalization policies specify whether an application component or its hosting Java EE container are to manage internationalization context. For container-managed components, the policy indicates which internationalization context the container scopes to invocations on that component. Server components configured to manage internationalization context, as well as EJB clients, must use the internationalization context API to manage the internationalization context elements scoped to their invocations.

Every application component has a default policy, which can be overridden and tailored for servlets and enterprise beans at assembly time.

At run time, application components can use the internationalization context API to get any element of the internationalization contexts scoped to an invocation. To programmatically access context elements, application components first resolve an internationalization context API reference, then call the appropriate API method to access the various context elements, such as the caller locale or the invocation time zone. These elements can be used in calls to Java SDK internationalization API methods; for example, to perform localizations such as formatting messages, configuring dates, or comparing strings.

Assembling internationalized applications

Perform this task to configure application components for deployment with the internationalization service.

About this task

Use an assembly tool to configure internationalization in the deployment descriptors for servlets and enterprise beans.

1. Set the **internationalization type**.

All servlets and enterprise beans have an internationalization type setting that specifies whether internationalization context is managed by the application component or by its hosting Java Platform, Enterprise Edition (Java EE) container during invocations of their respective life cycle and business methods. The internationalization type can be configured for all server application components except entity beans, which are container-managed only.

By default, all server components use container-managed internationalization (CMI). The default setting suffices in most cases; when it does not, modify the internationalization type setting by completing the steps that are described in one of the following topics:
- "Setting the internationalization type for servlets"
- "Setting the internationalization type for enterprise beans" on page 1631

2. Set the **container internationalization attribute**.

   You can associate CMI servlets, and business methods of CMI enterprise beans, with a container internationalization attribute. That attribute specifies which of three internationalization contexts (**Caller**, **Server**, or **Specified**) the container is to scope to an invocation. When running as specified, the container internationalization attribute also specifies the custom internationalization context elements.

   Named container internationalization attributes can be associated with sets of servlets or with sets of Enterprise JavaBeans (EJB) business methods. Initially, CMI servlets and business methods implicitly run as caller and do not associate with a container internationalization attribute. When the implicit behavior or an associated attribute setting is unsuitable, configure an attribute by completing the steps that are described in one of the following topics:
   - "Configuring container internationalization for servlets" on page 1630
   - "Configuring container internationalization for enterprise beans" on page 1631

Setting the internationalization type for servlets

This task sets the internationalization type for a servlet within a Web module.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported a dynamic Web project.

1. In the Java EE perspective, open the Web project for which you want to set the internationalization type.
   a. Double-click **Dynamic Web Projects**.
   b. Double-click the name of the Web project to see its contents.
   c. Double-click the deployment descriptor object.

   The Web Deployment Descriptor panel is displayed.
2. In the Web Deployment Descriptor panel, click the Servlets tab.
3. Scroll down to **WebSphere Programming Model Extensions** and then **Internationalization**.
4. From the **Servlets and JSPs** list of the Servlets panel, select the servlet for which you want to set the internationalization type.
5. Under **Internationalization**, select a value from the **Internationalization type** list. Valid values are `Application` or `Container`.
6. From the menu bar, click **File > Save**.

Results

The internationalization type setting is assigned to the servlet.

What to do next

If you selected container-managed internationalization, you can then set container-managed internationalization attributes for methods within the servlet. For more information, see ″Configuring container internationalization for servlets.″

Configuring container internationalization for servlets

This task configures container internationalization for a servlet within a Web module.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported a dynamic Web project.

You must also have set the internationalization type of one or more servlets in a Web project to `Container`.

About this task

This procedure relates one or more servlets to a container-managed internationalization attribute.

1. In the Java EE perspective, open the Web project for which you want to configure container internationalization.
   a. Double-click **Dynamic Web Projects**.
   b. Double-click the name of the Web project to see its contents.
   c. Double-click the deployment descriptor object.

   The Web Deployment Descriptor panel is displayed.
2. In the Web Deployment Descriptor panel, click the Servlets tab.
3. Scroll down to **WebSphere Programming Model Extensions** and then **Internationalization**.
4. Following **Container-managed Internationalization Attribute**, set the **Run As** field by selecting `Caller`, `Server`, or `Specified`.
5. If the servlet is to be run as Specified, select an internationalization policy from the **Specified** list or define a new policy.
   a. To define an internationalization policy, click **New**. The New Specified Initialization wizard is displayed.
   b. In the **Description** field, give the policy a name.
   c. If needed, set a time zone ID and add a time zone description. If you do not find the appropriate time zone in the ID list, click **Customize** to define one relative to Greenwich Mean Time (GMT).
   d. Create at least one locale for the policy. To create a locale, click **Add**; select a language and (optional) geographic region; specify a variant as needed. Add a locale description and click **OK** to finish. The new locale is added to the **Locales** list.
   e. If more than one locale is defined for the policy, select a locale from the **Locales** list and click **Finish**. Otherwise, just click **Finish**
6. From the menu bar, click **File > Save**.

Results

Selected servlets are now configured to run under the associated internationalization settings.

Setting the internationalization type for enterprise beans

This task sets the internationalization type for an enterprise bean within an Enterprise JavaBeans (EJB) module.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported an EJB project.

About this task

Container-managed internationalization (CMI) is the default type; entity beans cannot be set to application-managed internationalization (AMI). Use CMI also for stateless session beans that are enabled for Web services.

1. In the Java EE perspective, open the EJB project for which you want to set the internationalization type.
   a. Double-click **EJB Projects**.
   b. Double-click the name of the EJB project to see its contents.
   c. Double-click the deployment descriptor object.

   The EJB Deployment Descriptor panel is displayed.
2. In the EJB Deployment Descriptor panel, click the Internationalization tab. Any enterprise beans that are already configured for AMI are displayed in the **Internationalization type** list.
3. To set the internationalization type for any other enterprise beans to AMI, click **Add** following the **Internationalization type** list. The Internationalization Type wizard opens. Only message-driven or session beans can be selected.
4. Select the beans that you want to set and click **Finish** to exit the wizard.
5. From the menu bar, click **File > Save**.

Results

The internationalization type is assigned to the bean.

What to do next

For beans that use container-managed internationalization, you can then set container-managed internationalization attributes. For more information, see ″Configuring container internationalization for enterprise beans.″

Configuring container internationalization for enterprise beans

This task configures container internationalization for enterprise bean business methods.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported an EJB project.

You must also have one or more enterprise beans set to container-managed internationalization (CMI) by default.

About this task

This procedure relates one or more business methods to one or more container-managed internationalization (CMI) attributes. Use this procedure also for stateless session beans that are enabled for Web services.

1. In the Java EE perspective, open the EJB project for which you want to configure container internationalization.
   a. Double-click **EJB Projects**.
   b. Double-click the name of the EJB project to see its contents.
   c. Double-click the deployment descriptor object.

   The EJB Deployment Descriptor panel is displayed.
2. In the EJB Deployment Descriptor panel, click the Internationalization tab. Any business methods that are already configured are displayed in the **Internationalization attributes** list.
3. To configure a CMI business method, click **Add** following the **Internationalization attributes** list. The Internationalization Attributes wizard opens.
4. Set the **Run As** field by selecting `Caller`, `Server`, or `Specified`. Add a meaningful description. As a group, the CMI attribute settings comprise an internationalization policy.
   - The description appears as `Internationalization` *`description`* (*`runAsSetting`*) in the **Internationalization attributes** list when you are finished.
   - If you do not provide a description, the policy name appears as `Internationalization` (*`runAsSetting`*).

   If the bean is to be run as Specified, complete the following steps to specify the context elements that the container scopes to bean method invocations.
   a. Set a time zone ID and add a time zone description as needed. If you do not find the appropriate time zone in the ID list, click **Custom** to define one relative to Greenwich Mean Time (GMT).
   b. Set a locale. Select a language and (optional) geographic region; specify a variant as needed. Add a locale description as needed and click **OK** to finish.
5. Click **Next**.
6. Select the beans for which you want to configure method-level internationalization attributes and click **Next**.
7. Select the methods that you want to configure and click **Next**. A check box is displayed next to each method name that you select.
   - Click **Apply to All** to place a check box next to the displayed bean name.
   - Click **Select Beans** to select more beans with CMI.
8. Click **Finish** to exit the wizard.
9. From the menu bar, click **File > Save**.

Results

The bean methods are now configured to run under the associated internationalization settings.

Using the internationalization context API

Enterprise JavaBeans (EJB) client applications, servlets, and enterprise beans can programmatically obtain and manage internationalization context using the internationalization context API. For Web service client applications, you use the API to obtain and manage internationalization context in the same manner as for EJB clients.

Before you begin

The java.util and com.ibm.websphere.i18n.context packages contain all of the classes necessary to use the internationalization service within an EJB application.

1. Gain access to the internationalization context API.

   Resolve internationalization context API references once over the life cycle of an application component, within the initialization method of that component (for example, within the init method of servlets, or within the SetXxxContext method of enterprise beans). For Web service client programs, resolve a reference to the internationalization context API during initialization. For stateless session beans enabled for Web services, resolve the reference in the setSessionContext method.
2. Access caller locales and time zones.

   Every remote invocation of an application component has an associated caller internationalization context associated with the thread that is running that invocation. A caller context is propagated by the internationalization service and middleware to the target of a request, such as an Enterprise JavaBeans (EJB) business method or servlet service method. This task also applies to Web service client programs.
3. Access invocation locales and time zones.

   Every remote invocation of a servlet service or Enterprise JavaBeans (EJB) business method has an invocation internationalization context associated with the thread that is running that invocation. Invocation context is the internationalization context under which servlet and business method implementations run; it is propagated on subsequent invocations by the internationalization service and middleware. This task also applies to Web service client programs.

Results

The resulting components are said to use *application-managed internationalization* (AMI). For more information about AMI, see “Internationalization context: Management policies” on page 1648.

Example

Each supported application component uses the internationalization context API differently. Three code examples are provided that illustrate how to use the API within each component type. Differences in API usage, as well as other coding tips, are noted in comments that precede the relevant statement blocks.

Gaining access to the internationalization context API

Perform this task to access the internationalization service by resolving a reference to the internationalization context API.

About this task

Resolve internationalization context API references once over the life cycle of an application component, within the initialization method of that component (for example, within the init method of servlets, or within the SetXxxContext method of enterprise beans). For Web service client programs, resolve a reference to the internationalization context API during initialization. For stateless session beans enabled for Web services, resolve the reference in the setSessionContext method.

1. Resolve a reference to the UserInternationalization interface by performing a lookup on the Java Naming and Directory Interface (JNDI) name `java:comp/websphere/UserInternationalization`. For example:

```
//-------------------------------------------------------------------
// Internationalization context imports.
//-------------------------------------------------------------------
import com.ibm.websphere.i18n.context.*;
import javax.naming.*;
...

public class MyApplication {
  ...

  //-------------------------------------------------------------------
  // Resolve a reference to the UserInternationalization interface.
  //-------------------------------------------------------------------
  InitialContext initCtx = null;
  UserInternationalization userI18n = null;
  final String UserI18nUrl = "java:comp/websphere/UserInternationalization";
  try {
    initCtx = new InitialContext();
    userI18n = (UserInternationalization)initCtx.lookup(UserI18nUrl);
  }
  catch (NamingException ne) {
    // UserInternationalization URL is unavailable.
  }
```

If the UserInternationalization object is unavailable because of an anomaly or a restriction, the JNDI lookup invocation issues a javax.naming.NameNotFoundException exception that contains the java.lang.IllegalStateException instance.

2. Use the UserInternationalization reference to create references to the CallerInternationalization or InvocationInternationalization objects, which provide access to elements of the Caller or Invocation internationalization contexts, respectively. The CallerInternationalization reference can be bound to the Internationalization interface only; the InvocationInternationalization reference can be bound to either the Internationalization or the InvocationInternationalization interfaces, depending on whether the application requires read-only or read-write access to the invocation context. For example:

```
  ...
  //-------------------------------------------------------------------
  // Resolve references to the Internationalization and
  // InvocationInternationalization interfaces.
  //-------------------------------------------------------------------
  Internationalization callerI18n = null;
  InvocationInternationalization invocationI18n = null;
  try {
    callerI18n = userI18n.getCallerInternationalization();
    invocationI18n = userI18n.getInvocationInternationalization();
  }
  catch (IllegalStateException ise) {
    // An Internationalization interface(s) is unavailable.
  }
```

Accessing caller locales and time zones

Perform this task to access elements of the caller internationalization context.

Before you begin

An application component must first resolve a reference to the CallerInternationalization object and then bind it to the Internationalization interface.

About this task

Every remote invocation of an application component has an associated caller internationalization context associated with the thread that is running that invocation. A caller context is propagated by the internationalization service and middleware to the target of a request, such as an Enterprise JavaBeans (EJB) business method or servlet service method. This task also applies to Web service client programs.

1. Obtain the desired caller context elements.

```
java.util.Locale [] myLocales  = null;
try {
  myLocales  = callerI18n.getLocales();
}
catch (IllegalStateException ise) {
  // The Caller context is unavailable;
  // is the service started and enabled?
}
...
```

   The Internationalization interface contains the following methods to get caller internationalization context elements:
   - **Locale [] getLocales()** Returns the list of caller locales that are associated with the current thread.
   - **Locale getLocale()** Returns the first in the list of caller locales that are associated with the current thread.
   - **TimeZone getTimeZone()** Returns the SimpleTimeZone caller that is associated with the current thread.

   The Internationalization interface supports read-only access to internationalization context within application components. Methods of the Internationalization interface are available to all EJB application components and are used in the same manner for each, but the method semantics vary according to the component type. For instance, when obtaining the caller locale within an EJB client application, the interface returns the default locale of the host Java virtual machine (JVM); in contrast, when obtaining caller context within a servlet service method (for example, doPost or doGet methods), the interface returns the first locale (accept-language) propagated within the corresponding HTML request. See Internationalization context for a discussion of how the service propagates internationalization context throughout an application.
2. Use the caller context elements to localize computations under a locale or time zone of the calling process.

```
DateFormat df = DateFormat.getDateInstance(myLocale);
String localizedDate = df.getDateInstance().format(aDateInstance);
...
```

Accessing invocation locales and time zones

Perform this task to access elements of the invocation internationalization context.

Before you begin

An application component must first resolve a reference to the InvocationInternationalization object and then bind it to the InvocationInternationalization interface of the internationalization context API.

About this task

Every remote invocation of a servlet service or Enterprise JavaBeans (EJB) business method has an invocation internationalization context associated with the thread that is running that invocation. Invocation context is the internationalization context under which servlet and business method implementations run; it is propagated on subsequent invocations by the internationalization service and middleware. This task also applies to Web service client programs.

1. Obtain the desired invocation context elements.

```
java.util.Locale myLocale;
try {
  myLocale = invocationI18n.getLocale();
}
catch (IllegalStateException ise) {
  // The invocation context is unavailable;
  // is the service started and enabled?
}
...
```

The InvocationInternationalization interface contains the following methods to both get and set invocation internationalization context elements:

- **Locale [] getLocales()**. Returns the list of invocation locales that is associated with the current thread.
- **Locale getLocale()**. Returns the first in the list of invocation locales that is associated with the current thread.
- **TimeZone getTimeZone()**. Returns the SimpleTimeZone invocation that is associated with the current thread.
- **setLocales(Locale [])**. Sets the list of invocation locales that are associated with the current thread to the supplied list.
- **setLocale(Locale)**. Sets the list of invocation locales that are associated with the current thread to a list that contains the supplied locale.
- **setTimeZone(TimeZone)**. Sets the invocation time zone that is associated with the current thread to the supplied SimpleTimeZone.
- **setTimeZone(String)**. Sets the invocation time zone that is associated with the current thread to a SimpleTimeZone that has the supplied ID.

The InvocationInternationalization interface supports read and write access to invocation internationalization context within application components. However, according to internationalization context management policies, only components configured to manage internationalization context (application-managed internationalization, or AMI, components) have write access to invocation internationalization context elements. Calls to set invocation context elements within container-managed internationalization (CMI) application components result in a java.lang.IllegalStateException exception. Any differences in how application components can use InvocationInternationalization methods are explained in Internationalization context.

2. Use the invocation context elements to localize a computation under a locale or time zone of the calling process.

```
DateFormat df = DateFormat.getDateInstance(myLocale);
  String localizedDate = df.getDateInstance().format(aDateInstance);
  ...
```

Example

In the following code example, locale (en,GB) and simple time zone (GMT) transparently propagate on the call to the myBusinessMethod method. Server-side application components, such as `myEjb`, can use the InvocationInternationalization interface to obtain these context elements.

```
...
//--------------------------------------------------------------------
// Set the invocation context under which the business method or
// servlet will run and propagate on subsequent remote business
// method invocations.
//--------------------------------------------------------------------
try {
  invocationI18n.setLocale(new Locale("en", "GB"));
  invocationI18n.setTimeZone(SimpleTimeZone.getTimeZone("GMT"));
}
catch (IllegalStateException ise) {
  // Is the component CMI; is the service started and enabled?
}
myEjb.myBusinessMethod();
```

Within CMI application components, the Internationalization and InvocationInternationalization interfaces are semantically equivalent. You can use either of these interfaces to obtain the context associated with the thread on which that component is running. For instance, both interfaces can be used to obtain the list of locales propagated to the servlet doPost service method.

Example: Managing internationalization context in an EJB client program

Enterprise JavaBeans (EJB) client applications, Web service client applications, and enterprise beans programmatically obtain and manage internationalization context by using the internationalization context API (com.ibm.websphere.i18n.context).

The following code example illustrates how to use the internationalization context API within a contained EJB client program or Web service client program.

```
//----------------------------------------
// Basic Example: J2EE EJB client.
//----------------------------------------
package examples.basic;

//----------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//----------------------------------------
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;
import java.util.SimpleTimeZone;

public class EjbClient {

  public static void main(String args[]) {

    //------------------------------------------------
    // INTERNATIONALIZATION SERVICE: API references.
    //------------------------------------------------
    UserInternationalization userI18n = null;
    Internationalization callerI18n = null;
    InvocationInternationalization invocationI18n = null;

    //------------------------------------------------
    // INTERNATIONALIZATION SERVICE: JNDI name.
    //------------------------------------------------
    final String UserI18NUrl =
        "java:comp/websphere/UserInternationalization";

    //------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Resolve the API.
    //------------------------------------------------
    try {
     Context initialContext = new InitialContext();
     userI18n = (UserInternationalization)initialContext.lookup(
         UserI18NUrl);
     callerI18n = userI18n.getCallerInternationalization();
     invI18n = userI18n.getInvocationInternationalization ();
    } catch (NamingException ne) {
      log("Error: Cannot resolve UserInternationalization: Exception: " + ne);
    } catch (IllegalStateException ise) {
      log("Error: UserInternationalization is not available: " + ise);
    }
    ...

    //------------------------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Set invocation context.
    //
    // Under Application-managed Internationalization (AMI), contained EJB
    // client programs may set invocation context elements. The following
    // statements associate the supplied invocation locale and time zone
    // with the current thread. Subsequent remote bean method calls will
```

```
    // propagate these context elements.
    //---------------------------------------------------------------------
    try {
      invocationI18n.setLocale(new Locale("fr", "FR", ""));
      invocationI18n.setTimeZone("ECT");
    } catch (IllegalStateException ise) {
      log("An anomaly occurred accessing Invocation context: " + ise );
    }
    ...

    //---------------------------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Get locale and time zone.
    //
    // Under AMI, contained EJB client programs can get caller and
    // invocation context elements associated with the current thread.
    // The next four statements return the invocation locale and time zone
    // associated above, and the caller locale and time zone associated
    // internally by the service. Getting a caller context element within
    // a contained client results in the default element of the JVM.
    //---------------------------------------------------------------------
    Locale invocationLocale = null;
    SimpleTimeZone invocationTimeZone = null;
    Locale callerLocale = null;
    SimpleTimeZone callerTimeZone = null;
    try {
      invocationLocale = invocationI18n.getLocale();
      invocationTimeZone =
          (SimpleTimeZone)invocationI18n.getTimeZone();
      callerLocale = callerI18n.getLocale();
      callerTimeZone = SimpleTimeZone)callerI18n.getTimeZone();
    } catch (IllegalStateException ise) {
      log("An anomaly occurred accessing I18n context: " + ise );
    }

    ...
  } // main

  ...
  void log(String s) {
    System.out.println (((s == null) ? "null" : s));
  }
} // EjbClient
```

Example: Managing internationalization context in a servlet

Servlets programmatically obtain and manage internationalization context by using the internationalization context API (com.ibm.websphere.i18n.context).

