WebSphere® Application Server Network Deployment for Distributed Platforms,
Version 6.1

IBM

Developing and deploying applications

Compilation date: May 4, 2006

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 919-254-0206.

    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.

Chapter 1. Overview and new features for developing and deploying applications

Use the links provided in this topic to learn more about developing applications for deployment on this product.

"What is new for developers" on page 16

> This topic provides an overview of new and changed features of the programming model and application serving environment as it pertains to development and test efforts.

"Learn about WebSphere applications: Overview and new features"

> This topic provides an overview of the programming model.

"Accessing the Samples (Samples Gallery)" on page 11

> The Samples are a good way to become familiar with the programming model.

Learn about WebSphere applications: Overview and new features

Use the **Learn about WebSphere applications** section as a starting point to study the programing model, encompassing the many parts used in and by various application types supported by the application server.

The programming model for applications deployed on this product has the following aspects.
- Java specifications and other open standards for developing applications
- WebSphere programming model extensions to enhance application functionality
- Containers and services in the application server, used by deployed applications, and which sometimes can be extended

The diagram shows a single application server installation. The parts pertaining to the programming model are discussed here. Other parts comprise the product architecture, independent of the various application types outlined by the programming model.

1
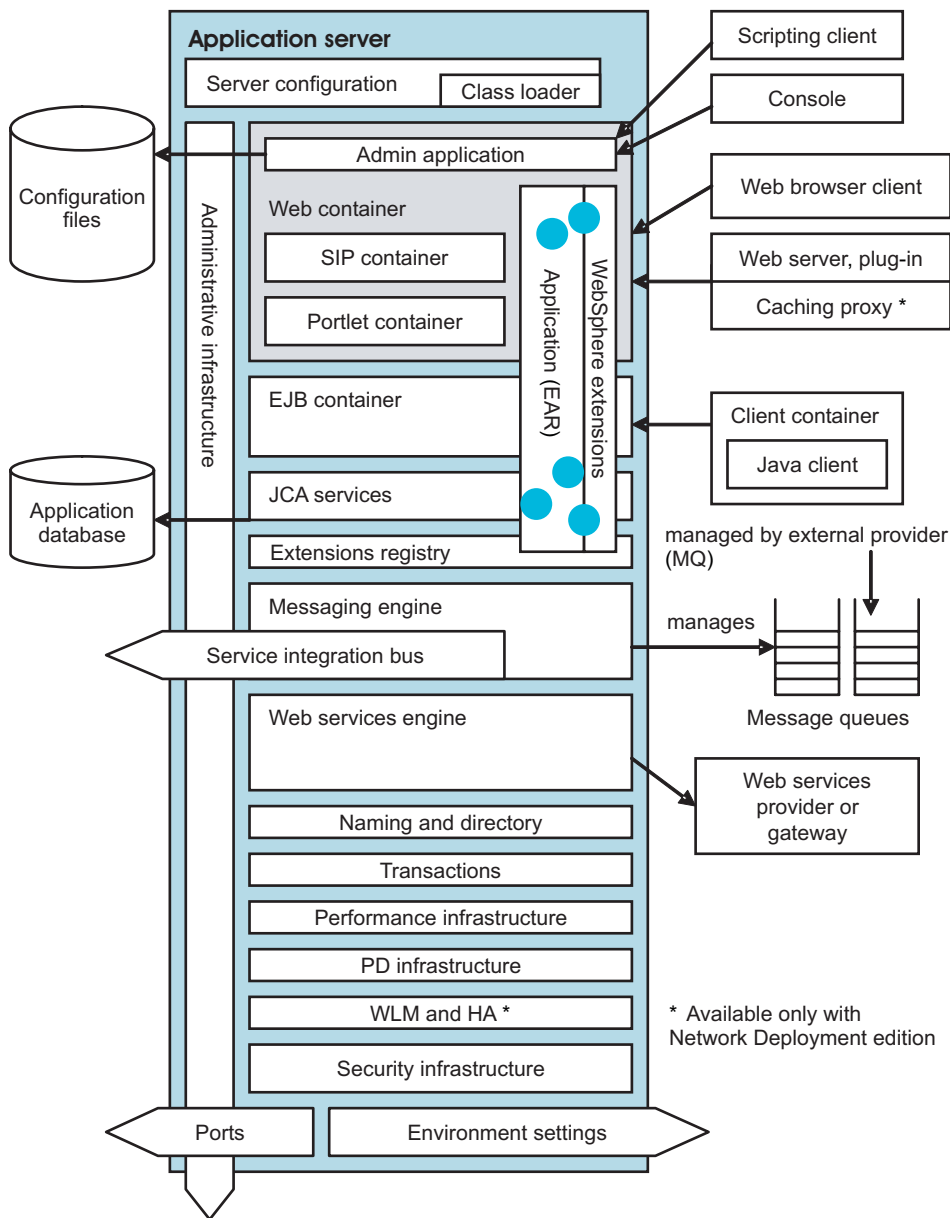
J2EE application components

Web applications run in the Web container

The Web container is the part of the application server in which Web application components run. Web applications are comprised of one or more related servlets, JavaServer Pages technology (JSP files), and Hyper Text Markup Language (HTML) files that you can manage as a unit. Combined, they perform a business logic function.

The Web container processes servlets, JSP files, and other types of server-side includes. Each application server runtime has one logical Web container, which can be modified, but not created or removed. Each Web container provides the following.

Web container transport chains

Requests are directed to the Web container using the Web container inbound transport chain. The chain consists of a TCP inbound channel that provides the connection to the network, an HTTP inbound channel that serves HTTP requests, a Web container channel over which requests for servlets and JSP files are sent to the Web container for processing.

Servlet processing

When handling servlets, the Web container creates a request object and a response object, then invokes the servlet service method. The Web container invokes the servlet's destroy method when appropriate and unloads the servlet, after which the JVM performs garbage collection.