The following code example illustrates how to use the internationalization context API within a servlet. Note comments in the init and doPost methods.

```
...
//---------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//---------------------------------------------------------------------
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;

public class J2eeServlet extends HttpServlet {

  ...
```

```
//-----------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: API references.
//-----------------------------------------------------------------
protected UserInternationalization userI18n = null;
protected Internationalization i18n = null;
protected InvocationInternationalization invI18n = null;

//-------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: JNDI name.
//-------------------------------------------------------------
public static final String UserI18NUrl =
    "java:comp/websphere/UserInternationalization";

protected Locale callerLocale = null;
protected Locale invocationLocale = null;

/**
 * Initialize this servlet.
 * Resolve references to the JNDI initial context and the
 * internationalization context API.
 */
public void init() throws ServletException {

  //-----------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Resolve API.
  //
  // Under Container-managed Internationalization (CMI), servlets have
  // read-only access to invocation context elements. Attempts to set these
  // elements result in an IllegalStateException.
  //
  // Suggestion: cache all internationalization context API references
  // once, during initialization, and use them throughout the servlet
  // lifecycle.
  //-----------------------------------------------------------------
  try {
    Context initialContext = new InitialContext();
    userI18n = (UserInternationalization)initialContext.lookup(UserI18nUrl);
    callerI18n = userI18n.getCallerInternationalization();
    invI18n = userI18n.getInvocationInternationalization();
  } catch (NamingException ne) {
    throw new ServletException("Cannot resolve UserInternationalization" + ne);
  } catch (IllegalStateException ise) {
    throw new ServletException ("Error: UserInternationalization is not
       available: " + ise);
  }
  ...
} // init

/**
 * Process incoming HTTP GET requests.
 * @param request Object that encapsulates the request to the servlet
 * @param response Object that encapsulates the response from the
 *    Servlet.
 */
public void doGet(
    HttpServletRequest  request,
    HttpServletResponse response)
  throws ServletException, IOException {
  doPost(request, response);
} // doGet

/**
 * Process incoming HTTP POST requests
 * @param request Object that encapsulates the request to
 *    the Servlet.
 * @param response Object that encapsulates the response from
 *    the Servlet.
```

```
   */
  public void doPost(
      HttpServletRequest  request,
      HttpServletResponse response)
    throws ServletException, IOException {

    ...
    //-------------------------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Get caller context.
    //
    // The internationalization service extracts the accept-languages
    // propagated in the HTTP request and associates them with the
    // current thread as a list of locales within the caller context.
    // These locales are accessible within HTTP Servlet service methods
    // using the caller internationalization object.
    //
    // If the incoming HTTP request does not contain accept languages,
    // the service associates the server's default locale. The service
    // always associates the GMT time zone.
    //
    //-------------------------------------------------------------------
    try {
      callerLocale = callerI18n.getLocale();   // caller locale
      // the following code enables you to get invocation locale,
      // which depends on the Internationalization policies.
      invocationLocale = invI18n.getLocale();  // invocation locale
    } catch (IllegalStateException ise) {
      log("An anomaly occurred accessing Invocation context: " + ise);
    }
    // NOTE: Browsers may propagate accept-languages that contain a
    // language code, but lack a country code, like "fr" to indicate
    // "French as spoken in France."  The following code supplies a
    // default country code in such cases.
    if (callerLocale.getCountry().equals(""))
      callerLocale = AccInfoJBean.getCompleteLocale(callerLocale);

    // Use iLocale in JDK locale-sensitive operations, etc.
    ...
  } // doPost

  ...
  void log(String s) {
    System.out.println (((s == null) ? "null" : s));
  }
} // CLASS J2eeServlet
```

Example: Managing internationalization context in a session bean

This code example illustrates how to perform a localized operation using the internationalization service within a session bean or Web service-enabled session bean.

```
...
//-------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//-------------------------------------------------------------
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;

/**
 * This is a stateless Session Bean Class
 */
public class J2EESessionBean implements SessionBean {
```

```
//-------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: API references.
//-------------------------------------------------------------
protected UserInternationalization        userI18n = null;
protected InvocationInternationalization  invI18n  = null;

//-------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: JNDI name.
//-------------------------------------------------------------
public static final String UserI18NUrl =
    "java:comp/websphere/UserInternationalization";
...

/**
 * Obtain the appropriate internationalization interface
 * reference in this method.
 * @param ctx javax.ejb.SessionContext
 */
public void setSessionContext(javax.ejb.SessionContext ctx) {

  //-------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Resolve the API.
  //-------------------------------------------------------------
  try {
    Context initialContext = new InitialContext();
    userI18n = (UserInternationalization)initialContext.lookup(
        UserI18NUrl);
    invI18n = userI18n.getInvocationInternationalization();
  } catch (NamingException ne) {
    log("Error: Cannot resolve UserInternationalization: Exception: " + ne);

  } catch (IllegalStateException ise) {
    log("Error: UserInternationalization is not available: " + ise);
  }
} // setSessionContext

/**
 * Set up resource bundle using I18n Service
 */
public void setResourceBundle()
{
  Locale invLocale   = null;

  //-------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Get invocation context.
  //-------------------------------------------------------------
  try {
    invLocale = invI18n.getLocale();
  } catch (IllegalStateException ise) {
    log ("An anomaly occurred while accessing Invocation context: " + ise );
  }
  try {
    Resources.setResourceBundle(invLocale);
    // Class Resources provides support for retrieving messages from
    // the resource bundle(s). See Currency Exchange sample source code.
  } catch (Exception e) {
    log("Error: Exception occurred while setting resource bundle: " + e);
  }
} // setResourceBundle

/**
 * Pass message keys to get the localized texts
 * @return java.lang.String []
 * @param key java.lang.String []
 */
public String[] getMsgs(String[] key) {
```

```
    setResourceBundle();
    return Resources.getMsgs(key);
  }

  ...
  void log(String s) {
    System.out.println(((s == null) ? ";null" : s));
  }
} // CLASS J2EESessionBean
```

Internationalization context API: Programming reference

Application components programmatically manage internationalization context through the UserInternationalization, Internationalization, and InvocationInternationalization interfaces in the com.ibm.websphere.i18n.context package.

The following code example introduces the internationalization context API:

```
public interface UserInternationalization {
  public Internationalization getCallerInternationalization();
  public InvocationInternationalization
  getInvocationInternationalization();
}

public interface Internationalization {
  public java.util.Locale[] getLocales();
  public java.util.Locale getLocale();
  public java.util.TimeZone getTimeZone();
}

public interface InvocationInternationalization
    extends Internationalization {
  public void setLocales(java.util.Locale[] locales);
  public void setLocale(java.util.Locale jmLocale);
  public void setTimeZone(java.util.TimeZonetimeZone);
  public void setTimeZone(String timeZoneId);
}
```

UserInternationalization interface

The UserInternationalization interface provides factory methods for obtaining references to the CallerInternationalization and InvocationInternationalization context objects. Use these references to access elements of the caller and invocation contexts correlated to the current thread.

Methods of the UserInternationalization interface:

Internationalization getCallerInternationalization()
: Returns a reference implementing the Internationalization interface that supports access to elements of the caller internationalization context correlated to the current thread. If the service is disabled, this method issues an IllegalStateException exception.

InvocationInternationalization getInvocationInternationalization()
: Returns a reference implementing the InvocationInternationalization interface. If the service is disabled, this method issues an IllegalStateException exception.

Internationalization interface

The Internationalization interface declares methods that provide read-only access to internationalization context. Given a caller or invocation internationalization context object created with the UserInternationalization interface, bind the object to the Internationalization interface to get elements of that context type. Observe that caller internationalization context can be accessed only through this interface.

Methods of the Internationalization interface:

Locale[] getLocales()
Returns the chain of locales within the internationalization context (object) that is bound to the interface, provided the chain is not null; otherwise this method returns a chain of length(1) containing the default locale of the Java virtual machine (JVM).

Locale getLocale()
Returns the first in the chain of locales within the internationalization context (object) that is bound to the interface, provided the chain is not null; otherwise this method returns the default locale of the JVM.

TimeZone getTimeZone()
Returns the caller time zone (that is, the SimpleTimeZone instance) that is associated with the current thread, provided the time zone is non-null; otherwise this method returns the process time zone.

InvocationInternationalization interface

The InvocationInternationalization interface declares methods that provide read and write access to InvocationInternationalization context. Given an invocation internationalization context object created with the UserInternationalization interface, bind the object to the InvocationInternationalization interface to get and set elements of the invocation context.

According to the container-managed internationalization (CMI) policy, all set methods, setXxx(), issue an IllegalStateException exception when called within a CMI servlet or enterprise bean.

Methods of the InvocationInternationalization interface:

void setLocales(java.util.Locale[] locales)
Sets the chain of locales to the supplied chain, *locales*, within the invocation internationalization context. The supplied chain can be null or have length(>= 0). When the supplied chain is null or has length(0), the service sets the chain of invocation locales to an array of length(1) containing the default locale of the JVM. Null entries can exist within the supplied locale list, for which the service substitutes the default locale of the JVM on remote invocations.

void setLocale(java.util.Locale locale)
Sets the chain of locales within the invocation internationalization context to an array of length(1) containing the supplied locale, *locale*. The supplied locale can be null, in which case the service instead sets the chain to an array of length(1) containing the default locale of the JVM.

void setTimeZone(java.util.TimeZone timeZone)
Sets the time zone within the invocation internationalization context to the supplied time zone, *time zone*. If the supplied time zone is not an exact instance of java.util.SimpleTimeZone or is null, the service sets the invocation time zone to the default time zone of the JVM instead.

void setTimeZone(String timeZoneId)
Sets the time zone within the invocation internationalization context to the java.util.SimpleTimeZone having the supplied ID, *timeZoneId*. If the supplied time zone ID is null or invalid (that is, the ID is not displayed in the list of IDs returned by the java.util.TimeZone.getAvailableIds method) the service sets the invocation time zone to the simple time zone having an ID of GMT, an offset of 00:00, and otherwise invalid fields.

Internationalization context:

An *internationalization context* is a distributable collection of internationalization information containing an ordered list, or chain, of locales and a single time zone, where the locales and time zone are instances of the java.util.Locale and java.util.TimeZone Java SDK types, respectively. A locale chain is ordered according to the user's preference.

The internationalization service manages and makes available two varieties of internationalization context: the *caller context*, which represents the caller's localization environment, and the *invocation context*, which represents the localization environment under which a business method runs. Server application

components use elements of the caller and invocation internationalization contexts to appropriately tailor locale-sensitive and time zone-sensitive computations.

The internationalization service does not support time zone types other than the java.util.SimpleTimeZone type that is found in the Java SDK. Unsupported time zone types silently map to the default time zone of the JVM when supplied to internationalization context API methods. For a complete description of the java.util.Locale, java.util.TimeZone and java.util.SimpleTimeZone types, refer the Java SDK API documentation.

Caller context

Caller internationalization context contains the locale chain and time zone received on incoming EJB business method and servlet service method invocations; it is the internationalization context propagated from the calling process. Use caller context elements within server application components to localize computations to the calling component. Caller context is read-only and can be accessed by all application components by using the Internationalization interface of the internationalization context API.

Caller context is computed in the following manner: On an EJB business method or servlet service method invocation, the internationalization service extracts the internationalization context from the incoming request and scopes this context to the method as the caller context. For any missing or null context element, the service inserts the corresponding default element of the JVM (for example, java.util.Locale.getDefault() or java.util.TimeZone.getDefault().) The service performs a similar insertion whenever missing or null Caller context elements are encountered on invocations of stateless session beans that are enabled for Web services.

Formally, caller context is the invocation context of the calling business method or application component.

Invocation context

Invocation internationalization context contains the locale chain and time zone under which EJB business methods and servlet service methods run. It is managed by either the hosting container or the application component, depending on the applicable internationalization policy. On outgoing business method requests, it is the context that propagates to the target process. Use invocation context elements to localize computations under the specified settings of the current application component.

Invocation context is computed in the following manner: On an incoming business method or servlet service method invocation, the internationalization service queries the associated context management policy. If the policy is container-managed internationalization (CMI), the container scopes the context designated by the policy to the invocation; otherwise the policy is application-managed internationalization (AMI), and the container scopes an empty context to the invocation that can be altered by the method implementation.

Application components can access invocation context elements through both the Internationalization and InvocationInternationalization interfaces of the internationalization context API. Invocation context elements can be set (overwritten) under the application-managed internationalization policy only.

On an outgoing business method request, the service obtains the currently scoped invocation context and attaches it to the request. This outgoing exported context becomes the caller context of the target invocation. When supplying invocation context elements, either for export on outgoing requests or through the API, the internationalization service always provides the most recent element set using the API; the service also supplies the corresponding default element of the JVM for any null invocation context element.

Because the internationalization context that is propagated over Web services (SOAP) requests contains a time zone ID rather than the entire state of a java.lang.SimpleTimeZone object, time zone information

might be lost when a Web service-enabled client program or session bean becomes involved in remote business computation.

Internationalization context: Propagation and scope:

The scope of internationalization context is implicit. Every Enterprise JavaBeans (EJB) client application, servlet service method, and EJB business method call has two internationalization contexts under which it runs.

For each application component call, the container enters the caller context and the call context, as indicated by the pertinent internationalization policy, into scope before the container delegates to the actual implementation. When the implementation returns, the service removes these contexts from scope. The internationalization service supplies no programmatic mechanism for components to explicitly manage the scope of internationalization context.

The service scopes internationalization context differently with respect to application component type:
- "EJB client programs (contained)"
- "Servlets"
- "Enterprise beans" on page 1646
- "Web service client programs (contained)" on page 1646
- "Stateless session beans that are enabled for Web services" on page 1646

Internationalization context observes by-value semantics over remote method requests. Changes to internationalization context elements that are scoped to a call do not affect the corresponding elements of the internationalization context that is scoped to the remote calling process. Also, modifications to context elements obtained using the internationalization context API do not affect the corresponding elements that are scoped to the invocation.

EJB client programs (contained)

Before it calls the main method of a client program, the Java EE client container introduces into scope invocation and caller internationalization some contexts that contain null elements. These contexts remain in scope throughout the life of the program. EJB client programs are the base in a chain of remote business method invocations and, technically, do not have a logical caller context. Accessing a caller context element yields the corresponding default element of the client JVM. On outgoing EJB business method requests, the internationalization service propagates the invocation context to the target process. Any unset (null) invocation context elements are replaced with the default of the JVM when exported by the internationalization context API or by outgoing requests.

Note:

> To propagate values other than the JVM defaults to remote business methods, EJB client programs, as well as AMI servlets or enterprise beans, must set (override) elements of the invocation context. To learn how to set invocation context elements, see "Accessing invocation locales and time zones" on page 1635.

Servlets

On every servlet service method (doGet or doPost) invocation, the Java EE Web container introduces caller and invocation internationalization contexts into scope before delegating to the service method implementation. The caller context contains the accept-languages propagated in the HTTP servlet request, typically from a Web browser. The invocation context contains whichever context is indicated by the container internationalization attribute of the internationalization policy that is associated with the servlet. Any unset (null) invocation context elements are replaced with the default of the server JVM when exported by the internationalization context API or by outgoing requests. The caller and invocation contexts

remain effective until immediately after the implementation returns, at which time the container removes them from scope.

Enterprise beans

On every EJB business method invocation, the Java EE EJB container introduces caller and invocation internationalization contexts into scope before delegating to the business method implementation. The caller context contains the internationalization context elements imported from the incoming IIOP request; if the incoming request lacks a particular internationalization context element, the container scopes a null element. The invocation context contains whichever context is indicated by the container internationalization attribute of the internationalization policy that is associated with the business method.

On outgoing EJB business method requests, the service propagates the invocation context to the target process. Any unset (null) invocation context elements are replaced with the default of the server JVM when exported by the internationalization context API or by outgoing requests. The caller and invocation contexts remain effective until immediately after the implementation returns, when the container removes them from scope.

Consider a simple EJB application with a Java client that calls the remote myBeanMethod bean method. On the client side, the application can use the Internationalization Service API to set invocation context elements. When the client calls myBeanMethod(), the service exports the client invocation context to the outgoing request. On the server side, the service detaches the imported context from the incoming request and scopes it to the method as its caller context; the service also scopes the invocation context to the method as indicated by the associated internationalization context management policy. The EJB container then calls the myBeanMethod method, which can use the internationalization context API to access elements of either the caller or invocation contexts. When the myBeanMethod method returns, the EJB container removes these contexts from scope.

Web service client programs (contained)

Before it calls the main method of a Web service client program, the client container introduces into scope both invocation and caller internationalization contexts that contain null elements. These contexts remain in scope throughout the duration of the program. Web service client programs are the base in a chain of remote business method invocations and, technically, do not have a logical caller context. Accessing a Caller context element yields the corresponding default element of the client virtual machine.

On outgoing Web service requests, the internationalization service transparently creates a SOAP header block that contains the invocation context that is associated with the current thread; the SOAP representation of invocation context is propagated through the request to the target process. Any unset (that is, null) invocation context elements are replaced with the default element of the JVM when exported by the internationalization context API or by outgoing requests. Also, because the header contains only a time zone ID, the additional state of the time zone object (java.lang.SimpleTimeZone) of the invocation context might be lost, because it does not get propagated through the request.

Note:

> To propagate values other than the JVM defaults to remote business methods, Web service client programs, as well as AMI servlets or enterprise beans, must set (override) elements of the invocation context. For more information, see “Accessing invocation locales and time zones” on page 1635.

Stateless session beans that are enabled for Web services

On every method invocation of a Web service-enabled bean, the EJB container introduces caller and invocation internationalization contexts into scope before delegating control to the business method implementation. The caller context contains the internationalization context elements that are imported

from the SOAP header block of the incoming request. If the incoming request lacks a particular internationalization context element, the container introduces a null element into scope. The invocation context contains whichever context is indicated by the container internationalization attribute of the internationalization policy that is associated with the business method.

On outgoing EJB business method requests, the service propagates the invocation context to the target process. Any unset (that is, null) invocation context elements are replaced with the default element of the server JVM when exported by the internationalization context API or by outgoing requests. The caller and invocation contexts remain effective until immediately after control returns from the business method implementation, at which time the container removes them from scope.

On outgoing Web service requests, the internationalization service transparently creates a SOAP header block that contains the invocation context associated with the current thread. The SOAP representation of the invocation context is propagated through the request to the target process. Any unset (that is, null) invocation context elements are replaced with the default element of the JVM when exported by the internationalization context API or by outgoing requests.

Thread association considerations

The Web and EJB containers scope internationalization contexts to a method by associating the method with the thread that run the method implementation. Similarly, methods of the internationalization context API either associate context with, or obtain context associated with, the thread on which these methods run.

In cases where new threads are spawned within an application component (for instance, a user-generated thread inside the service method of a servlet, or a system-generated event handling thread in an AWT client) the internationalization contexts associated with the parent thread does not automatically transfer to the newly-spawned thread. In such instances, the service exports the default locale and time zone of the JVM on any remote business method request and on any API calls that run on the new thread.

If the default context is inappropriate, the desired invocation context elements must be explicitly associated to the new thread by using the setXxx methods of the InvocationInternationalization interface. Currently, internationalization context management policies enable invocation context to be set within EJB client programs, as well as within servlets, session beans, and message-driven beans that use application-managed internationalization.

Example: Representing internationalization context in a SOAP header:

This code example illustrates how internationalization context is represented within the SOAP header of a Web service request.

```
<InternationalizationContext>
   <Locales>
      <Locale>
         <LanguageCode>ja</LanguageCode>
         <CountryCode>JP</CountryCode>
         <VariantCode>Nihonbushi</VariantCode>
      </Locale>
      <Locale>
         <LanguageCode>fr</LanguageCode>
         <CountryCode>FR</CountryCode>
      </Locale>
      <Locale>
         <LanguageCode>en</LanguageCode>
         <CountryCode>US</CountryCode>
      </Locale>
   </Locales>
   <TimeZoneID>JST</TimeZoneID>
</InternationalizationContext>
```

This representation is valid against the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="InternationalizationContext"
                 type="InternationalizationContextType">
    </xsd:element>

    <xsd:complexType name="InternationalizationContextType">
        <xsd:sequence>
            <xsd:element name="Locales"
                         type="LocalesType">
            </xsd:element>
            <xsd:element name="TimeZoneID"
                         type="xsd:string">
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LocalesType">
        <xsd:sequence>
            <xsd:element name="Locale"
                         type="LocaleType"
                         minOccurs="0"
                         maxOccurs="unbounded">
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LocaleType">
        <xsd:sequence>
            <xsd:element name="LanguageCode"
                         type="xsd:string"
                         minOccurs="0"
                         maxOccurs="1">
            </xsd:element>
            <xsd:element name="CountryCode"
                         type="xsd:string"
                         minOccurs="0"
                         maxOccurs="1">
            </xsd:element>
            <xsd:element name="VariantCode"
                         type="xsd:string"
                         minOccurs="0"
                         maxOccurs="1">
            </xsd:element>
         </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Internationalization context: Management policies:

Internationalization policies prescribe how Java EE application components or their hosting containers manage internationalization context on component invocations. Two internationalization context management policies apply to all component types: Application-managed internationalization (AMI) and Container-managed internationalization (CMI).

These policies are represented in two parts:
- Internationalization type
- Container internationalization attribute

The service defines a default, or implicit, internationalization policy for every application component type. At development time, assemblers can override the default policy for server component types by explicitly

configuring their internationalization type, and optional container internationalization attributes. Policies configured during assembly are preserved in the deployment descriptor for the application.

All components have an internationalization type that indicates whether it is AMI or CMI; that is, whether a component is to deploy under the application-managed or the container-managed internationalization policy. Application assemblers can set the internationalization type for servlets, session beans, and message-driven beans. Entity beans are implicitly CMI and EJB clients are implicitly AMI; neither can be configured otherwise.

For CMI servlets and enterprise beans, optional container internationalization attributes can be specified to indicate which invocation internationalization context the container is to scope to service or business methods. A CMI service or business method invocation can run under the context of the caller's process, under the default context of the server JVM, or under a custom context specified in the attribute. Assemblers can specify one container internationalization attribute per disjoint set of CMI servlets within a Web module, or one Attribute per disjoint set of business methods of CMI beans within an EJB module. A container internationalization attribute can be associated with more than one method, but a method cannot be associated with more than one attribute.

When an application server launches an application, the internationalization service collects policy information from the deployment descriptor, then uses this information to construct and associate an internationalization policy to every component invocation. A policy is denoted as:

```
[<Internationalization Type>,<Container Internationalization Attribute>]
```

Several cases exist in which the deployment descriptor seems to lack policy information, for example: EJB client applications have no configurable internationalization policy settings; AMI components do not have container internationalization attributes; and you are not required to specify container internationalization attributes for CMI components. When the service cannot obtain the explicit internationalization type and container attribute settings from a well-formed deployment descriptor, it implicitly inserts the appropriate setting into the policy.

The service observes the following conventions when applying policies to invocations:

- Servlets (service) and EJB business methods lacking all internationalization policy information in the deployment descriptor implicitly run under policy `[CMI,RunAsCaller]`.
- CMI servlets and business methods lacking a container internationalization attribute in the deployment descriptor implicitly run under policy `[CMI,RunAsCaller]`.
- AMI servlets and business methods always lack container internationalization attributes in the deployment descriptor, but implicitly run under the logical policy `[AMI,RunAsServer]`.
- EJB clients always lack internationalization policy information in the deployment descriptor. By definition, EJB clients are implicitly AMI types and run under the invocation context of the JVM; they run under the logical policy `[AMI,RunAsServer]`.

For conditions other than these cited examples, such as a malformed deployment descriptor, refer to Internationalization service errors.

Internationalization policies for EJB clients and HTTP clients cannot be configured; HTTP clients do, however, run under the language priority settings of the hosting Web browser. These settings are configurable under the options dialog of most Web browsers. Refer to your Web browser documentation for details.

Internationalization type:

Every server application component has an *internationalization type* setting that indicates whether the invocation internationalization context is managed by the component or by the hosting Java EE container.

Server application components can be deployed to use one of two types of internationalization context management:

- Application-managed internationalization (AMI)
- Container-managed internationalization (CMI)

A server component can be deployed as AMI or CMI, but not both; CMI is the default. The setting applies to the entire component on every invocation. Entity beans use CMI only. Enterprise JavaBeans (EJB) client applications do not have an internationalization type setting; they implicitly use AMI.

Application-managed internationalization

Under the AMI deployment policy, component developers assume complete control over the invocation internationalization context. AMI components can use the internationalization context API to programmatically set invocation context elements.

AMI components are expected to manage invocation context. Invocations of AMI components implicitly run under the default locale and time zone of the hosting JVM. Invocation context elements not set using the API default to the corresponding elements of the JVM when accessed through the API or when exported on business methods. To export context elements other than the JVM defaults, AMI servlets, AMI enterprise beans, and EJB client applications must set (overwrite) invocation elements using the internationalization context API. Moreover, the container logically suspends the caller context that is imported on the AMI servlet lifecycle method and AMI EJB business method invocations. To continue propagating the context of the calling process, AMI servlets and enterprise beans must use the API to transfer caller context elements to the invocation context.

Specify AMI for server components that have internationalization context management requirements that are not supported by container-managed internationalization (CMI).

Container-managed internationalization

CMI is the preferred internationalization context management policy for server application components; it is also the default policy. Under CMI, the internationalization service collaborates with the Web and EJB containers to set the invocation internationalization context for servlets and enterprise beans. The service sets invocation context according to the container internationalization attribute of the policy that is associated with a servlet (service method) or an EJB business method.

A CMI policy has a container internationalization attribute that indicates which internationalization context the container is to scope to an invocation. For details, see Container internationalization attributes. By default, invocations of CMI components run under the caller's internationalization context; or rather, they adhere to the implicit policy `[CMI,RunasCaller]` whenever the servlet or business is not associated with an attribute in the deployment descriptor. For complete details, see Internationalization context: Management policies.

Methods within CMI components can obtain elements of the invocation context using the internationalization context API, but cannot set them. Any attempt to set invocation context elements within CMI components results in a java.lang.IllegalStateException exception.

Specify container-managed internationalization for server application components that require standard internationalization context management. Then specify the container internationalization attributes for CMI servlets and for business methods of CMI enterprise beans that you do not want to run under the caller's internationalization context.

Container internationalization attributes:

The internationalization policy of every CMI servlet and EJB business method has a *container internationalization attribute* that specifies which internationalization context the container is to scope to its invocation.

The container internationalization attribute has three main fields:
- Run as
- Locales
- Time zone ID

As a convenience, you can create named container internationalization attributes and associate them to the following subsets:
- CMI servlets within a Web module
- Business methods of CMI enterprise beans within an Enterprise JavaBeans (EJB) module
- Business methods of Web service-enabled session beans. In the following descriptions, the term *supported enterprise bean* refers to both CMI enterprise beans and Web service-enabled session beans.

Run-as field

The **Run-as** field specifies one of three types of invocation context that a container can scope to a method. For servlet service and EJB business methods, the container constructs the invocation internationalization context according to the **Run as** field setting and associates this context to the current thread before delegating to the method implementation.

By default, invocations of servlet service methods and EJB business methods implicitly run as caller (`RunAsCaller`) unless the **Run as** field of a policy attribute specifies otherwise. EJB client applications and AMI server components always run as server (`RunAsServer`).

You can specify the following invocation context types with the **Run as** field are:

Caller The container calls the method under the internationalization context of the calling process. For any missing context element, the container supplies the corresponding default context element of the Java virtual machine (JVM). Select run as caller when you want the invocation to run under the invocation context of the calling process.

Server
The container calls the method under the default locale and time zone of the JVM. Select run as server when you want the invocation to run under the invocation context of the JVM.

Specified
The container calls the method under the internationalization context specified in the attribute. Select run as specified when you want the invocation to run under the custom invocation context that is specified in the policy; then provide the custom context elements by completing the Locales and Time zone ID fields.

Note: Java Message Service (JMS) messages do not contain internationalization context. Although container-managed message-driven beans can be configured to run as caller, the container associates the default elements of the server process when calling the onMessage method of any message-driven bean that is configured as `[CMI, RunAsCaller]`. You can also configure the **Run as** field for Web service business methods.

Locales field

The **Locales** field specifies an ordered list of locales that the container scopes to an invocation. A locale represents a specific geographical, cultural, or political region and contains three fields:
- **Language code**. Ideally, language code is one of the lower-case, two-character codes that are defined by the ISO 639 standard; however, language code is not restricted to ISO codes and is not a required field. A valid locale must specify a language code if it does not specify a country code.
- **Country code**. Ideally, country code is one of the upper-case, two-character codes that are defined by the ISO 3166 standard; however, country code is not restricted to ISO codes and is not a required field. A valid locale must specify a country code if it does not specify a language code.
- **Variant**. Variant is a vendor-specific code. Variant is not a required field and serves only to supplement the language and country code fields according to application- or platform-specific requirements.

A valid locale must specify at least a language code or a country code; the variant is always optional. The first locale of the list is returned when accessing invocation context using the getLocale method of the internationalization context API.

Time zone ID field

The **Time zone ID** field specifies an abbreviated identifier for a time zone that the container scopes to an invocation. You can also configure the **Time zone ID** field for Web service business methods.

A time zone represents a temporal offset and computes daylight savings information. A valid ID indicates any time zone supported by the java.util.TimeZone type. Specifically, a valid ID is any of the IDs that appear in the list of time zone IDs returned by method java.util.TimeZone.getAvailableIds(), or a custom ID having the form `GMT[+|-]hh[[:]mm]`; for example, `America/Los_Angeles`, `GMT-08:00` are valid time zone IDs.

Object pools

Using object pools

An object pool helps an application avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused.

About this task

Object pools are not meant to be used for pooling JDBC connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To use an object pool, the product administrator must define an *object pool manager* using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Note: The Object pool manager service is only supported from within the EJB container or Web container. Looking up and using a configured object pool manager from a Java 2 Platform Enterprise Edition (J2EE) application client container is not supported.

1. Start the administrative console.
2. Click **Resources > Object pool managers**.
3. Specify a **Scope** value and click **New**.
4. Specify the required properties for work manager settings.
   Scope The scope of the configured resource. This value indicates the location for the configuration file.
   Name The name of the object pool manager. This name can be up to 30 ASCII characters long.
   JNDI Name
   The Java Naming and Directory Interface (JNDI) name for the pool manager.
5. [Optional] Specify a **Description** and a **Category** for the object pool manager.

Results

After you have completed these steps, applications can find the object pool manager by doing a JNDI lookup using the specified JNDI name.

Example

The following code illustrates how an application can find an object pool manager object:

```
InitialContext ic = new InitialContext();
ObjectPoolManager opm = (ObjectPoolManager)ic.lookup("java:comp/env/pool");
```

When the application has an ObjectPoolManager, it can cache an object pool for classes of the types it wants to use. The following is an example:

```
ObjectPool arrayListPool = null;
ObjectPool vectorPool = null;
try
{
 arrayListPool = opm.getPool(ArrayList.class);
 vectorPool = opm.getPool(Vector.class);
}
catch(InstantiationException e)
{
 // problem creating pool
}
catch(IllegalAccessException e)
{
 // problem creating pool
}
```

When the application has the pools, the application can use them as in the following example:

```
ArrayList list = null;
try
{
 list = (ArrayList)arrayListPool.getObject();
 list.clear(); // just in case
 for(int i = 0; i < 10; ++i)
 {
  list.add("" + i);
 }
 // do what ever we need with the ArrayList
}
finally
{
 if(list != null) arrayListPool.returnObject(list);
}
```

This example presents the basic pattern for using object pooling. If the application does not return the object, then the only adverse effect is that the object cannot be reused.

Object pool managers

Object pool managers control the reuse of application objects and Developer Kit objects, such as Vectors and HashMaps.

Multiple object pool managers can be created in an Application Server cell. Each object pool manager has a unique cell-wide Java Naming and Directory Interface (JNDI) name. Applications can find a specific object pool manager by doing a JNDI lookup using the specific JNDI name.

The object pool manager and its associated objects implement the following interfaces:

```
public interface ObjectPoolManager
{
 ObjectPool getPool(Class aClass)
  throws InstantiationException, IllegalAccessException;
 ObjectPool createFastPool(Class aClass)
  throws InstantiationException, IllegalAccessException;

}

public interface ObjectPool
{
 Object getObject();
 void returnObject(Object o);

}
```

The getObject() method removes the object from the object pool. If a getObject() call is made and the pool is empty, then an object of the same type is created. A returnObject( ) call puts the object back into the object pool. If returnObject() is not called, then the object is no longer allocatable from the object pool. If the object is not returned to the object pool, then it can be garbage collected.

Each object pool manager can be used to pool any Java object with the following characteristics:
- The object must be a public class with a public default constructor.
- If the object implements the java.util.Collection interface, it must support the optional clear() method.

Each pooled object class must have its own object pool. In addition, an application gets an object pool for a specific object using either the ObjectPoolManager.getPool() method or the ObjectPoolManager.createFastPool() method. The difference between these methods is that the getPool() method returns a pool that can be shared across multiple threads. The createFastPool() method returns a pool that can only be used by a single thread.

If in a Java virtual machine (JVM), the getPool() method is called multiple times for a single class, the same pool is returned. A new pool is returned for each call when the createFastPool() method is called. Basically, the getPool() method returns a pool that is thread-synchronized.

The pool for use by multiple threads is slightly slower than a fast pool because of the need to handle thread synchronization. However, extreme care must be taken when using a fast pool.

Consider the following interface:

```
public interface PoolableObject
{
 void init();
 void returned();
}
```

If the objects placed in the pool implement this interface and the ObjectPool.getObject() method is called, the object that the pool distributes has the init() method called on it. When the ObjectPool.returnObject() method is called, the PoolableObject.returned() method is called on the object before it is returned to the object pool. Using this method objects can be pre-initialized or cleaned up.

It is not always possible for an object to implement PoolableObject. For example, an application might want to pool ArrayList objects. The ArrayList object needs clearing each time the application reuses it. The application might extend the ArrayList object and have the ArrayList object implement a poolable object. For example, consider the following:

```
public class PooledArrayList extends ArrayList implements PoolableObject
{
 public PooledArrayList()
 {
 }

 public void init() {
 }

 public void returned()
 {
  clear();
 }
}
```

If the application uses this object, in place of a true ArrayList object, the ArrayList object is cleared automatically when it is returned to the pool.

Clearing an ArrayList object simply marks it as empty and the array backing the ArrayList object is not freed. Therefore, as the application reuses the ArrayList, the backing array expands until it is big enough

for all of the application requirements. When this point is reached, the application stops allocating and copying new backing arrays and achieves the best performance.

It might not be possible or desirable to use the previous procedure. An alternative is to implement a custom object pool and register this pool with the object pool manager as the pool to use for classes of that type. The class is registered by the WebSphere administrator when the object pool manager is defined in the cell. Take care that these classes are packaged in Java Archive (JAR) files available on all of the nodes in the cell where they might be used.

Object pool managers collection

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers**.

To use an object pool, the product administrator must define an object pool manager using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Name:

Specifies the name by which the object pool manager is known for administrative purposes.

Data type String
Range 1 through 30 ASCII characters

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name for the object pool manager.

Data type String

Scope:

Specifies the scope of the configured resource. This value indicates the location for the configuration file.

Description:

Specifies the description of the object pool manager.

Data type String

Category:

Specifies the category name used to classify or group this object pool manager.

Data type String

Object pool managers settings:

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers >** *objectpoolmanager_name*

To use an object pool, the product administrator must define an object pool manager using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Scope:

Specifies the scope of the configured resource. This value indicates the location for the configuration file.

Name:

The name by which the object pool manager is known for administrative purposes.

Data type	String
Range	1 through 30 ASCII characters

JNDI Name:

The Java Naming and Directory Interface (JNDI) name for the object pool manager.

Data type	String

Description:

A description of the object pool manager.

Data type	String

Category:

A category name used to classify or to group this object pool manager.

Data type	String

Custom object pool collection:

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers >** *objectpoolmanager_name* **> Custom object pools**.

Use custom object pools to insert additional logic around the following mechanisms:

- Constructing an object pool (A list of properties can be set)
- Flushing the object pool
- Getting objects from the pool
- Returning objects from the pool

These features allow for actions such as, clearing the state of an object when returning it to the pool, configuring the state of an object when retrieving it from the pool, or configuring generic pools and sending instructions on how to behave using custom properties.

To use an object pool the product administrator must define an object pool manager using the administrative console. You can create multiple object pool managers in an Application Server cell.

Pool class name:

Specifies the fully qualified class name of the objects that are stored in the custom object pool.

Data type String

Pool implementation class name:

Specifies the fully qualified class name of the implementation class for the custom object pool.

Data type String

Custom object pool settings:

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers >** *objectpoolmanager_name* **> Custom object pools >** *objectpool_name*.

Use custom object pools to insert additional logic around the following mechanisms:

- Constructing an object pool (A list of properties can be set)
- Flushing the object pool
- Getting objects from the pool
- Returning objects from the pool

These features allow for actions such as, clearing the state of an object when returning it to the pool, configuring the state of an object when retrieving it from the pool, or configuring generic pools and sending instructions on how to behave using custom properties.

To use an object pool, the product administrator must define an object pool manager using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Pool Class Name:

The fully qualified class name of the objects that are stored in the object pool.

Data type String

Pool Impl Class Name:

The fully qualified class name of the CustomObjectPool implementation class for this object pool.

Data type String

Object pool service settings

Use this page to enable or disable the object pool service, which manages object pool resources used by the server.

To view this administrative console page, click **Servers > Application Servers >** *server_name* > Container services > **Object Pool Service**.

Enable service at server startup:

Specifies whether the server attempts to start the object pool service.

Default Cleared

Range

Selected
When the application server starts, it attempts to start the object pool service automatically.

Cleared
The server does not try to start the object pool service. If object pool resources are used on this server, then the system administrator must start the object pool service manually or select this property, and then restart the server.

Object pools: Resources for learning

This topic provides links to find relevant supplemental information about object pools.

Use the following links to find relevant supplemental information about object pools. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Furthermore, these links provide guidance on using object pools. Since object pooling is a general topic and the WebSphere Application Server product implementation is only one way to use it, you must understand when object pooling is necessary. These articles help you make that decision.

Programming model and decisions

- Build your own ObjectPool in Java to boost application speed
- Improve the robustness and performance of your ObjectPool
- Recycle broken objects in resource pools

MBeans for object pool managers and object pools

Legacy MBean names for object pool managers and object pools are deprecated. The legacy names are based on the object pool manager name (which is not required to be unique) rather than the object pool manager JNDI name.

About this task

For object pools, the legacy name is also lacking any identifier of the version of the pooled class. Additionally, object pool Performance Monitoring Instrumentation (PMI) statistics are aggregated for object pools with the same legacy object pool MBean name.

For example, if the object pool manager and pooled class are as follows:

```
object pool manager name:               My ObjectPool
object pool manager JNDI name:          op/MyObjectPool
pooled class name:                      java.util.ArrayList
hash code of java.util.ArrayList.class: 1111eb3f (hexadecimal)
```

the legacy object pool manager MBean name will be:

```
ObjectPoolManager_My ObjectPool
```

and the legacy object pool MBean name will be:

```
ObjectPool_My ObjectPool_java.util.ArrayList
```

Instead of using the deprecated legacy MBean names, use the MBean names that are based on the JNDI name of the object pool manager.