Servlets can perform such tasks as supporting dynamic Web page content, providing database access, serving multiple clients at one time, and filtering data.

JSP files enable the separation of the HTML code from the business logic in Web pages. IBM extensions to the JSP specification make it easy for HTML authors to add the power of Java technology to Web pages, without being experts in Java programming.

HTML and other static content processing

Requests for HTML and other static content that are directed to the Web container are served by the Web container inbound chain. However, in most cases, using an external Web server and Web server plug-in as a front end to the Web container is more appropriate for a production environment.

Session management

Support is provided for the javax.servlet.http.HttpSession interface as described in the Servlet application programming interface (API) specification.

An *HTTP 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, many Web applications allow users to dynamically collect data as they move through the site, based on a series of selections on pages they visit. Where the user goes next, or what the site displays next, might depend on what the user has chosen previously from the site. To maintain this data, the application stores it in a ″session.″

SIP applications and their container

SIP applications are Java programs that use at least one Session Initiation Protocol (SIP) servlet. SIP is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging.

Portlet applications and their container

Portlet applications are special reusable Java servlets that appear as defined regions on portal pages. Portlets provide access to many different applications, services, and Web content.

EJB applications run in the EJB container

The EJB container provides all of the runtime services needed to deploy and manage enterprise beans. It is a server process that handles requests for both session and entity beans.

Enterprise beans are Java components that typically implement the business logic of J2EE applications, as well as accessing data. The enterprise beans, packaged in EJB modules, installed in an application server do not communicate directly with the server. Instead, the EJB container is an interface between EJB components and the application server. Together, the container and the server provide the enterprise bean runtime environment.

The container provides many low-level services, including threading and transaction support. From an administrative perspective, the container handles data access for the contained beans. A single container can host more than one EJB Java archive (JAR) file.

Client applications and other types of clients

In a client-server environment, clients communicate with applications running on the server. *Client applications* or *application clients* generally refers to clients implemented according to a particular set of

Java specifications, and which run in the client container of a J2EE-compliant application server. Other clients in the WebSphere Application Server environment include clients implemented as Web applications (*Web clients*), clients of Web services programs (*Web services clients*), and clients of the product systems administration (*administrative clients*).