For the example above, the JNDI name-based object pool manager MBean name is:

```
ObjectPoolManager_op/MyObjectPool
```

and the JNDI name-based object pool MBean name is:

```
ObjectPool_op/MyObjectPool_java.util.ArrayList.class@1111eb3f
```

Formats for MBean names

Type	Name format
Deprecated legacy object pool manager MBean name:	ObjectPoolManager_[object pool manager name]
JNDI name-based object pool manager MBean name:	ObjectPoolManager_[object pool manager JNDI name]
Deprecated legacy object pool MBean name:	ObjectPool_[object pool manager name]_[pooled class name]
JNDI name-based object pool MBean name:	ObjectPool_[object pool manager JNDI name]_[pooled class name].class@[hexadecimal representation of the hash code of the pooled class' java.lang.Class reference]

In all of the above formats, characters that are not valid for MBean names are replaced with the '.' character.

Scheduler

Using schedulers

Schedulers enable Java Platform, Enterprise Edition (Java EE) application tasks to run at a requested time. Schedulers also enable application developers to create their own stateless session Enterprise JavaBeans (EJB) components to receive event notifications during a task life cycle, allowing the plugging-in of custom logging utilities or workflow applications.

About this task

You can schedule the following types of tasks:

- Invoke a session bean method

- Send a Java Message Service (JMS) message to a queue or topic

Stateless session EJB components are also used to provide generic calendaring. Developers can either use the supplied calendar bean or create their own for their existing business calendars. For example, one of your business processes might involve invoicing for services. With the scheduler's use of stateless EJB components, you can schedule when periodic email distributions are to be sent to your customers who have received invoices. The scheduler service performs these tasks, repeating as necessary, according to the metadata for that task.

A scheduler is the mechanism by which the timer service for Enterprise Java Beans 2.1 runs. You can configure the EJB timer service to use many of the features that schedulers provide See the timer service for Enterprise Java Beans 2.1 documentation for more details.

Use the following table to determine which persistent timer service is best for you:

Schedulers	**EJB timers**
Run stateless session EJB components and sends JMS messages	Run all EJB types except for stateful session beans
Persistent, transactional and highly available.	Persistent, transactional and highly available.
Tasks guaranteed to run only once	Timers guaranteed to run only once, if the timer EJB uses a container-managed global transaction
Run repeating tasks using any calculation rules	Run repeating tasks using a repeating interval defined in milliseconds
Uses a modified fixed-delay time calculation to determine repeating intervals (next run time based on the start-time of the previous task)	Uses a fixed-rate time calculation to determine repeating intervals (time of the next task is based on the original scheduled time).
Programmatic task monitoring capability with the use of the NotificationSink stateless session EJB component	No programmatic timer monitoring
Abort late or time-sensitive tasks from running	Abort late or time-sensitive tasks from running (achieved through manual detection within the javax.ejb.TimedObject implementation).
Manage any task lifecycle (find, suspend, resume, cancel and purge tasks programmatically and through Java Management Extensions (JMX)).	Find and cancel its timers programmatically. Administrators find and cancel timers using a command-line utility.
Store a limited amount of text with the data, like a **Name** (arbitrary data stored externally).	Store arbitrary data with a timer

This task demonstrates how to manage, develop and interoperate with schedulers and subsequent tasks.

1. Manage the scheduler service. This topic includes instructions for creating and configuring schedulers, creating and configuring a database for schedulers and administering schedulers.
2. Develop and schedule tasks. This topic includes instructions for developing various types of tasks, receiving notifications from a task, submitting tasks to a scheduler, and managing tasks.

   Note: Creating and manipulating scheduled tasks through the Scheduler API interface is only supported from within the Enterprise Java Beans (EJB) container or Web container (JavaServer Pages or servlets). Looking up and using a configured scheduler from a Java EE application client container is not supported.
3. Interoperate with schedulers. This topic explains how to manage scheduler in a clustered environment with mixed WebSphere Application Server product versions and mixed platforms.

Example: Using default scheduler calendars

The SIMPLE and CRON calendars can be used from any J2EE application. This topic describes that process.

Using default scheduler calendars involves looking-up the default UserCalendarHome Enterprise JavaBeans (EJB) home object, creating the UserCalendar bean and calling the applyDelta() method. For details on the applyDelta method as well as the syntax for the SIMPLE and CRON calendars, see the UserCalendar interface topic.

Example:

```
import java.util.Date;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import com.ibm.websphere.scheduler.UserCalendar;
import com.ibm.websphere.scheduler.UserCalendarHome;

// Create an initial context
InitialContext ctx = new InitialContext();

// Lookup and narrow the default UserCalendar home.
UserCalendarHome defaultCalHome=(UserCalendarHome)
 PortableRemoteObject.narrow(ctx.lookup(
      UserCalendarHome.DEFAULT_CALENDAR_JNDI_NAME),
    UserCalendarHome.class);

// Create the default UserCalendar instance.
UserCalendar defaultCal = defaultCalHome.create();

// Calculate a date using CRON based on the current
// date and time.  Return the next date that is
// Saturday at 2AM
Date newDate =
    defaultCal.applyDelta(new Date(),
        "CRON", "0 0 2 ? * SAT");
```

Scheduler daemon

A scheduler daemon is a background thread that searches for tasks to run in the database.

A scheduler daemon is started for each scheduler defined on each server. If `Scheduler 1` is configured on `server1`, then only one scheduler daemon runs on `server1` unless it is cloned. If `Scheduler 1` is defined at the node scope level, then the scheduler will run on each server within that node.

The poll interval determines the frequency at which the persistent store is queried. By default, this value is set to 30 seconds. When a task is found that is scheduled to run within the current poll interval, an asynchronous beans alarm is set. The task then runs as close to this time as possible using an alarm thread from the scheduler's associated work manager. Thus, the number of alarm threads configured on the work manager determines how many concurrent tasks are executed. No tasks are lost. If we reach this limit, then new tasks are simply queued to be executed when an alarm thread becomes available. The actual firing time is dictated by server load and availability of free threads in the alarm thread pool of the associated work manager.

Scheduler daemons in a cluster

When multiple schedulers are configured to use the same tables (as is the case in a clustered environment), any of the daemons can find a task and set the alarm in its Java virtual machine (JVM). The task is executed in the virtual machine where the scheduler daemon first runs, until the daemon is stopped and another daemon starts. If an application on `server1` schedules a task to run and `server2` was started before `server1`, then the task runs on `server2`.

Example: Stopping and starting scheduler daemons using Java Management Extensions API:

Use the wsadmin scripting tool to invoke a `Jacl` script and stop and start a scheduler daemon.

This example JACL script can be invoked using the wsadmin scripting tool. It will attempt to stop and start a scheduler daemon.

```
# Example JACL Script to restart a Scheduler Daemon

set schedJNDIName sched/MyScheduler

# Find the WASScheduler MBean
regsub -all {/} $schedJNDIName "." schedJNDIName
set mbeanName Scheduler_$schedJNDIName
puts "Looking up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]

# Invoke the stopDaemon operation.
puts "Stopping the daemon..."
$AdminControl invoke $sched stopDaemon
puts "The daemon has stopped."

# Invoke the startDaemon operation.
puts "Starting the daemon..."
$AdminControl invoke $sched startDaemon 0
puts "The daemon has started."
```

Example: Dynamically changing scheduler daemon poll intervals using Java Management Extensions API:

Use the wsadmin scripting tool to invoke a `Jacl` script and dynamically change scheduler daemon poll intervals.

To dynamically change scheduler daemon poll intervals, use the wsadmin scripting tool to invoke this example `JACL` script. Invoking this example sets the poll interval of the scheduler daemon to 60 seconds.

```
# Example JACL Script to set the Scheduler daemon's poll interval

set schedJNDIName sched/MyScheduler

# Find the WASScheduler MBean
regsub -all {/} $schedJNDIName "." schedJNDIName
set mbeanName Scheduler_$schedJNDIName
puts "Looking-up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]

# Set the poll interval to 60 seconds (60000 ms)
$AdminControl setAttribute $sched pollInterval 60000
puts "Poll interval set."
```

Interoperating with schedulers

Schedulers support forward compatibility. Tasks created in previous versions of WebSphere Application Server Enterprise Edition 5.0 or WebSphere Business Integration Server Foundation 5.1 continue to run in WebSphere Application Server, Version 6.x schedulers. Tasks that you create using Version 6.x are not compatible with product schedulers from Version 5.x. Version 5.x schedulers do not run any Version 6.x tasks.

Schedulers and versions

All schedulers that are configured to use the same database and tables are considered a clustered scheduler. To guarantee that your tasks run correctly, all servers in a scheduler cluster must be at the same version. If the servers are at different versions, tasks created with a Version 6.x scheduler might not run. If a mixed-Version environment is required for a short period of time, then all scheduler poll daemons should be stopped on all Version 5.x servers to allow a Version 6.x server to run all tasks. This action allows the Version 6.x schedulers to obtain leases and run tasks that have been created with a Version 6.x scheduler.

Running tasks created with schedulers prior to Version 5.0.2 is not supported. See the topic, ″Interoperating with the Scheduler service,″ in the WebSphere Application Server Enterprise Edition Version 5.0.2 information center for details on how to migrate these tasks to a more recent version. See the Information Center Library to access the Version 5.0.2 information center.

Scheduler calendars

The scheduler provides stateless session bean interfaces which allow creating common calendars which can be used by the scheduler and any Java Platform, Enterprise Edition (Java EE) application.

The `SchedulerCalendars.ear` application is available and provides a default UserCalendar Enterprise Java Beans (EJB) implementation which allows using the SIMPLE and CRON calendars. Although this application is not required when using the scheduler, it is available to use from any Java EE application.

For details on how the SIMPLE and CRON calendars behave, see the API documentation for the com.ibm.websphere.scheduler.UserCalendar interface.

Specifying a UserCalendar with the scheduler

A UserCalendar is specified using the setUserCalendar() method of the TaskInfo interface of the scheduler. This interface allows you to select the Java Naming and Directory Interface (JNDI) name of the home interface of a UserCalendar bean. Because some UserCalendar bean implementations might handle multiple types of calendars, the interface also allows you to optionally select which type of calendar to use. A list of valid calendar types can be retrieved by invoking the getCalendarNames() method of the UserCalendar interface.

If the setUserCalendar() method is not invoked, or if a value of null or empty-string is specified for the home JNDI name parameter, then the default UserCalendar is used internally by the scheduler. When the default UserCalendar is accessed internally, it is not necessary that the `SchedulerCalendars.ear` system application be installed.

You might want to use the default UserCalendar directly in your other Java EE applications, apart from the scheduler. In this case, you may use the UserCalendarHome.DEFAULT_CALENDAR_JNDI_NAME value to look up the default UserCalendar from your applications. You may also supply this value to the setUserCalendar() method of the TaskInfo interface. You will need to ensure the `SchedulerCalendars.ear` system application was either automatically installed or that you have installed it manually.

Scheduler service settings

Use this page to enable or disable the scheduler service. The scheduler service manages scheduler resources used by the server. The administrative console page used to configure the scheduler service is not available for version 6 (and above) servers. It is only available for version 5.*x* servers.

To view this administrative console page, click **Servers > Application Servers >** *server_name* **> Scheduler Service**.

Startup:

Specifies whether the server attempts to start the scheduler service.

Default Selected

Range

Selected
When the application server starts, it attempts to start the scheduler service automatically.

Cleared
The server does not try to start the scheduler service. If scheduler resources are to be used on this server, the system administrator must start the scheduler service manually or select this property, then restart the server.

Installing default scheduler calendars

The default scheduler SIMPLE and CRON calendars are available in the `SchedulerCalendars.ear` system application and are automatically installed on standalone server profiles. System applications cannot be installed and uninstalled like traditional Java Platform, Enterprise Edition (Java EE) applications.

About this task

The following steps are required to map the `SchedulerCalendars.ear` system application on a server or cluster in a network deployment environment.

1. Start the wsadmin tool and connect to the deployment manager.
2. Install the system application.
   - To install on a non-clustered server:
     - Using Jacl:

```
$AdminApp install
"\${WAS_INSTALL_ROOT}/systemApps/SchedulerCalendars.
ear" {-systemApp -appname SchedulerCalendars -cell
 mycell -node mynode -server myserver}
```

     - Using Jython list:

```
AdminApp.install('${WAS_INSTALL_ROOT}/systemApps/
SchedulerCalendars.ear', ['-systemApp', '–appName',
 'SchedulerCalendars','-cell', 'mycell', '-node',
 'mynode', '-server', 'myserver'])
```

     - Using Jython string:

```
AdminApp.install('${WAS_INSTALL_ROOT}/systemApps/
SchedulerCalendars.ear', '[-systemApp –appName
 SchedulerCalendars –cell mycell –node mynode
 -server myserver]')
```

       Where:

Value	Option
mycell	the value of the cell option
mynode	the value of the node option
myserver	the value of the server option

   - To install on a cluster:
     - Using Jacl:

```
$AdminApp install
"\${WAS_INSTALL_ROOT}/systemApps/SchedulerCalendars.
ear" {-systemApp -appname SchedulerCalendars -cell
 mycell -cluster mycluster}
```

     - Using Jython list:

```
AdminApp.install('${WAS_INSTALL_ROOT}/systemApps/
SchedulerCalendars.ear', ['-systemApp', '–appName',
 'SchedulerCalendars','-cell', 'mycell', '-cluster',
 'mycluster'])
```

– Using Jython string:

```
AdminApp.install('${WAS_INSTALL_ROOT}/systemApps/
SchedulerCalendars.ear', '[-systemApp –appName
 SchedulerCalendars –cell mycell -cluster mycluster]')
```

Where:

Value	Option
mycell	the value of the cell option
mycluster	the value of the cluster option

3. Save the configuration changes.
4. Synchronize the node.

Uninstalling default scheduler calendars

The default scheduler SIMPLE and CRON calendars are available in the `SchedulerCalendars.ear` system application and are automatically installed on standalone server profiles. System applications cannot be installed and uninstalled like traditional Java Platform, Enterprise Edition (Java EE) applications.

About this task

Remove the SchedulerCalendars system application on federated node, as follows:

1. Open a command window on the federated node.
2. Run the following command:

   On Unix platforms:

   ```
   $Install_Root/bin/wsadmin.sh -conntype none -profile $Profile_Name
   ```

   On Windows platforms:

   ```
   $Install_Root/bin/wsadmin -conntype none -profile $Profile_Name
   ```

   where:

   - *Install_Root* is the directory where WebSphere Application Server is installed.
   - *Profile_Name* is the name of the profile where the target server is located.

3. At the **wsadmin>** prompt, enter the following command for each server that exists on the node where you want to have the SchedulerCalendars application available:

   ```
   wsadmin> $AdminApp uninstall SchedulerCalendars "-cell $MyCell -node $MyNode -server $MyServer"
   ```

   where:

   - *$Install_Root* is the directory where WebSphere Application Server is installed.
   - *$MyCell*, *$MyNode*, and *$MyServer* are the values with the name of the cell, node, and server.

   Note: Each of these values are case-sensitive.

4. Repeat step three for each server in the current profile for which you will uninstall the SchedulerCalendars application.
5. When uninstallation is complete for the system application on all appropriate servers, enter the following commands:

   ```
   wsadmin> $AdminConfig save
   wsadmin> exit
   ```

6. Repeat steps 1-5 for each federated node or profile where the SchedulerCalendars application is to be removed.
7. Using the Administrative Console or scripting, start or restart the servers to unload the SchedulerCalendars application.

Results

The SchedulerCalendars application should now be removed.

Developing and scheduling tasks

To develop and schedule tasks, use a configured scheduler.

1. Look up a configured scheduler. Each configured scheduler is available from two different programming models:
   - A Java Platform, Enterprise Edition (Java EE) server application, such as a servlet or Enterprise JavaBeans (EJB) component, can use the Scheduler API. Schedulers are accessed by looking them up using a Java Naming and Directory Interface (JNDI) name or resource reference.
   - Java Management Extensions (JMX) applications, such as wsadmin scripts, can use the Scheduler API using WASScheduler MBeans.
2. Develop the task.

   The Scheduler API supports different implementations of the TaskInfo interface, each of which can be used to schedule a particular type of work. Refer to one of the following topics for details:
   - Developing a task that calls a session bean.
   - Develop a task that sends a Java Message Service (JMS) message. This task object can send a JMS message to either a queue or a topic.

   Note: Creating and manipulating scheduled tasks through the Scheduler interface is only supported from within the EJB container or Web container (Enterprise beans or servlets). Looking up and using a configured scheduler from a Java EE application client container is not supported.
3. Receive scheduler notifications. A notification sink is set on a task in order to receive the notification events that are generated by a scheduler when it performs an operation on the task.
4. Use custom calendars. You can assign a UserCalendar session bean to a task that allows schedulers to use custom and predefined date algorithms to determine when a task should run. See the UserCalendar interface for details.
5. Submit tasks to a scheduler. After a TaskInfo object has been created, it can be submitted to the scheduler for task creation by calling the Scheduler.create() method.
6. Manage tasks with a scheduler.
7. Secure tasks with a scheduler.

Accessing schedulers

Each configured scheduler is available using the Scheduler API from a Java Platform, Enterprise Edition (Java EE) server application, such as a servlet or Enterprise JavaBean (EJB) module. Use a Java Naming and Directory Interface (JNDI). name or resource reference to access schedulers. Each scheduler is also available using the Java™ Management Extensions (JMX) API, using its associated WASScheduler MBean.

About this task

Scheduler and WASScheduler interfaces are the starting point for all scheduler activities. Each scheduler is independent and allows task life cycle operations, such as creating new tasks.

1. Locate schedulers using the javax.naming.Context.lookup() method from a Java EE server application, such as a servlet or EJB module like the following example:

```
//lookup the scheduler to be used
import com.ibm.websphere.scheduler.Scheduler;
import javax.naming.InitialContext;
Scheduler scheduler = (Scheduler)new InitialContext.lookup("java:comp/env/sched/MyScheduler");
```

2. Use wsadmin to locate a WASScheduler MBean using JACL scripting:

```
set jndiName sched/MyScheduler

# Map the JNDI name to the mbean name.  The mbean name is
# formed by replacing the / in the JNDI namewith . and prepending
# Scheduler_
regsub -all {/} $jndiName "." jndiName
set mbeanName Scheduler_$jndiName

puts "Looking-up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
puts $sched
```

Results

The scheduler is now available to use from a Java EE server application or from a JMX API client. To create a task see the topics, Developing a task that calls a session bean or Developing a task that sends a JMS message.

Developing a task that calls a session bean

The Scheduler API and WASScheduler MBean API support different implementations of the TaskInfo interface, each of which can be used to schedule a particular type of work. This topic describes how to create a task to call a method on a TaskHandler session bean.

About this task

To create a task to call a method on a TaskHandler session bean, use these steps.