Client applications and their container

The client container is installed separately from the application server, on the client machine. It enables the client to run applications in an EJB-compatible J2EE environment. The diagram shows a Java client running in the client container.

This product provides a convenient "launchClient tool" on page 333 for starting the application client, along with its client container runtime.

Depending on the source of technical information, client applications sometimes are called application clients. In this documentation, the two terms are synonymous.

Web clients, known also as web browser clients

The diagram shows a Web browser client, which can be known simply as a Web client, making a request to the Web container of the application server. A Web client or Web browser client runs in a Web browser, and typically is a Web application.

Web services clients

Web services clients are yet another kind of client that might exist in your application serving environment. The diagram does not depict a Web services client. The Web services information includes information about this type of client.

Administrative clients

The diagram shows two kinds of administrative clients: a scripting client and the administrative console that is the graphical user interface (GUI) for administering this product. Both are accessing parts of the systems administration infrastructure. In the sense that they are basically the same for whatever kind of applications you are deploying on the server, administrative clients are part of the product architecture. However, because many of these clients are programs you create, they are discussed as part of the programming model for completeness.

Web services

Web services

The diagram shows the Web services engine, part of the Web services support in the application server runtime. Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. They implement a services oriented architecture (SOA), which supports the connecting or sharing of resources and data in a flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

The product acts as both a Web services provider and as a requestor. As a provider, it hosts Web services that are published for use by clients. As a requester, it hosts applications that invoke Web services from other locations. The diagram shows the Web services engine in this capacity, contacting a Web services provider or gateway.

Data access, messaging, and J2EE resources

Data access resources

Connection management for access to enterprise information systems (EIS) in the application server is based on the J2EE Connector Architecture (JCA) specification. The diagram shows JCA services helping an application to access a database in which the application retrieves and persists data.

The connection between the enterprise application and the EIS is done through the use of EIS-provided resource adapters, which are plugged into the application server. The architecture specifies the connection management, transaction management, and security contracts between the application server and EIS.

The Connection Manager (not shown) in the application server pools and manages connections. It is capable of managing connections obtained through both resource adapters defined by the JCA specification and data sources defined by the JDBC 2.0 Extensions specification.

JDBC resources (JDBC providers and data sources) are a type of *J2EE resource* used by applications to access data. Although data access is a broader subject than that of JDBC resources, this information often groups data access under the heading of J2EE resources for simplicity.

JCA resource adapters are another type of J2EE resource used by applications. The JCA defines the standard architecture for connecting the J2EE platform to heterogeneous EIS. Imagine an ERP, mainframe transaction processing, database systems, and legacy applications not written in the Java programming language.

The JCA resource adapter is a system-level software driver supplied by EIS vendors or other third-party vendors. It provides the connectivity between J2EE application servers or clients and an EIS. To use a resource adapter, install the resource adapter code and create configurations that use that adapter. The product provides a predefined relational resource adapter for your use.

Messaging resources and messaging engines

JMS support enables applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). Applications can use message-driven beans to automatically to automatically retrieve messages from JMS destinations and JCA endpoints without explicitly polling for messages.

For inbound non-JMS requests, message-driven beans use a Java Connector Architecture (JCA) 1.5 resource adapter written for that purpose. For JMS messaging, message-driven beans can use a JCA-based messaging provider such as the default messaging provider that is part of WebSphere Application Server.

The messaging engine supports the following types of message providers.

Default messaging provider (service integration bus)

The default messaging provider uses the service integration bus for transport. The default message provider provides point-to-point functions, as well as publish and subscribe functions. Within this provider, you define JMS connection factories and destinations that correspond to service integration bus destinations.

WebSphere MQ provider

You can use WebSphere MQ as the external JMS provider. The application server provides the JMS client classes and administration interface, while WebSphere MQ provides the queue-based messaging system.

Generic JMS provider

You can use another messaging provider as long as it implements the ASF component of the JMS 1.0.2 specification. JMS resources for this provider cannot be configured using the administrative console.

transition: Version 6 replaces the Version 5 concept of a JMS server with a messaging engine built into the application server, offering the various kinds of providers mentioned previously. The Version 5 messaging provider is offered for configuring resources for use with Version 5 embedded messaging. You also can use the Version 5 default messaging provider with a service integration bus.

EJB 2.1 introduces an ActivationSpec for connecting message-driven beans to destinations. For compatibility with Version 5, you still can configure JMS message-driven beans (EJB 2.0) against a listener port. For those message-driven beans, the message listener service provides a listener manager that controls and monitors one or more JMS listeners, each of which monitors a JMS destination on behalf of a deployed message-driven bean.

Service integration bus

The service integration bus provides a unified communication infrastructure for messaging and service-oriented applications. The service integration bus is a JMS provider that provides reliable message transport and uses intermediary logic to adapt message flow intelligently into the network. It supports the attachment of Web services requestors and providers. Its capabilities are fully integrated into product architecture, including the security, system administration, monitoring, and problem determination subsystems.

The service integration bus is often referred to as just a bus. When used to host JMS applications, it is often referred to as a messaging bus. It consists of the following parts (not shown at this level of detail in the diagram).

Bus members
Application servers added to the bus.

Messaging engine
The component that manages bus resources. It provides a connection point for clients to produce or from where to consume messages.

Destinations
The place within the bus to which applications attach to exchange messages. Destinations can represent Web services endpoints, messaging point-to-point queues, or messaging publish and subscribe topics. Destinations are created on a bus and hosted on a messaging engine.

Message store
Each messaging engine uses a set of tables in a supported data store (such as a JDBC database) to hold information such as messages, subscription information, and transaction states.

Through the service integration bus Web services enablement, you can:

- Make an internal service that is already available at a service destination available as a Web service.
- Make an external Web service available at a service destination.
- Use the Web Services Gateway to map an existing service, either an internal service or an external Web service, to a new Web service that appears to be provided by the gateway.

Mail, URLs, and other J2EE resources
The following kinds of J2EE resources are used by applications deployed on a J2EE-compliant application server.

- JDBC resources and other technology for data access (previously discussed)
- JCA resource adapters (previously discussed)
- JMS resources and other messaging support (previously discussed)
- JavaMail support, for applications to send Internet mail

  The **JavaMail APIs** provide a platform and protocol-independent framework for building Java-based mail client applications. The APIs require service providers, known as protocol providers, to interact with mail servers that run on the appropriate protocols.

  A mail provider encapsulates a collection of protocol providers, including Simple Mail Transfer Protocol (SMTP) for sending mail; Post Office Protocol (POP) for receiving mail; and Internet Message Access Protocol (IMAP) as another option for receiving mail. To use another protocol, you must install the appropriate service provider for the protocol.

  JavaMail requires not only service providers, but also the JavaBeans Activation Framework (JAF), as the underlying framework to handle complex data types that are not plain text, such as Multipurpose Internet Mail Extensions (MIME), URL pages, and file attachments.

- URLs, for describing logical locations

URL providers implement the functionality for a particular URL protocol, such as HTTP, enabling communication between the application and a URL resource that is served by a particular protocol. A default URL provider is included for use by any URL resource with protocols based on the supported Java 2 Standard Edition specification, such as HTTP, FTP, or File. You also can plug in your own URL providers that implement additional protocols.

- Resource environment entries, for mapping logical names to physical names

The java:comp/env environment provides a single mechanism by which both the JNDI name space objects and local application environment objects can be looked up. The product provides numerous local environment entries by default.

The J2EE specification also provides a mechanism for defining customer environment entries by defining entries in the standard deployment descriptor of an application. The J2EE specification uses the following methods to separate the definition of the resource environment entry from the application.

- – Requiring the application server to provide a mechanism for defining separate administrative objects that encapsulate a resource environment entry. The administrative objects are accessible using JNDI in the application server local name space (java:comp/env).
- – Specifying the administrative object's JNDI lookup name and expected returned object type. This specification is performed in the aforementioned resource environment entry in the deployment descriptor.

The product supports the use of resource environment entries with the following administrative concepts.