1. Create a new enterprise application with an Enterprise JavaBeans (EJB) module. This application hosts the TaskHandler EJB module.
2. Create a stateless session bean in the EJB Module that implements the process() method in the com.ibm.websphere.scheduler.TaskHandler remote interface. Place the business logic you want created in the process() method. The process() method is called when the task runs. The Home and Remote interfaces must be set as follows in the deployment descriptor bean:
   - com.ibm.websphere.scheduler.TaskHandlerHome
   - com.ibm.websphere.scheduler.TaskHandler
3. Create an instance of the BeanTaskInfo interface by using the following example factory method. Using a JavaServer Pages (JSP) file, servlet or EJB component, create the instance as shown in the following code example. This code should coexist in the same application as the previously created TaskHandler EJB module:

   ```
   // Assume that a scheduler has already been looked-up in JNDI.
   BeanTaskInfo taskInfo = (BeanTaskInfo) scheduler.createTaskInfo(BeanTaskInfo.class)
   ```

   You can also use the wsadmin tool to create the instance as shown in the following JACL scripting example:

   ```
   set taskHandlerHomeJNDIName ejb/MyTaskHandler

   # Map the JNDI name to the mbean name.  The mbean name is formed by replacing the / in the jndi name
   # with . and prepending Scheduler_
   regsub -all {/} $jndiName "." jndiName
   set mbeanName Scheduler_$jndiName

   puts "Looking-up Scheduler MBean $mbeanName"
   set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
   puts $sched
   ```

```
# Get the ObjectName format of the Scheduler MBean
set sched0 [$AdminControl makeObjectName $sched]

# Create a BeanTaskInfo object using invoke_jmx
puts "Creating BeanTaskInfo"
set params [java::new {java.lang.Object[]} 1]
$params set 0 [java::field com.ibm.websphere.scheduler.BeanTaskInfo class]

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 java.lang.Class

set ti [$AdminControl invoke_jmx $sched0 createTaskInfo $params $sigs]
set bti [java::cast com.ibm.websphere.scheduler.BeanTaskInfo  $ti]
puts "Created the BeanTaskInfo object: $bti"
```

Note: Creating a BeanTaskInfo object does not add the task to the persistent store. Rather, it creates a placeholder for the necessary data. The task is not added to the persistent store until the create() method is called on a Scheduler, as described in the topic Submitting tasks to schedulers.

4. Set parameters on the BeanTaskInfo object. These parameters define which session bean is called and when. The TaskInfo interface contains various set() methods that you can use to control execution of the task, including when the task runs and what work the task does when it runs.

   The BeanTaskInfo interface requires that the TaskHandler Java™ Naming and Directory Interface (JNDI) name or TaskHandlerHome is set using the setTaskHandler method. If using the WASScheduler MBean API to set the task handler, then the JNDI name must be the fully-qualified global JNDI name.

   The TaskInfo interface specifies additional control points, as documented in the API documentation.Set parameters using the TaskInfo interface API method as shown in the following code example:

```
//create a date object which represents 30 seconds from now
java.util.Date startDate = new java.util.Date(System.currentTimeMillis()+30000);

//find the session bean to be called when the task executes
Object o = new InitialContext().lookup("java:comp/env/ejb/MyTaskHandlerHome");
TaskHandlerHome home = (TaskHandlerHome)javax.rmi.PortableRemoteObject.narrow(o,TaskHandlerHome.class);

//now set the start time and task handler to be called in the task info
taskInfo.setTaskHandler(home);
taskInfo.setStartTime(startDate);
```

   You can also set parameters using the following JACL scripting example:

```
# Setup the task
puts "Setting up the task..."
# Set the startTime if you want the task to run at a specific time, for example:
$bti setStartTime [java::new {java.util.Date long} [java::call System currentTimeMillis]]

# Set the StartTimeInterval so the task runs in 30 seconds from now
$bti setStartTimeInterval 30seconds

# Set JNDI name of the EJB which will get called when the task runs.  Since there is no
# application J2EE Context when the task is created by the MBean, this must be a
# global JNDI name.
$bti setTaskHandler $taskHandlerHomeJNDIName

# Do not purge the task when it's complete
$bti setAutoPurge false

# Set the name of the task.  This can be any string value.
$bti setName Created_by_MBean

# If the task needs to run with specific authorization you can set the tasks Authentication Alias
```

```
# Authentication aliases are created using the Admin Console.
# $bti setAuthenticationAlias {myRealm/myAlias}

puts "Task setup completed."
```

Results

A BeanTaskInfo object has been created that contains all of the relevant data to call an EJB method.

What to do next

Submit the task to a scheduler for creation, as described in the topic Submitting a task to a scheduler.

Developing a task that sends a Java Message Service message

The Scheduler API and WASScheduler MBean API support different implementations of the TaskInfo interface, each of which can be used to schedule a particular type of work. This topic describes how to create a task that sends a Java Message Service (JMS) message to a queue or topic.

About this task

To create a task that sends a Java Message Service (JMS) message to a queue or topic, use these steps.

1. Create an instance of the MessageTaskInfo interface using the Scheduler.createTaskInfo() factory method. Using a JavaServer Pages (JSP) file, servlet or EJB container, create the instance as shown in the following code example:

   ```
   //lookup the scheduler to be used
   Scheduler scheduler = (Scheduler)new InitialContext.lookup("java:comp/env/Scheduler");

   MessageTaskInfo taskInfo = (MessageTaskInfo) scheduler.createTaskInfo(MessageTaskInfo.class);
   ```

   You can also use the wsadmin tool, create the instance as shown in the following JACL scripting example:

   ```
   # Sample create a task using MessageTaskInfo task type
   # Call this mbean with the following parameters:
   #    <scheduler jndiName>     = JNDI name of the scheduler resource,
   #                               for example scheduler/myScheduler
   #    <JNDI name of the QCF>   = The global JNDI name of the Queue Connection Factory.
   #    <JNDI name of the Queue> = The global JNDI name of the Queue destination

   set jndiName [lindex $argv 0]
   set jndiName_QCF [lindex $argv 1]
   set jndiName_Q [lindex $argv 2]

   # Map the JNDI name to the mbean name.  The mbean name is formed by replacing the / in the jndi name
   # with . and prepending Scheduler_
   regsub -all {/} $jndiName "." jndiName
   set mbeanName Scheduler_$jndiName

   puts "Looking-up Scheduler MBean $mbeanName"
   set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
   puts $sched

   # Get the ObjectName format of the Scheduler MBean
   set schedO [$AdminControl makeObjectName $sched]

   # Create a MessageTaskInfo object using invoke_jmx
   puts "Creating MessageTaskInfo"
   set params [java::new {java.lang.Object[]} 1]
   $params set 0 [java::field com.ibm.websphere.scheduler.MessageTaskInfo class]

   set sigs [java::new {java.lang.String[]} 1]
   $sigs set 0 java.lang.Class
   ```

```
set ti [$AdminControl invoke_jmx $schedO createTaskInfo $params $sigs]
set mti [java::cast com.ibm.websphere.scheduler.MessageTaskInfo  $ti]
puts "Created the MessageTaskInfo object: $mti"
```

Note: Creating a MessageTaskInfo object does not add the task to the persistent store. Rather, it creates a placeholder for the necessary data. The task is not added to the persistent store until the create() method is called on a Scheduler, as described in the topic Submitting a task to a scheduler.

2. Set parameters on the MessageTaskInfo object. The TaskInfo interface contains various set() methods that can be used to control execution of the task, including when the task runs and what work the task does when it starts.

   The TaskInfo interface specifies additional behavior settings, as documented in the API documentation. Using a JavaServer Pages (JSP) file, servlet or EJB container, create the instance as shown in the following code example:

```
//create a date object which represents 30 seconds from now
java.util.Date startDate = new java.util.Date(System.currentTimeMillis()+30000);


//now set the start time and the JNDI names for the queue connection factory and the queue
taskInfo.setConnectionFactoryJndiName("jms/MyQueueConnectionFactory");
taskInfo.setDestination("jms/MyQueue");
taskInfo.setStartTime(startDate);
```

   You can also use the wsadmin tool, to create the instance as shown in the following JACL scripting example:

```
# Setup the task
puts "Setting up the task..."
# Set the startTime if you want the task to run at a specific time, for example:
$mti setStartTime [java::new {java.util.Date long} [java::call System currentTimeMillis]]

# Set the StartTimeInterval so the task runs in 30 seconds from now
$mti setStartTimeInterval 30seconds

# Set the global JNDI name of the QCF & Queue to send the message to.
$mti setConnectionFactoryJndiName $jndiName_QCF
$mti setDestinationJndiName $jndiName_Q

# Set the message
$mti setMessageData "Test Message"

# Do not purge the task when it's complete
$mti setAutoPurge false

# Set the name of the task.  This can be any string value.
$mti setName Created_by_MBean

# If the task needs to run with specific authorization you can set the tasks Authentication Alias
# Authentication aliases are created using the Admin Console.
# $mti setAuthenticationAlias {myRealm/myAlias}

puts "Task setup completed."
```

Results

A MessageTaskInfo object has been created that contains all of the relevant data for a task that sends a JMS message.

What to do next

Submit the task to a scheduler for creation, as described in the topic Submitting a task to a scheduler.

Scheduling long-running tasks

The default behavior of the scheduler is designed to run business logic that runs for a short period of time. In version 6.0.2 and later, two API methods on the com.ibm.websphere.scheduler.TaskInfo interface help avoid some of the problems that can occur when running tasks for an extended time.

About this task

The TaskInfo.setQOS method supports tasks with both a transactional and non-transactional quality of service. When running tasks that run for long periods, you can use the TaskInfo.QOS_ATLEASTONCE quality of service to run the task without a global transaction. This process prevents various timeout issues that can occur when resources are held by a long-running transaction. See Transactions and schedulers for details on the TaskInfo.setQOS method and how it can be used.

Using the TaskInfo.setExpectedDuration method, the scheduler can to adjust timeout values, as appropriate, for a given task for all qualities of service. The application server attempts to adjust various run-time parameters to accommodate the estimated run time of the task.

1. When you assemble the TaskInfo object with the Scheduler API or the WASScheduler MBean, use the following methods on the TaskInfo interface:
   a. Set the quality of service.
      1) If the task must be transactional, use the setQOS method with the QOS_ONLYONCE constant, which is the default, if not set.
      2) If the task does not need to be transactional, use the setQOS method with the QOS_ATLEASTONCE constant.
   b. Set the expected duration.
      1) Use the setExpectedDuration method to set the expected duration of the task in seconds.
2. Schedule the task using the Scheduler.create method.

What to do next

Access schedulers.

Receiving scheduler notifications

Various notification events are generated by a scheduler when it performs an operation on a task. These notifications events are described in this topic.

About this task

The notification events generated by a scheduler when it performs a task include:

Scheduled
A task has been scheduled.

Purged
A task has been permanently deleted from the persistent store.

Suspended
A task was suspended.

Resumed
A task was resumed.

Complete
A task has run completely. If it was a repeating task, all repeats have been performed.

Cancelled
A task has been cancelled. It will not run again.

Firing A task is prepared to run.

Fired A task completed successfully.

Fire failed
A task could not run successfully.

To receive notification events, call the setNotificationSink() method on the TaskInfo interface before creating the task. The setNotificationSink() method enables you to specify the session bean that is to act as the callback, and a mask that restricts which events are generated.

1. Create a NotificationSink session bean. Create a stateless session bean that implements the handleEvent() method in the com.ibm.websphere.scheduler.NotificationSink remote interface. The handleEvent() method is called when the notification is fired. The Home and Remote interfaces can be set as follows in the bean's deployment descriptor:

   ```
   com.ibm.websphere.scheduler.NotificationSinkHome
   com.ibm.websphere.scheduler.NotificationSink
   ```

   The NotificationSink interface defines the following method:

   ```
   public void handleEvent(TaskNotificationInfo task) throws java.rmi.RemoteException;
   ```

2. Specify the notification sink session bean prior to submitting the task to the Scheduler using the TaskInfo interface API setNotificationSink() method.

   If using the WASScheduler MBean API to set the notification sink, then the Java™ Naming and Directory Interface (JNDI) name must be the fully-qualified global JNDI name. Using a JavaServer Pages (JSP) file, servlet or Enterprise JavaBeans (EJB) component, look up and set the notification sink on a task as shown in the following code example:

   ```
   TaskInfo taskInfo = ...
   Object o = new InitialContext().lookup("java:comp/env/ejb/NotificationSink");
   NotificationSinkHome home = (NotificationSinkHome )javax.rmi.PortableRemoteObject.narrow
   (o,NotificationSinkHome.class);
   taskInfo.setNotificationSink(home,TaskNotificationInfo.ALL_EVENTS);
   ```

   You can also use the wsadmin tool to set the notification sink callback session bean as shown in the following JACL scripting example:

   ```
   # Use the NotificationSinkHome's Global JNDI name
   # Assume that a TaskInfo was already created...
   $taskInfo setNotificationSink "ejb/MyNotificationSink"
   ```

3. Specify the event mask. The event mask is specified as an integer bitmap. You can either use an individual mask such as TaskNotificationInfo.CREATED to receive specific events, TaskNotificationInfo.ALL_EVENTS to receive all events or a combination of specific events. If you use Java, your script might look like the following example:

   ```
   int eventMask = TaskNotificationInfo.FIRED | TaskNotificationInfo.COMPLETE;
   taskInfo.setNotificationSink(home,eventMask);
   ```

   If you use JACL, your script might look like the following example:

   ```
   # Set the event mask based on two event constants.
   set eventmask [expr [java::field com.ibm.websphere.scheduler.TaskNotificationInfo FIRED] +
    [java::field com.ibm.websphere.scheduler.TaskNotificationInfo COMPLETE]]

   # Set our Notification Sink based on our global JNDI name AND event mask.
   # Note:  We need to use the full method signature here since the
   # method resolver can't always detect the right method.
   $taskInfo {setNotificationSink String int} "ejb/MyNotificationSink" $eventmask
   ```

Results

A notification sink bean is now set on a TaskInfo object and can now be submitted to a scheduler using the create method.

Submitting a task to a scheduler

This topic describes the process of submitting a task to a configured scheduler.

Before you begin

This task assumes that you have already configured a scheduler and created and configured a TaskInfo object that calls a session bean or sends a Java™ Messaging Service (JMS) message.

About this task

Once you have developed a TaskInfo object that contains all relevant data for a task, submit the task to a scheduler for creation. When the task is created, the scheduler runs it.

Create the task. After you configure TaskInfo, submit it to the appropriate scheduler, using the Scheduler API create method.

```
// Create the TaskInfo using the Scheduler that you already looked up and print out the Task ID
TaskStatus ts = scheduler.create(taskInfo);
System.out.println("Task created with id: " + ts.getTaskId()"
```

You can also create the task using the wsadmin tool as shown in the following JACL scripting example:

```
# Create the TaskInfo using the WASScheduler MBean that you previously located and print out the Task ID
puts "Creating the task..."

set params [java::new {java.lang.Object[]} 1]
$params set 0 $taskInfo

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 com.ibm.websphere.scheduler.TaskInfo

set taskStatus [java::cast com.ibm.websphere.scheduler.TaskStatus [$AdminControl invoke_jmx $schedO
 create $params $sigs]]

puts "Task Created.  TaskID= [$taskStatus getTaskId]"

puts $taskStatus
```

When the call to the create() method is complete, the task exists in the persistent store and is run at the time specified in the TaskInfo object. If a global transactional context is present on the thread, and the create() transaction rolls back or is aborted, the task does not run.
The TaskStatus object, which has been returned by the call to the create() method, contains information about the state of the task, as well as the task ID. The task ID is the unique identifier for this task, and is required if the task is to be suspended, resumed, cancelled, and so on, at a later time.

Note: The TaskStatus object is only a snapshot of the current state of the task. Use the Scheduler.getStatus() method to receive the current state when needed.

Task management methods using a scheduler

The scheduler provides several task management methods.

When a task is created by calling the create() method on a scheduler, a TaskStatus object is returned to the caller. The TaskStatus object contains the task ID, which is a unique identifier. The Scheduler API and WASScheduler MBean define several additional methods that pertain to the management of tasks, each of which accepts the task ID as a parameter. The following task management methods are defined:

suspend()
: Suspends a task. The task does not run until it has been resumed.

resume()
: Resumes a previously suspended task.

cancel()
: Cancels a task. The task is not run and cannot be resumed.

purge()
: Permanently deletes a cancelled task from the persistent store.

getStatus()
Returns the current status of the task.

Use the following API example to create and cancel a task:

```
//Create the task.
TaskInfo taskInfo = ...
TaskStatus status = scheduler.create(taskInfo);

//Get the task ID
String taskId = status.getTaskId();

//Cancel the task. Specify the purgeAlso flag so that the task does not remain in the persistent store
scheduler.cancel(taskId,true);
```

Use the following example JACL script operations in the wsadmin tool to create and cancel a task:

```
set jndiName sched/MyScheduler

# Map the JNDI name to the mbean name.  The mbean name is
# formed by replacing the / in the jndi name with . and prepending
# Scheduler_
regsub -all {/} $jndiName "." jndiName
set mbeanName Scheduler_$jndiName

puts "Looking-up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
puts $sched

# Get the ObjectName format of the Scheduler MBean
set schedO [$AdminControl makeObjectName $sched]

# Create a TaskInfo object...
# (Some code excluded...)
set params [java::new {java.lang.Object[]} 1]
$params set 0 $taskInfo

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 com.ibm.websphere.scheduler.TaskInfo

set taskStatus [java::cast com.ibm.websphere.scheduler.TaskStatus [$AdminControl invoke_jmx $schedO
 create $params $sigs]]

set taskID [$taskStatus getTaskId]
puts "Task Created.  TaskID= $taskID"

# Cancel the task using the Task ID from the TaskStatus object returned during create.
set params [java::new {java.lang.Object[]} 1]
$params set 0 false

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 java.lang.boolean

set taskStatus [java::cast com.ibm.websphere.scheduler.TaskStatus [$AdminControl invoke_jmx $schedO
 cancel $params $sigs]]
```

Transactionality. All methods of the Scheduler API are transactional. If a global transactional context is present, it is used to perform the operation. If an unexpected exception is thrown, the transaction is marked to roll back, and the caller must handle it appropriately. If an expected or declared exception is thrown, the transaction remains intact and the caller must choose to roll back or to commit the transaction. If the transaction is rolled back at some point, all scheduler operations performed within the transaction are also rolled back.

If a local transactional context is present, it is suspended and a new global transactional context begins. Likewise, if no transactional context is active, a global transactional context begins. In both cases, if an unexpected exception is thrown, the transaction rolls back. If a declared exception is thrown, the transaction is committed.

If another thread is concurrently modifying the task in question, a TaskPending exception is thrown. This is because schedulers lock the database optimistically. The calling application can then retry the operation.

Task management functions may block if the task is currently running. Because the scheduler guarantees that each task will run only once, the task must be locked for the duration of a running task. Likewise, if a task is changed using one of the management functions but the global transaction is not committed, any other management functions issued from another transaction for that task will be blocked.

A stateless session bean task's TaskHandler.process() method can change it's own state. However, the task must be running within the same transaction as the scheduler. Therefore, a running task can only modify itself if it is using the Required or Mandatory container managed transaction types. If the Requires New transaction type is specified on the process() method, all management functions will deadlock.

All methods defined by the Scheduler API are described in the API documentation.

Identifying tasks that are currently running

When a task runs, the task database record is locked until the task completes. This topic describes how to determine whether or not a task is running.

About this task

Prior to version 6.0.2, all tasks ran in a single global transaction. This process not only prevented the task from running more than once successfully, but it also blocked all attempts at reading the state of the task, since each task used read-committed transaction isolation.

There are two methods for determining whether a task is running:

1. **NotificationSink**

   A NotificationSink EJB can be set on the task using the setNotificationSink method on the TaskInfo object. The NotificationSink bean can then log the life cycle of the task to a separate database record in a custom table. This would result in a history log of the task that can be queried independently from the scheduler. This solution works for all versions of the scheduler service. See Receiving Scheduler Notifications for details.

2. **Delayed Execution and Uncommitted Read**

   In Version 6.0.2 and later, two behaviors enable the scheduler find and retrieve API methods, such as getTask, getTaskStatus or findTasksByName, to see the current state of the task without blocking. To see the current state of the task, including its uncommitted running state, complete the following steps:

1. Enable read-uncommitted transaction isolation for the scheduler read methods to prevent these methods from blocking while a task is running. To set the default transaction isolation for read methods, see Configuring scheduler default transaction isolation for read operation details.

   Note: If the scheduler database does not support uncommitted reads, such as Oracle, it might not be possible to determine if a task is running unless you use the QOS_ATLEASTONCE quality of service.

2. Use the TaskInfo.EXECUTION_DELAYEDUPDATE option on the TaskInfo.setTaskExecutionOptions method to force the scheduler to write the TaskStatus.RUNNING state to the task when that task starts running.

Stopping tasks that are failing

The scheduler runs tasks in a global transactional context, by default. If a task is failing due to a configuration problem or application error, the scheduler attempts to retry the task until the scheduler failure threshold is reached. This topic describes how to stop the tasks that are failing.

About this task

When the task reaches the failure threshold, the scheduler stops running the task until the scheduler daemon is restarted using the WASScheduler MBean, the scheduler fails over to another server, or until the scheduler is resumed using the resume method on the Scheduler API or WASScheduler MBean.

1. Cancel or suspend a transactional (QOS_ONLYONCE) task that is continually failing. This action can be difficult if the scheduler has not yet reached the failure threshold. The cancel and suspend Scheduler API methods or WASScheduler MBean operations block until the task fails or the method times out, while waiting for a database lock and throws a TaskPending exception. If this is occurs, then the application can retry the cancel or suspend operation until it completes.
2. Alternatively, stop the scheduler daemon using the stopDaemon operation on the WASScheduler MBean to avoid running the task multiple times, and run the cancel or suspend operation while it is stopped. While the daemon is stopped, the scheduler does not run tasks. However, all MBean operations and API methods are still available.

Scheduler tasks and Java EE context

When a task is created using the Scheduler API create() method, the Java Platform, Enterprise Edition (Java EE) thread context of the creator is stored with the scheduled task. When the task runs, the original Java EE thread context is reapplied to the thread before calling the customer TaskInfo instance.

The scheduler service utilizes the asynchronous beans deferred start mechanism to propagate Java EE service context information to a task when it runs. The amount of service context information that is propagated is controlled by the Service Context settings on the WorkManager configuration object that schedulers reference. For example, security and internationalization service contexts can be enabled. See Using asynchronous beans for details on how to configure the Application Server to propagate these service contexts.

Transactions and schedulers:

The scheduler runs a task in a single global transaction, by default. You can use the QOS_ONLYONCE or QOS_ATLEASTONCE quality of service to specify whether the task runs as a single unit of work once or as independent transactions.

Transaction behavior when running a task

Because the scheduler runs a task in a single global transaction, by default, the transaction is open until the task completes or fails. The resources involved in that transaction are subject to various timeouts and the thread of the task could be identified as hung if the task runs for a long period of time that can span many minutes or hours.

QOS_ONLYONCE

Scheduled tasks execute only one time successfully when using the QOS_ONLYONCE quality of service. This action is accomplished by grouping all of the work done in the task as a single unit of work. When each task fires, the following events occur in a single global transactional context:

1. The context of the application that created the task is applied to the thread.
2. A global transactional context is started.
3. The next fire time and start-by time are calculated using the UserCalendar bean or the DefaultUserCalendar.

Note: If using the TaskInfo.setTaskExcecutionOptions method with the TaskInfo.EXECUTION_DELAYEDUPDATE option, this step will occur after the record is updated.

4. The task database task record is updated in the database with the state of the next task or deleted if the task is complete and the task's auto-purge setting is true.
5. The task database record is updated in the database with the state of the next task or deleted if the task is complete and the task's auto-purge setting is true. If the EXECUTION_DELAYEDUPDATE option is used, the database will not reflect the next state of the task, but the current state with the TaskStatus.RUNNING state set.
6. If the NotificationSink bean is set, a FIRING notification is fired.
7. The BeanTaskInfo or MessageTaskInfo object starts.
8. If the task fails and the NotificationSink bean is set, a FIRE_FAILED notification is fired on a separate transaction.
9. If the task's NotificationSink bean is set, then the various notifications are fired as required.
10. If the EXECUTION_DELAYEDUPDATE option is used for the task, the database will be updated a second time with the next state of the task.
11. The global transaction is committed.

Because all events belonging to a task are executed in a single global transactional context, consider the following points in order to avoid transaction-related errors:

- Each resource participating in the task transaction must be two-phase XA capable.

  This includes the Java Database Connectivity (JDBC) datasource that is configured for the scheduler, any Java Messaging Service (JMS) services used by the MessageTaskInfo objects, and any resources used within any of the UserCalendar, TaskHandler, or NotificationSink beans that have a transaction setting of ″Required″.
- One resource can be single-phase, if last participant support is enabled for the application that created the transaction. Enable last participant support using an assembly tool. You can also enable last participant support through the administrative console. See the topic, ″Last participant support extension settings″ for details.

All unexpected exceptions are logged to the activity log and all events participating in the task's global transaction are rolled back. This includes changes to the task's database record, which force the task to be executed again when the scheduler daemon polls the database during the next poll cycle. The UserCalendar, TaskHandler, and NotificationSink beans can choose not to participate in the global transaction by configuring the bean transaction setting to ″Requires new″.

QOS_ATLEASTONCE

Scheduled tasks that use the QOS_ATLEASTONCE quality of service do not have a single transactional context. In this case, each calendar calculation, event notification and database update occurs in an independent transaction:

1. The context of the application that created the task is applied to the thread.
2. The task's database record is updated with the RUNNING state of the task.
3. UserCalendar, NotificationSink beans are called.
4. The BeanTaskInfo or MessageTaskInfo is started.
5. Result notifications are sent.
6. The database is updated with the next state of the task, if the task has not been changed since the RUNNING state was written.

If a failure happens after the RUNNING state is written to the database and before the result is written, then the task may run more than one time.

When using QOS_ATLEASTONCE, all NotificationSink, UserCalendar and TaskHandler beans must not mandate a transaction (TX_MANDATORY), since there is no global transaction available when the task

runs. The EJB components use ″Required″ or ″Requires new″ container managed transaction or a bean managed transaction.

Transaction behavior when using the Scheduler API methods or WASScheduler MBean operations

All Scheduler interface methods participate in a single global transactional context. If a global transactional context is already present on the thread when the create(), suspend(), resume(), cancel(), and purge() methods are executed, then the existing global transaction is used. Otherwise, a new global transaction begins.

If the method participates in the global transaction of the caller and an unexpected error occurs, then the transaction is marked to roll back. If the exception is a declared exception, then the exception is resubmitted to the caller, and the transaction is left alone for the caller to commit or roll back.

If the method starts its own global transaction and any exception occurs, then the transaction is rolled back, and the exception is resubmitted to the caller.

Scheduler task user authorization:

The scheduler service uses the asynchronous beans deferred start mechanism to propagate Java Platform, Enterprise Edition (Java EE) service context information to a task when it runs. If you plan to secure your application using the JAAS security context of the administrative security mechanism built into WebSphere Application Server, create each task with the correct credentials on the thread.

Tasks run with specified security credentials using the following methods:

- Using the Java Authentication and Authorization Service (JAAS) security context on the thread at the time the task was created. See the topic, Deferred start and security in the Asynchronous beans section of the information center.
- Using the setAuthenticationAlias method on the TaskInfo object.
- Using a specified security identity on a BeanTaskInfo task TaskHandler EJB method.

The scheduler service utilizes the asynchronous beans deferred start mechanism to propagate Java EE service context information to a task when it runs. The amount of service context information that is propagated is controlled by the Service Context settings on the WorkManager configuration object that schedulers reference. For example, security and internationalization service contexts can be enabled. See Using asynchronous beans for details on how to configure the Application Server to propagate these service contexts.

Java Authentication and Authorization Service Security context

If you intend to secure your application using the JAAS security context of the administrative security mechanism built into WebSphere Application Server, create each task with the correct credentials on the thread. Once each task has the correct credentials, you can disable and re-enable administrative security without causing any security problems. If you do not set the security context when the scheduler task is created and you later enable security in the target application, a security exception or error message might display, such as SECJ0053E. You might also see this error if two or more schedulers on different servers are accessing the same tables (a clustered or redundant scheduler) and the security settings are different.

The JAAS security context is not set if any of the follow conditions are true:

- administrative security is disabled.
- Security context policies are disabled on the configured WorkManager for the associated scheduler configuration.
- A credential is not set on the thread. For example, the enterprise bean or servlet that is used to create the scheduled task is not secured, or the task was created with a WASScheduler MBean.

If any of the previously mentioned conditions are true when you create your task and you need to enable security on your application server or application, you must complete the following steps for each task:

1. Find the task using the Scheduler API find or get methods.
2. Cancel the task using the Scheduler.cancel() API.
3. Recreate the task using the Scheduler.create() method with security enabled. Submitting a task that was retrieved from the scheduler using the find or get methods will automatically generate a new task ID.

Security order of precedence

As previously noted, there are three ways of verifying that a task will run with the correct user credentials. In addition, each TaskInfo implementation may have its own way of supplying user information, which may override the standard mechanisms. If multiple methods are used, refer to the following lists to determine which security mechanism is going to be employed.

BeanTaskInfo

1. TaskHandler security identity set on the process() method of the Enterprise Java Bean file
2. Authentication Alias set with the setAuthenticationAlias method on the TaskInfo interface
3. JAAS security context

MessageTaskInfo

1. Authentication Alias set with the setAuthenticationAlias method on the TaskInfo interface
2. The setUsername and setPassword methods on the MessageTaskInfo interface. See the Deprecated features article for more information.

Securing scheduler tasks

Scheduled tasks are protected using application isolation and administrative roles. This topic describes how to secure scheduler tasks.

About this task

If a task is created using a Java Platform, Enterprise Edition (Java EE) server application, only applications with the same name can access those tasks. Tasks created with a WASScheduler MBean using the AdminClient interface or scripting are not part of a J2EE application and have access to all tasks regardless of the application with which they were created. Tasks created with a WASScheduler MBean are only accessible from the WASScheduler MBean API and are not accessible from the Scheduler API.

If the Use Administration Roles attribute is enabled on a scheduler and administrative security is enabled on the Application Server, all Scheduler API methods and WASScheduler MBean API operations enforce access based on the WebSphere Administration Roles. If either of these attributes are disabled, then all API methods are fully accessible by all users.

1. Enable security for all application servers.
2. Manage schedulers.

Scheduler configuration or topology

The scheduler uses a database to persist information concerning which tasks to run and when. Errors might occur when changing the application server topology or when changing the application or server configuration. When you change the configuration or topology, carefully consider how this action affects the scheduler.

Restricting security

If you created tasks with an application server while security is disabled, and you later decide to enable security, then the scheduler might have difficulty running tasks. When you create a task, the security

context of the application thread is automatically stored with the task. If security is not stored with the task (see Scheduler task user authorization), and you later enable security on the server or application where the task is to run, then the following errors might be logged:

```
SECJ0053E: Authorization failed for /UNAUTHENTICATED while invoking (Home)com/ibm/websphere/scheduler
/TaskHandler create:2 securityName: /UNAUTHENTICATED;accessID: UNAUTHENTICATED is not granted any of
the required roles: MySecurityRole
```

Before you enable security on the server or application, determine if any tasks might be adversely affected. If so, use the Scheduler API or WASScheduler MBean to cancel the tasks and recreate them after you configure security.

Application server topology changes

The scheduler stores javax.ejb.HomeHandle objects for TaskHandler, NotificationSink and UserCalendar *homes* when the task is created. When you run the task later, these home handles are reinflated and used to access the Enterprise JavaBeans (EJB) component home. When the home handle references an EJB on a single-server environment, the home handles have affinity to that server. When the home handle references an EJB component on a cluster, then the home handles have affinity to the cluster.

If the application server or the Workload Managed (WLM) cluster that a home handle is referencing is not available, then the scheduler fails to run the task, and the following error is logged:

```
SCHD0063E: A task with ID 123 failed to run on Scheduler MyScheduler (sched/MyScheduler) because of
an exception: {cause of failure}
```

If you upgrade the application server to a cluster, or if the Object Request Broker (ORB) ORB_LISTENER_ADDRESS is not set to a fixed port number (see Configuring Inbound Transports), then the task might also fail, since the information stored within the home handle does not have the appropriate information to find the desired server.

Upgrading to a scheduler cluster

A scheduler cluster (not to be confused with a WLM cluster) is a collection of scheduler configurations on different application servers that share the same Java™ Naming and Directory Interface (JNDI) name, Java Database Connectivity (JDBC) data source and table prefix. If you upgrade a stand-alone scheduler to a clustered scheduler, then the application and any associated resources that the application requires must be available. If this is not the case, the scheduled task fails to run and error messages might be logged:

```
SCHD0103W: The Scheduler MyScheduler (sched/MyScheduler) was unable to run task 123 because the
application or module is unavailable: MyTaskHandlerEJB
```

To avoid issues with application availability and achieve optimal results, use the same servers in a scheduler cluster as those used in a WLM cluster.

Reusing scheduler tables

When changing any topology, moving from development to production environments, or making any configuration changes that make the environment more restrictive, you might get optimal results if you use a different set of scheduler tables. Reusing scheduler tables that have scheduled tasks from previous releases without careful planning might cause problems:

- EJB components running on unexpected application servers.
- Tasks failing to run due to invalid or missing security credentials.
- Tasks failing to run due to invalid or missing Java Platform, Enterprise Edition (Java EE) context information.

Diagnosing such problems is challenging and requires analyzing logs on all servers that have a scheduler installed and configured. When the problem tasks are located, the tasks can be cancelled using the

Scheduler APIs, or the tables can be dropped and recreated.

Scheduler interface

Use the `com.ibm.websphere.scheduler.Scheduler` Java object (in the Java™ Naming and Directory Interface (JNDI) namespace for the scheduler configuration) to find a reference to a scheduler and work with tasks.

A `com.ibm.websphere.scheduler.Scheduler` Java object exists in the JNDI namespace for each scheduler configuration. A reference to a scheduler can be obtained by performing a lookup on the JNDI name; however, the lookup is valid only from the server process where the scheduler instance exists. Once a reference has been obtained, tasks can be created, suspended, cancelled, and so on, if the caller has access to the scheduler instance.

For details, see Interface Scheduler in the API documentation.

Task creation
: The task is created in the persistent store using the global transactional context of the caller, if present. See the topic, “Transactions and schedulers” on page 1676, for more details. Since this is a transactional operation, the task cannot be run or modified from another thread until the current transaction commits.

Task modification
: Tasks that have been created can be modified with the suspend(), resume(), cancel(), and purge() methods. These methods take a Task Identifier string as a parameter, which is generated by the create() method and can be found in the TaskStatus object. If a task is currently running or being modified by another thread, an operation that attempts to modify the state of the task might block on the attempt. Tasks can only be modified by the same application (EAR file) that was used to create the task.

Task execution
: Tasks are run in the thread pool specified by the configuration’s work manager. If multiple schedulers are configured to share the same database tables, the scheduler is clustered and the tasks found in the table can be run on any of the schedulers, whether or not they are in the same server, node, or cell.

Task lookup
: Tasks can be located using the Name property that was assigned at creation time. This is useful when you need to modify a group of tasks and tracking individual task ID’s is not convenient.

TaskInfo interface:

TaskInfo objects contain the information to create a task. Several implementations of this class exist, one for each type of task that can run.

Available TaskInfo implementations include:

BeanTaskInfo
: Calls a stateless session bean.

MessageTaskInfo
: Sends a Java™ Messaging Service (JMS) message to a queue or publishes a message to a topic. For details, see the Interface TaskInfo in the API documentation.

  After a TaskInfo object is created, it can be submitted to the scheduler for task creation by calling the Scheduler.create() method.

For details about the TaskInfo interface, see the API documentation .

TaskHandler interface:

A task handler is a user-defined stateless session bean that is called by tasks created using a BeanTaskInfo object.

A task handler bean uses the following home and remote interfaces, which are defined in the deployment descriptor using an assembly tool, such as Rational Application Developer:

```
com.ibm.websphere.scheduler.TaskHandlerHome
com.ibm.websphere.scheduler.TaskHandler
```

The bean itself needs to implement the process() method defined in the remote interface. For details, see the Interface TaskHandler in the API documentation.

Once an EJB is created and available within an enterprise application, it can be called by a BeanTaskInfo task when it runs. See the Developing a task that calls a session bean topic for details.

When a task is created using a BeanTaskInfo object, the process() method on the TaskHandler session bean is called whenever the task runs. Because the TaskStatus object for the task is passed as a parameter to the process() method, the task handler determines different types of information about the task, such as when it will fire next, the number of repeats remaining, its name and its ID.

The process() method can also change its own state. However, the task must be running within the same transaction as the scheduler. Therefore, a running task can only modify itself if it is using the **Required** or **Mandatory** container managed transaction types. If the **Requires New** transaction type is specified on the process()method, all management functions deadlock.

NotificationSink interface:

A notification sink is a user-defined stateless session bean that is called when the task changes state.

A notification sink bean uses the following home and remote interfaces, which are defined in the deployment descriptor using an assembly tool, such as Rational Application Developer:

```
com.ibm.websphere.scheduler.NotificationSinkHome
com.ibm.websphere.scheduler.NotificationSink
```

The bean itself needs to implement the handleEvent() method defined in the remote interface. For details, see the Interface NotificationSink section of the API documentation and the Receiving scheduler notifications topic.

A NotificationSink provides an event notification callback on a task-by-task basis. A notification sink is set on the TaskInfo interface, using the setNotificationSink() method. If a notification sink is not specified on a task, all notifications are lost; however, the status of a task can be determined by calling the getStatus() method from the Scheduler interface. A notification callback is made for each of the following events:

- Scheduled
- Suspended
- Resumed
- Fired
- Firing
- Fire Failed
- Complete
- Purged

UserCalendar interface:

A user calendar is a user-defined stateless session bean that is called by tasks when they need to calculate date-related values.

A user calendar bean uses the following home and remote interfaces, which are defined in the deployment descriptor using an assembly tool, such as Rational Application Developer:

```
com.ibm.websphere.scheduler.UserCalendarHome
com.ibm.websphere.scheduler.UserCalendar
```

The bean itself needs to implement the applyDelta(), validate(), and getCalendarNames() methods defined in the remote interface. For details, see the Interface UserCalendar in the API documentation.

User calendars are used to calculate time intervals, such as the time between task runs. A user calendar takes a java.util.Date object, applies the interval string and returns the resulting java.util.Date.

User calendars are set with the setUserCalendar() method on the TaskInfo interface and called by the scheduler run-time code when a delta calculation is necessary.

The following methods on the TaskInfo interface specify delta strings that use the user calendar for calculation:

- setStartTimeInterval
- setStartByInterval
- setRepeatInterval

Default user calendar

If a user calendar has not been specified using the TaskInfo.setUserCalendar() method, a default user calendar is used. The default calendar allows for simple delta specifications, such as seconds, minutes, hours, days, and months. See the API documentation for details on the default calendar. The default user calendar also provides a CRON-like syntax for calculating absolute times versus time deltas.

Calendar identifiers

A single user calendar can contain logic for multiple calendars. A calendar specifier string determines which calendar is used. For example, a calendar bean might be implemented to recognize the interval *day*. However, the identifier also recognizes two calendar implementations: *standard* (for a standard calendar day) and *business* (for a business day).

Internationalization and time zones

Scheduler makes use of the java.util.Date class when storing and processing dates. Internally, this class saves the time as milliseconds since the Epoch, Greenwich Mean Time. Since the Date is not converted to local time until converted to a string, the scheduler respects the time zone where the date was created.

Writing user calendars

Because the user calendar is a stateless session bean, the same Java Platform, Enterprise Edition (Java EE) programming model available to other session beans is available to the user calendar as well.

Startup beans

Using startup beans

There are two types of startup beans: application startup beans and Module startup beans.

About this task

A module startup bean is a session bean that is loaded when an EJB Jar file starts. Module startup beans enable Java Platform Enterprise Edition (Java EE) applications to run business logic automatically, whenever an EJB module starts or stops normally. An application startup bean is a session bean that is loaded when an application starts. Application startup beans enable Java EE applications to run business logic automatically, whenever an application starts or stops normally.

Startup beans are especially useful when used with asynchronous bean features. For example, a startup bean might create an alarm object that uses the Java Message Service (JMS) to periodically publish heartbeat messages on a well-known topic. This enables clients or other server applications to determine whether the application is available. Refer to the Enabling an application to wait for a messaging engine to start article if you are using the default JMS provider.

1. For Application startup beans, use the home interface, com.ibm.websphere.startupservice.AppStartUpHome, to designate a bean as an Application startup

bean. For Module startup beans, use the home interface, com.ibm.websphere.startupservice.ModStartUpHome, to designate a bean as a Module startup bean.

2. For Application startup beans, use the remote interface, com.ibm.websphere.startupservice.AppStartUp, to define start() and stop() methods on the bean. For Module startup beans, use the remote interface, com.ibm.websphere.startupservice.ModStartUp, to define start() and stop() methods on the bean.

   The startup bean start() method is called when the module or application starts and contains business logic to be run at module or application start time.

   The start() method returns a boolean value. **True** indicates that the business logic within the start() method ran successfully. Conversely, **False** indicates that the business logic within the start() method failed to run completely. A return value of False also indicates to the Application server that application startup is aborted.

   The startup bean stop() methods are called when the module or application stops and contains business logic to be run at module or application stop time. Any exception thrown by a stop() method is logged only. No other action is taken.

   The start() and stop() methods must never use the TX_MANDATORY transaction attribute. A global transaction does not exist on the thread when the start() or stop() methods are invoked. Any other TX_* attribute can be used. If TX_MANDATORY is used, an exception is logged, and the application start is aborted.

   The start() and stop() methods on the remote interface use **Run-As** mode. **Run-As** mode specifies the credential information to be used by the security service to determine the permissions that a principal has on various resources. If security is on, the **Run-As** mode needs to be defined on all of the methods called. The identity of the bean without this setting is undefined.

   There are no restrictions on what code the start() and stop() methods can run, since the full Application Server programming model is available to these methods.

3. Use an *optional* environment property integer, `wasStartupPriority`, to specify the start order of multiple startup beans in the same Java Archive (JAR) file. If the environment property is found and is the wrong type, application startup is aborted. If no priority value is specified, a default priority of `0` is used. It is recommended that you specify the priority property. Beans that have specified a priority are sorted using this property. Beans with numerically lower priorities are run first. Beans that have the same priority are run in an undefined order. All priorities must be positive integers. Beans are stopped in the opposite order to their start priority. The priority values for module startup beans and application startup beans are mutually exclusive. All modules will be started prior to the application being declared as ″started″ and therefore the start() methods for module startup beans within an application will be invoked prior to the start() methods for any application startup beans. Likewise, all application startup bean stop() methods for a specific Java Archive (JAR) file will be invoked prior to any module startup bean stop() methods for that JAR.
4. For module startup beans, the order in which EJB modules are started can be adjusted via the ″Starting weight″ value associated with each module
5. To control who can invoke startup bean methods via WebSphere Security do the following:
   a. Define the method permissions for the Start() and Stop() methods as you would for any EJB module. (See ″Defining method permissions for EJB modules″.)
   b. Ensure that the user that is mapped to the Security Role defined for the startup bean methods is the same user that is defined as the Server user ID within the User Registry.

What to do next

View the startup beans service settings.

Startup beans service settings

Use this page to enable startup beans that control whether application-defined startup beans function on this server. Startup beans are session beans that run business logic through the invocation of start and stop methods when applications start and stop. If the startup beans service is disabled, then the automatic

invocation of the start and stop methods does not occur for deployed startup beans when the parent application starts or stops. This service is disabled by default. Enable this service only when you want to use startup beans. Startup beans are especially useful when used with asynchronous beans.

To view this administrative console page, click **Servers** > **Application servers** >*server_name* > **Container services** > **Startup beans service**.

Enable service at server startup:

Specifies whether the server attempts to initiate the startup beans service.

Default
Cleared

Range

Selected
When the application server starts, it attempts to initiate the startup bean service automatically.

Cleared
The server does not try to initiate the startup beans service. All startup beans do not start or stop with the application. If you use startup beans on this server, then the system administrator must start the startup beans service manually or select this property, and then restart the server.

Work area

Task overview: Implementing shared work areas

About this task

The work area service enables application developers to implicitly propagate information beyond the information passed in remote calls. Applications can create a work area, insert information into it, and make remote invocations. The work area is propagated with each remote method invocation, eliminating the need to explicitly include an appropriate argument in the definition of each method. The methods on the server side can use or ignore the information in the work area as appropriate.

Before proceeding with the steps to implement work areas, as described below, review the topic Work area service: Overview.

1. Developing applications that use work areas. Applications interact with the work area service by implementing the UserWorkArea interface.
2. Managing work areas. The work area service is managed using the administrative console.
3. Configuring work area partitions You can create multiple work areas with additional configuration options than the UserWorkArea partition.
4. Propagating work area context over Web services You can propagate work area context on a Web service call instead of an RMI/IIOP call.

Overview of work area service

The work area service passes information explicitly as an argument or implicitly to remote methods.

One of the foundations of distributed computing is the ability to pass information, typically in the form of arguments to remote methods, from one process to another. When application-level software is written over middleware services, many of the services rely on information beyond that passed in the application's remote calls. Such services often make use of the implicit propagation of private information in addition to the arguments passed in remote requests; two typical users of such a feature are security and transaction services. Security certificates or transaction contexts are passed without the knowledge or intervention of the user or application developer. The implicit propagation of such information means that application

developers do not have to manually pass the information in method invocations, which makes development less error-prone, and the services requiring the information do not have to expose it to application developers. Information such as security credentials can remain secret.

The work area service gives application developers a similar facility. Applications can create a work area, insert information into it, and make remote invocations. The work area is propagated with each remote method invocation, eliminating the need to explicitly include an appropriate argument in the definition of every method. The methods on the server side can use or ignore the information in the work area as appropriate. If methods in a server receive a work area from a client and subsequently invoke other remote methods, the work area is transparently propagated with the remote requests. When the creating application is done with the work area, it terminates it.

There are two prime considerations in deciding whether to pass information explicitly as an argument or implicitly by using a work area. These considerations are:
- Pervasiveness: Is the information used in a majority of the methods in an application?
- Size: Is it reasonable to send the information even when it is not used?

When information is sufficiently pervasive that it is easiest and most efficient to make it available everywhere, application programmers can use the work area service to simplify programming and maintenance of code. The argument does not need to go onto every argument list. It is much easier to put the value into a work area and propagate it automatically. This is especially true for methods that simply pass the value on but do nothing with it. Methods that make no use of the propagated information simply ignore it.

Work areas can hold any kind of information, and they can hold an arbitrary number of individual pieces of data, each stored as a property.

Use the work area service in the administrative console to configure the UserWorkArea partition. The UserWorkArea partition is the partition that is available in JNDI naming under the name ″java:comp/websphere/UserWorkArea″ as demonstrated in the article, Accessing the UserWorkArea partition. The UserWorkArea partition is the default work area partition created automatically, if it has not been disabled, and is available through JNDI naming to all users. Any configuration option made to the UserWorkArea partition under the work area service panel in the administrative console does not affect the work area partition service or any partitions defined in it, and conversely. For example, if you select the enable or disable option in the work area service panel, this does not affect the work area partition service or any partition within it.

Work area property modes:

The information in a work area consists of a set of properties; a property consists of a key-value-mode triple. The key-value pair represents the information contained in the property; the key is a name by which the associated value is retrieved. The mode determines whether you can modify or remove the property.

Property modes

There are four possible mode values for properties, as shown in the following code example:

Code example: The PropertyModeType definition

```
public final class PropertyModeType {
   public static final PropertyModeType normal;
   public static final PropertyModeType read_only;
   public static final PropertyModeType fixed_normal;
   public static final PropertyModeType fixed_readonly;
};
```

A property's mode determines three things:
- Whether the value associated with the key can be modified

- Whether the property can be deleted
- Whether the mode associated with the key-value pair can be modified

The two read-only modes forbid changes to the information in the property; the two fixed modes forbid deletion of the property.

The work area service does not provide methods specifically for the purpose of modifying the value of a key or the mode associated with a property. To change information in a property, applications simply rewrite the information in the property; this has the same effect as updating the information in the property. The mode of a property governs the changes that can be made. Modifying key-value pairs describes the restrictions each mode places on modifying the value and deleting the property. Changing modes describes the restrictions on changing the mode.

Changing modes

The mode associated with a property can be changed only according to the restrictions of the original mode. The read-only and fixed read-only properties do not permit modification of the value or the mode. The fixed normal and fixed read-only modes do not allow the property to be deleted. This set of restrictions leads to the following permissible ways to change the mode of a property within the lifetime of a work area:

- If the current mode is normal, it can be changed to any of the other three modes: fixed normal, read-only, fixed read-only.
- If the current mode is fixed normal, it can be changed only to fixed read-only.
- If the current mode is read-only, it can be changed only by deleting the property and re-creating it with the desired mode.
- If the current mode is fixed read-only, it cannot be changed.
- If the current mode is not normal, it cannot be changed to normal. If a property is set as fixed normal and then reset as normal, the value is updated but the mode remains fixed normal. If a property is set as fixed normal and then reset as either read-only or fixed read-only, the value is updated and the mode is changed to fixed read-only.

Note: The key, value, and mode of any property can be effectively changed by terminating (completing) the work area in which the property was created and creating a new work area. Applications can then insert new properties into the work area. This is not precisely the same as changing the value in the original work area, but some applications can use it as an equivalent mechanism.

Nested work areas:

Applications can nest work areas to define and scope properties for specific tasks without having to make the work areas available to all parts of the application.

When an application creates a work area, a work area context is associated with the creating thread. If the application thread creates another work area, the new work area is nested within the existing work area and becomes the current work area. All properties defined in the original, enclosing work area are visible to the nested work area. The application can set additional properties within the nested work area that are not part of the enclosing work area.

An application working with a nested work area does not actually see the nesting of enclosing work areas. The current work area appears as a flat set of properties that includes those from enclosing work areas. In the figure below, the enclosing work area holds several properties and the nested work area holds additional properties. From the outermost work area, the properties set in the nested work area are not visible. From the nested work area, the properties in both work areas are visible.

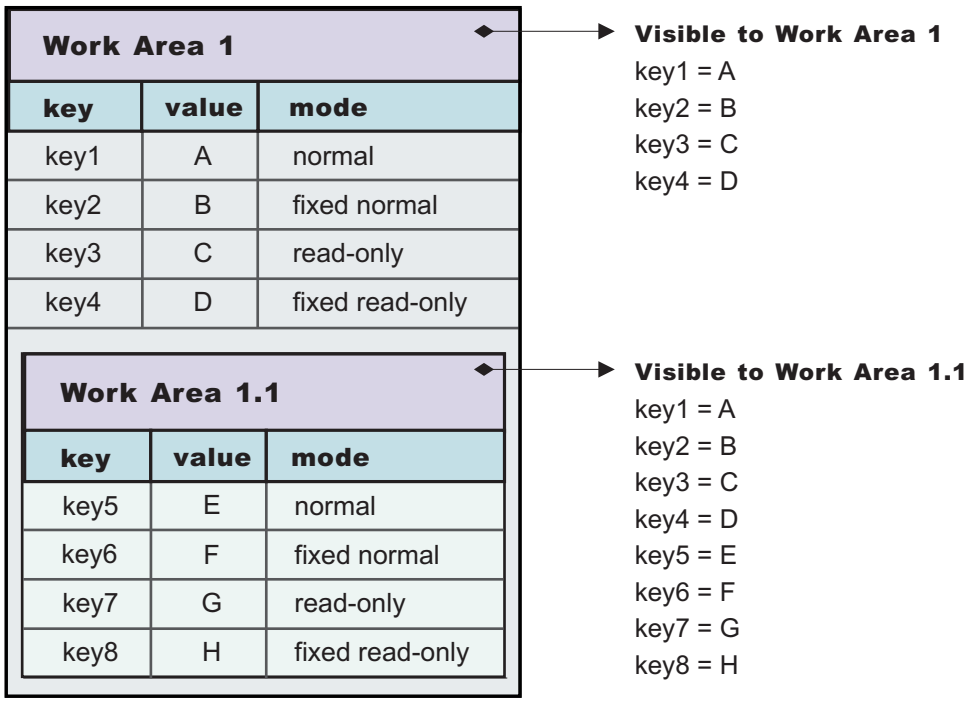


Figure 18. Defining new properties in nested work areas

Nesting can also affect the apparent settings of the properties. Properties can be deleted from or directly modified only within the work areas in which they were set, but nested work areas can also be used to temporarily override information in the property without having to modify the property. Depending on the modes associated with the properties in the enclosing work area, the modes and the values of keys in the enclosing work area can be overridden within the nested work area.

The mode associated with a property when it is created determines whether nested work areas can override the property. From the perspective of a nested work area, the property modes used in enclosing work areas can be grouped as follows:

- Modes that permit a nested work area to override the mode or the value of a key locally. The modes that permit overriding are:
  - Normal
  - Fixed normal
- Modes that do not permit a nested work area to override the mode or the value of a key locally. The modes that do not permit overriding are:
  - Read-only
  - Fixed read-only

If an enclosing work area defines a property with one of the modes that can be overridden, a nested work area can specify a new value for the key or a new mode for the property. The new value or mode becomes the value or mode seen by subsequently nested work areas. Changes to the mode are governed by the restrictions described in Changing modes. If an enclosing work area defines a property with one of the modes that cannot be overridden, no nested work area can specify a new value for the key.

A nested work area can delete properties from enclosing work areas, but the changes persist only for the duration of the nested work area. When the nested work area is completed, any properties that were added in the nested area vanish and any properties that were deleted from the nested area are restored.

The following figure illustrates the overriding of properties from an enclosing work area. The nested work area redefines two of the properties set in the enclosing work area. The other two cannot be overridden. The nested work area also defines two new properties. From the outermost work area, the properties set

or redefined in the nested work are not visible. From the nested work area, the properties in both work areas are visible, but the values seen for the redefined properties are those set in the nested work area.

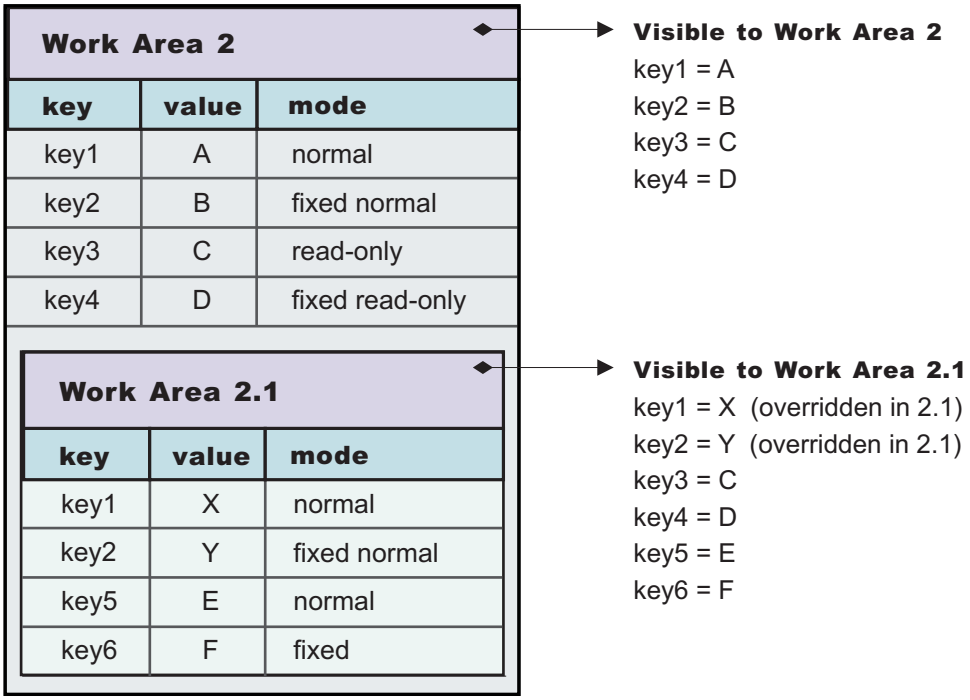


Figure 19. Redefining existing properties in nested work areas

Distributed work areas:

Work area context propagates to a target object on a remote invocation on both bidirectional and non-bidirectional defined work area partitions. The propagation of work area context operates differently depending on whether a work area partition is defined as bidirectional. If the partition is defined as bidirectional, the context propagates from a target object back to the originator.

Non-bidirectional work area partitions (UserWorkArea partition)

If a remote invocation is issued from a thread associated with a work area, a copy of the work area is automatically propagated to the target object, which can use or ignore the information in the work area as necessary. If the calling application has a nested work area associated with it, a copy of the nested work area and all its ancestors is propagated to the target. The target application can locally modify the information, as allowed by the property modes, by creating additional nested work areas; this information is propagated to any remote objects it invokes. However, no changes made to a nested work area on a target object are propagated back to the calling object. The caller's work area is unaffected by changes made in the remote method.

Bidirectional work area partitions

If a remote invocation is issued from a thread associated with a work area, a copy of the work area is automatically propagated to the target object, which can use or ignore the information in the work area as necessary. If the calling application has a nested work area associated with it, a copy of the nested work area and all its ancestors is propagated to the target. The target application can locally modify the information, as allowed by the property modes, this information is propagated to any remote objects it invokes. In a partition that is not defined as bidirectional, a target application must begin a nested work area before making changes to the imported work area. However, if a partition is defined as bidirectional, a target application need not begin a nested work area before operating on an imported work area. By not

beginning a nested work area, any new context set into the work area, or any context changes made by the target application, is not only propagated on future remote invocations but is also propagated back to the originating application (that is, the one who initiated the remote invocation) thus allowing bidirectional propagation of work area context. If the target application does not want new or changed context to propagate back to the originating application, then the target application must begin a nested work area to scope the context to its process. However, the new or changed context in the nested work area propagates on any future remote invocation the target application may make.

WorkArea service: Special considerations:

Developers who use work areas should consider the following issues that could potentially cause problems: interoperability between the EJB and CORBA programming models; and the use of work areas with Java's Abstract Windowing Toolkit.

EJB and CORBA interoperability

Although the work area service can be used across the EJB and CORBA programming models, many composed data types cannot be successfully used across those boundaries. For example, if a SimpleSampleCompany instance is passed from the WebSphere environment into a CORBA environment, the CORBA application can retrieve the SimpleSampleCompany object encapsulated within a CORBA Any object from the work area, but it cannot extract the value from it. Likewise, an IDL-defined struct defined within a CORBA application and set into a work area is not readable by an application using the UserWorkArea class.

Note: Applications can avoid this incompatibility by directly setting only primitive types, like integers and strings, as values in work areas, or by implementing complex values with structures designed to be compatible, like CORBA valuetypes.

Also, CORBA Anys that contains either the tk_null or tk_void typecode can be set into the work area by using the CORBA interface. However, the work area specification cannot allow the Java 2 Platform, Enterprise Edition (J2EE) implementation to return null on a lookup that retrieves these CORBA-set properties without incorrectly implying that there is no value set for the corresponding key. For example, when a user attempts to retrieve a nonexistent key from a work area, the work area service returns null to indicate that the specified key does not contain a value, implying that the key itself is not in use or does not exist. In the case where CORBA Anys contains either tk_null or tk_void, when a user requests the key associated with one of these values, the work area service returns null as expected. In this case, the key may actually exist and the work area service was simply returning the key's value of null. Therefore, when working with CORBA Anys, a user must not make any implications when a null is returned from a work area because it could mean that either there isn't a property associated with the given key, or that there is a property associated with the given key and it contains a tk_null or tk_void, for example, a null in the J2EE environment. If a J2EE application tries to retrieve CORBA-set properties that are non-serializable, or contain CORBA nulls or void references, the com.ibm.websphere.workarea.IncompatibleValue exception is raised.

Using work areas with Java's Abstract Windowing Toolkit (AWT)

Work areas must be used cautiously in applications that use Java's Abstract Windowing Toolkit (AWT). The AWT implementation is multithreaded, and work areas begun on one thread are not available on another. For example, if a program begins a work area in response to an AWT event, such as pressing a button, the work area might not be available to any other part of the application after the execution of the event completes.

Work area service performance considerations: The work area service is designed to address complex data passing patterns that can quickly grow beyond convenient maintenance. A *work area* is a

note pad that is accessible to any client that is capable of looking up Java Naming Directory Interface (JNDI). After a work area is established, data can be placed there for future use in any subsequent method calls to both remote and local resources.

You can utilize a work area when a large number of methods require common information or if information is only needed by a method that is significantly further down the call graph. The former avoids the need for complex parameter passing models where the number of arguments passed becomes excessive and hard to maintain. You can improve application function by placing the information in a work area and subsequently accessing it independently in each method, eliminating the need to pass these parameters from method to method. The latter case also avoids unnecessary parameter passing and helps to improve performance by reducing the cost of marshalling and de-marshalling these parameters over the Object Request Broker (ORB) when they are only needed occasionally throughout the call graph.

When attempting to maximize performance by using a work area, cache the UserWorkArea partition that is retrieved from JNDI wherever it is accessed. You can reduce the time spent looking up information in JNDI by retrieving it once and keeping a reference for the future. JNDI lookup takes time and can be costly.

Additional caching mechanisms available to a user-defined partition are defined by the configuration property, ″Deferred Attribute Serialization″. This mechanism attempts to minimize the number of serialization and deserialization calls. See Work area partition service for further explanation of this configuration attribute.

The maxSendSize and maxReceiveSize configuration parameters can affect the performance of the work area. Setting these two values to 0 (zero) effectively turns off the policing of the size of context that can be sent in a work area. This action can enhance performance, depending on the number of nested work areas an application uses. In applications that use only one work area, the performance enhancement might be negligible. In applications that have a large number of nested work areas, there might be a performance enhancement. However, a user must note that by turning off this policing it is possible that an extremely large amount of data might be sent to a server.

Performance is degraded if you use a work area as a direct replacement to passing a single parameter over a single method call. The reason is that you incur more overhead than just passing that parameter between method calls. Although the degradation is usually within acceptable tolerances and scales similarly to passing parameters with regard to object size, consider degradation a potential problem before utilizing the service. As with most functional services, intelligent use of the work areas yields the best results.

The work area service is a tool to simplify the job of passing information from resource to resource, and in some cases can improve performance by reducing the overhead that is associated with a parameter passing when the information is only sparsely accessed within the call graph. Caching the instance retrieved from JNDI is important to effectively maximize performance during runtime.

Developing applications that use work areas

About this task

Applications interact with the work area service by implementing the UserWorkArea interface. This interface, shown below, defines all of the methods used to create, manipulate, and terminate work areas.

```
package com.ibm.websphere.workarea;

public interface UserWorkArea {
   void begin(String name);
   void complete() throws NoWorkArea, NotOriginator;

   String getName();
   String[] retrieveAllKeys();
   void set(String key, java.io.Serializable value)
      throws NoWorkArea, NotOriginator, PropertyReadOnly;
```

```
   void set(String key, java.io.Serializable value, PropertyModeType mode)
      throws NoWorkArea, NotOriginator, PropertyReadOnly;
   java.io.Serializable get(String key);
   PropertyModeType getMode(String key);
   void remove(String key)
      throws NoWorkArea, NotOriginator, PropertyFixed;
}
```

Note: Enterprise JavaBeans (EJB) applications can use the UserWorkArea interface only within the implementation of methods in either the remote or local interface, or both; likewise, servlets can use the interface only within the service method of the HTTPServlet class. Use of work areas within any life cycle method of a servlet or enterprise bean is considered a deviation from the work area programming model and is not supported.

The work area service defines the following exceptions for use with the UserWorkArea interface:

NoWorkArea
Raised when a request requires an associated work area but none is present.

NotOriginator
Raised when a request attempts to manipulate the contents of an imported work area.

PropertyReadOnly
Raised when a request attempts to modify a read-only or fixed read-only property.

PropertyFixed
Raised by the remove method when the designated property has one of the fixed modes.

1. Access a partition by either:
   - Accessing the UserWorkArea partition, to access the UserWorkArea partition.
   - Accessing a user defined work area partition, to access a user defined work area.

   The following steps use the UserWorkArea partition as an example; however, you can use a user defined partition in the same way.
2. Begin a new work area. Use the begin method to create a new work area and associate it with the calling thread. A work area is scoped to the thread that began the work area and is not accessible by multiple threads. The begin method takes a string as an argument; the string is used to name the work area. The argument must not be null, which causes the java.lang.NullPointer exception to be raised. In the following code example, the application begins a new work area with the name SimpleSampleServlet:

   ```
   public class SimpleSampleServlet {
   ...
      try {
         ...
         userWorkArea = (UserWorkArea)jndi.lookup(
            "java:comp/websphere/UserWorkArea");
      }
      ...

      userWorkArea.begin("SimpleSampleServlet");
      ...
   }
   ```

   The begin method is also used to create nested work areas; if a work area is associated with a thread when the begin method is called, the method creates a new work area nested within the existing work area.

   The work area service makes no use of the names associated with work areas; You can name work areas in any way that you choose. Names are not required to be unique, but the usefulness of the names for debugging is enhanced if the names are distinct and meaningful within the application. Applications can use the getName method to return the name associated with a work area by the begin method.
3. Set properties in the work area. An application with a current work area can insert properties into the work area and retrieve the properties from the work area. The UserWorkArea interface provides two

set methods for setting properties and a get method for retrieving properties. The two-argument set method inserts the property with the property mode of normal. The three-argument set method takes a property mode as the third argument. Both set methods take the key and the value as arguments. The key is a String; the value is an object of the type java.io.Serializable. None of the arguments can be null, which causes the java.lang.NullPointer exception to be raised. The SimpleSample application below uses objects of two classes, the SimpleSampleCompany class and the SimpleSampleProperty class, as values for properties. The SimpleSampleCompany class is used for the site identifier, and the SimpleSamplePriority class is used for the priority. These classes are shown in following code example:

```
public class SimpleSampleServlet {
   ...
   userWorkArea.begin("SimpleSampleServlet");

   try {
      // Set the site-identifier (default is Main).
      userWorkArea.set("company",
         SimpleSampleCompany.Main, PropertyModeType.read_only);

      // Set the priority.
      userWorkArea.set("priority", SimpleSamplePriority.Silver);
   }

   catch (PropertyReadOnly e) {
     // The company was previously set with the read-only or
     // fixed read-only mode.
     ...
   }

    catch (NotOriginator e) {
     // The work area originated in another process,
     // so it can't be modified here.
     ...
   }

   catch (NoWorkArea e) {
      // There is no work area begun on this thread.
      ...
   }

   // Do application work.
   ...
}
```

The get method takes the key as an argument and returns a Java Serializable object as the value associated with the key. For example, to retrieve the value of the company key from the work area, the code example above uses the get method on the work area to retrieve the value.

Setting property modes. The two-argument set method on the UserWorkArea interface takes a key and a value as arguments and inserts the property with the default property mode of normal. To set a property with a different mode, applications must use the three-argument set method, which takes a property mode as the third argument. The values used to request the property modes are as follows:

- **Normal**: PropertyModeType.normal
- **Fixed normal**: PropertyModeType.fixed_normal
- **Read-only**: PropertyModeType.read_only
- **Fixed read-only**: PropertyModeType.fixed_readonly

4. “Managing local work with a work area” on page 1695.
5. Complete the work area. After an application has finished using a work area, it must complete the work area by calling the complete method on the UserWorkArea interface. This terminates the association with the calling thread and destroys the work area. If the complete method is called on a nested work area, the nested work area is terminated and the parent work area becomes the current work area. If there is no work area associated with the calling thread, a NoWorkArea exception is created. Every work area must be completed, and work areas can be completed only by the originating process. For

example, if a server attempts to call the complete method on a work area that originated in a client, a NotOriginator exception is created. Work areas created in a server process are never propagated back to an invoking client process.

Note: The work area service claims full local-remote transparency. Even if two beans happen to be deployed in the same server, and therefore the same JVM and process, a work area begun on an invocation from another is completed and the bean in which the request originated is always in the same state after any remote call.

The following code example shows the completion of the work area created in the client application.

```
public class SimpleSampleServlet {
   ...
   userWorkArea.begin("SimpleSampleServlet");
   userWorkArea.set("company",
       SimpleSampleCompany.Main, PropertyModeType.read_only);
   userWorkArea.set("priority", SimpleSamplePriority.Silver);
   ...

   // Do application work.
   ...

   // Terminate the work area.
   try {
      userWorkArea.complete();
   }

   catch (NoWorkArea e) {
      // There is no work area associated with this thread.
      ...
   }

   catch (NotOriginator e) {
      // The work area was imported into this process.
      ...
   }
  ...
}
```

The following code example shows the sample application completing the nested work area it created earlier in the remote invocation.

```
public class SimpleSampleBeanImpl implements SessionBean {

    public String [] test() {
      ...

      // Begin a nested work area.
      userWorkArea.begin("SimpleSampleBean");
      try {
        userWorkArea.set("company",
                         SimpleSampleCompany.London_Development);
      }
      catch (NotOriginator e) {
      }

      SimpleSampleCompany company =
         (SimpleSampleCompany) userWorkArea.get("company");
      SimpleSamplePriority priority =
         (SimpleSamplePriority) userWorkArea.get("priority");

      // Complete all nested work areas before returning.
      try {
        userWorkArea.complete();
      }
      catch (NoWorkArea e) {
      }
```

```
            catch (NotOriginator e) {
            }
        }
    }
```

Example

In the following example, the client creates a work area and inserts two properties into the work area: a site identifier and a priority. The site-identifier is set as a read-only property; the client does not allow recipients of the work area to override the site identifier. This property consists of the key company and a static instance of a SimpleSampleCompany object. The priority property consists of the key priority and a static instance of a SimpleSamplePriority object. The object types are defined as shown in the following code example:

```
public static final class SimpleSampleCompany {
   public static final SimpleSampleCompany Main;
   public static final SimpleSampleCompany NewYork_Sales;
   public static final SimpleSampleCompany NewYork_Development;
   public static final SimpleSampleCompany London_Sales;
   public static final SimpleSampleCompany London_Development;
}

public static final class SimpleSamplePriority {
   public static final SimpleSamplePriority Platinum;
   public static final SimpleSamplePriority Gold;
   public static final SimpleSamplePriority Silver;
   public static final SimpleSamplePriority Bronze;
   public static final SimpleSamplePriority Tin;
}
```

The client then makes an invocation on a remote object. The work area is automatically propagated; none of the methods on the remote object take a work area argument. On the remote side, the request is first handled by the SimpleSampleBean; the bean first reads the site identifier and priority properties from the work area. The bean then intentionally attempts, and fails, both to write directly into the imported work area and to override the read-only site-identifier property.

The SimpleSampleBean successfully begins a nested work area, in which it overrides the client's priority, then calls another bean, the SimpleSampleBackendBean. The SimpleSampleBackendBean reads the properties from the work area, which contains the site identifier set in the client and priority set in the SimpleSampleBean. Finally, the SimpleSampleBean completes its nested work area, writes out a message based on the site-identifier property, and returns.

What to do next

For additional information about work area, see the com.ibm.websphere.workarea package in the API. The generated API documentation is available in the information center table of contents from the path **Reference → APIs - Application programming interfaces**.

Managing local work with a work area

Before you begin

Be sure that your client has a reference to the UserWorkArea interface, as described in the topic Accessing the UserWorkArea partition or a reference to a user defined partition as defined in Accessing a user defined work area partition. The following steps use the UserWorkArea partition as an illustration. However a user defined partition can be used in the exact same way.

About this task

In a business application that uses work areas, server objects typically retrieve the work area properties and use them to guide local work.

1. Retrieve the name of the active work area to determine whether the calling thread is associated with a work area.

   Applications use the getName method on the UserWorkArea interface to retrieve the name of the current work area. If the thread is not associated with a work area, the getName method returns null. In the following code example, the name of the work area corresponds to the name of the class in which the work area was begun.

```
public class SimpleSampleBeanImpl implements SessionBean {

    ...

    public String [] test() {
       // Get the work-area reference from JNDI.
       ...

       // Retrieve the name of the work area. In this example,
       // the name is used to identify the class in which the
       // work area was begun.
       String invoker = userWorkArea.getName();
       ...
   }
}
```

2. Overriding work area properties. Server objects can override client work area properties by creating their own, nested work area.
3. Retrieve properties from a work area by using the get method.

   The get method is intentionally lightweight; there are no declared exceptions to handle. If there is no active work area, or if there is no such property set in the current work area, the get method returns null.

   Note: The get method can raise a NotSerializableError in the relatively rare scenario in which CORBA clients set composed data types and invoke enterprise-bean interfaces.

   The following example shows the retrieval of the site-identifier and priority properties by the SimpleSampleBean. Notice that one property was set into an outer work area by the client and the other property was set into the nested work area by the server-side bean; the nesting is transparent to the retrieval of the properties.

```
public class SimpleSampleBeanImpl implements SessionBean {

    public String [] test() {
      ...

      // Begin a nested work area.
      userWorkArea.begin("SimpleSampleBean");
      try {
        userWorkArea.set("company",
                         SimpleSampleCompany.London_Development);
      }
      catch (NotOriginator e) {
      }

      SimpleSampleCompany company =
         (SimpleSampleCompany) userWorkArea.get("company");
      SimpleSamplePriority priority =
         (SimpleSamplePriority) userWorkArea.get("priority");
       ...
   }
}
```

4. Optional: Retrieve a list of all the keys visible from a work area.

The UserWorkArea interface provides the retrieveAllKeys method for retrieving a list of all the keys visible from a work area. This method takes no arguments and returns an array of strings. The retrieveAllKeys method returns null if there is no work area associated with the thread. If there is an associated work area that does not contain any properties, the method returns an array of size 0.

5. Query the mode of a work area property using the getMode method.

   The UserWorkArea interface provides the getMode method determine the mode of a specific property. This method takes the property's key as an argument and returns the mode as a PropertyModeType object. If the specified key does not exist in the work area, the method returns PropertyModeType.normal, indicating that the property can be set and removed without error.

6. Optional: Delete a work area property.

   The UserWorkArea interface provides the remove method to delete a property from the current scope of a work area. If the property was initially set in the current scope, removing it deletes the property. If the property was initially set in an enclosing work area, removing it deletes the property until the current scope is completed. When the current work area is completed, the deleted property is restored.

   The remove method takes the property's key as an argument. Only properties with the modes normal and read-only can be removed. Attempting to remove a fixed property creates the PropertyFixed exception. Attempting to remove properties in work areas that originated in other processes creates the NotOriginator exception.

Example

The server side of the SimpleSample application example, which is included in the topic,"Developing applications that use work areas" on page 1691, accepts remote invocations from clients. With each remote call, the server also gets a work area from the client if the client has created one. The work area is propagated transparently. None of the remote methods includes the work area on its argument list.

In the example application, the server objects use the work area interface for demonstration purposes only. For example, the SimpleSampleBean intentionally attempts to write directly to an imported work area, which creates the NotOriginator exception. Likewise, the bean intentionally attempts to mask the read only SimpleSampleCompany, which triggers the PropertyReadOnly exception. The SimpleSampleBean also nests a work area and successfully overrides the priority property before invoking the SimpleSampleBackendBean. A true business application would extract the work area properties and use them to guide the local work. The SimpleSampleBean mimics this by writing a message that function is denied when a request emanates from a sales environment.

Work area service settings

Use this page to manage the work area service.

The work area service manages the scope and implicit propagation of application context. The work area service panel in the administrative console configures the UserWorkArea partition only and has no effect on the work area partition service panel.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Business Process Services** → **Work area service**.

For additional information about work area, see the com.ibm.websphere.workarea package in the Application Programming Interfaces (API) documentation. The generated API documentation is available in the information center table of contents from the path **Reference** → **APIs - Application Programming Interfaces**.

Enable service at server startup:

Specifies whether the server attempts to start the work area service.

Selected
When the application server starts, it attempts to start the work area service automatically.

Cleared
The server does not try to start the work area service. If work areas are used on this application server, the system administrator must start the service manually or select this property and then restart the server.

Maximum send size:

Specifies the maximum size of data that can be sent within a single work area.

Data type	Integer
Units	Bytes
Default	10000
Range	-1, 0 (no limit) and 1 to 2147483647

The following values are also used to define the maximum send size.

-1	Default.
0	No limit.

Maximum receive size:

Specifies the maximum size of data that a single work area can receive.

Data type	Integer
Units	Bytes
Default	10000
Range	-1, 0 (no limit) and 1 to 2147483647

The following values are also used to define the maximum receive size.

-1	Default.
0	No limit.

Enable Web service propagation:

Specifies whether the work area is propagated on Web service requests. This option is disabled by default.

Overriding work area properties
About this task

Work areas are inherently associated with the process that creates them. In the sample application, the client begins a work area and sets into it the site-identifier and priority properties. This work area is propagated to the server when the client makes a remote invocation.

Applications nest work areas in order to temporarily override properties imported from a client process. The nesting mechanism is automatic; invoking begin on the UserWorkArea interface from within the scope of an existing work area creates a nested work area that inherits the properties from the enclosing work area. Properties set into the nested work area are strictly associated with the process in which the work area was begun; the nested work area must be completed within the process that created them. If a work area is not completed by the creating process, the work-area facility terminates the work area when the

process exits. After a nested work area is completed, the original view of the enclosing work area is restored. However, the view of the complete set of work areas associated with a thread cannot be decomposed by downstream processes.

Applications set properties into a work area using property modes in ensure that a particular property is fixed (not removable) or read-only (not overrideable) within the scope of the given work area.

Example

In the following code example, the server-side sample bean attempts to write directly to the imported work area; because the UserWorkArea partition is not defined to be bidirectional, this action is not permitted, and the NotOriginator exception is thrown. When the UserWorkArea partition is not defined as bidirectional, the sample bean must begin its own work area in order to override any imported properties, as shown in the second code example. If a work area in a user defined partition is used and is defined as bidirectional, this bean can set context into the work area before beginning another work area. This context set in the bidirectional case propagates back to the caller. See Work area partition servicefor additional information.

```
public class SimpleSampleBeanImpl implements SessionBean {

   public String [] test() {
     ...
     String invoker = userWorkArea.getName();

     try {
       userWorkArea.set("key", "value");
     }
     catch (NotOriginator e) {
     }
     ...
  }
}
```

The following code example demonstrates beginning a nested work area, using the name of the creating class to identify the nested work area.

```
public class SimpleSampleBeanImpl implements SessionBean {

   public String [] test() {
      ...
      String invoker = userWorkArea.getName();
      try {
        userWorkArea.set("key", "value");
      }
      catch (NotOriginator e) {
      }

      // Begin a nested work area. By using the name of the creating
      // class as the name of the work area, we can avoid having
      // to explicitly set the name of the creating class in
      // the work area.
      userWorkArea.begin("SimpleSampleBean");

      ...
   }
}
```

In the example application, the client sets the site-identifier property as read-only; that guarantees that the request is always associated with the client's company identity. A server cannot override that value in a nested work area. In the following code example, the SimpleSampleBean attempts to change the value of the site-identifier property in the nested work area it created.

```
public class SimpleSampleBeanImpl implements SessionBean {

  public String [] test() {
      ...

      String invoker = userWorkArea.getName();
      try {
        userWorkArea.set("key", "value");
      }
      catch (NotOriginator e) {
      }

      // Begin a nested work area.
      userWorkArea.begin("SimpleSampleBean");

      try {
        userWorkArea.set("company",
                         SimpleSampleCompany.London_Development);
      }
      catch (NotOriginator e) {
      }
      ...
  }
}
```

retrieveAllKeys method

About this task

The UserWorkArea interface provides the retrieveAllKeys method for retrieving a list of all the keys visible from a work area. This method takes no arguments and returns an array of strings. The retrieveAllKeys method returns null if there is no work area associated with the thread. If there is an associated work area that does not contain any properties, the method returns an array of size 0.

For additional information about work area, see the com.ibm.websphere.workarea package in the API documentation. The generated API documentation is available in the information center table of contents from the path **Reference** → **APIs - Application Programming Interfaces**.

Appendix. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories infer specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are was.install.root and WAS_HOME.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv7config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are server.root and user.install.root.

  In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will be always be *app_server_root*/profiles/default because only the profile name ″default″ is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E.

  The corresponding product variable is smpe.install.root.

  The default is /usr/lpp/zWebSphere/V7R0.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.