- – A *resource environment entry* defines the binding target (JNDI name), factory class, and return object type (via the link to a referenceable) of the resource environment entry.
- – A *referenceable* defines the class name of the factory that returns object instances implementing a Java interface.
- – A *resource environment provider* groups together the referenceable, resource environment entries and any required custom properties.

Security

Security programming model and infrastructure

The product provides security infrastructure and mechanisms to protect sensitive J2EE resources and administrative resources and to address enterprise end-to-end security requirements on authentication, resource access control, data integrity, confidentiality, privacy, and secure interoperability.

Security infrastructure and mechanisms protect Java 2 Platform, Enterprise Edition (J2EE) resources and administrative resources, addressing your enterprise security requirements. In turn, the security infrastructure of this product works with the existing security infrastructure of your multiple-tier enterprise computing framework. Based on open architecture, the product provides many plug-in points to integrate with enterprise software components to provide end-to-end security.

The security infrastructure involves both a programming model and elements of the product architecture that are independent of the application type.

Additional services for use by applications

Naming and directory

Each application server provides a naming service that in turn provides a Java Naming and Directory Interface (JNDI) name space. The service is used to register resources hosted on the application server. The JNDI implementation is built on top of a Common Object Request Broker Architecture (CORBA) naming service (CosNaming).

JNDI provides the client-side access to naming and presents the programming model used by application developers. CosNaming provides the server-side implementation and is where its name

space is actually stored. JNDI essentially provides a client-side wrapper of the name space stored in CosNaming, and interacts with the CosNaming server on behalf of the client.

Clients of the application server use the naming architecture to obtain references to objects related to those applications. The objects are bound into a mostly hierarchical structure called the name space. It consists of a set of name bindings, each one of which is a name relative to a specific context and the object bound with that name. The name space can be accessed and manipulated through a name server.

This product provides the following naming and directory features.
- Distributed name space, for additional scalability
- Transient and persistent partitions, for binding at various scopes
- Federated name space structure across multiple servers
- Configured bindings for defining bindings bound by the system at server startup
- Support for CORBA Interoperable Naming Service (INS) object URLs

Note that with the addition of virtual member manager to provide federated repository support for product security, the product now offers more extensive and sophisticated identity management capabilities than ever before, especially in combination with other WebSphere and Tivoli products.

Object Request Broker (ORB)

The product uses an ORB to manage interaction between client applications and server applications, as well as among product components. An ORB uses IIOP to enable clients to make requests and receive requests from servers in a network distributed environment.

The ORB provides a framework for clients to locate objects in the network and call operations on those objects as though the remote objects were located in the same running process as the client, providing location transparency.

Although not shown in the diagram, one place in which the ORB comes into play is where the client container is contacting the EJB container on behalf of a Java client.

Transactions

Part of the application server is the transaction service. The product provides advanced transactional capabilities to help application developers avoid custom coding. It provides support for the many challenges related to integrating existing software assets with a J2EE environment. These measures include ActivitySessions (described below).

Applications running on the server can use transactions to coordinate multiple updates to resources as one unit of work such that all or none of the updates are made permanent. Transactions are started and ended by applications or the container in which the applications are deployed.

The application server is a transaction manager that supports coordination of resource managers and participates in distributed global transactions with other compliant transaction managers.

The server can be configured to interact with databases, JMS queues, and JCA connectors through their local transaction support when distributed transaction support is not required.

How applications use transactions depends on the type of application, for example:
- A session bean either can manage its transactions itself, or delegate the management of transactions to the container.
- Entity beans use container-managed transactions.
- Web components, such as servlets, use bean-managed transactions.

The product handles transactions with the following components.
- A transaction manager supports the enlistment of recoverable XAResources and ensures each resource is driven to a consistent outcome, either at the end of a transaction, or after a failure and restart of the application server.

- A container manages the enlistment of XAResources on behalf of deployed applications when it performs updates to transactional resource managers such as databases. Optionally, the container can control the demarcation of transactions for EJB applications that have enterprise beans configured for container-managed transactions.
- An API handles bean-managed enterprise beans and servlets, allowing such application components to control the demarcation of their own transactions.

WebSphere extensions

WebSphere programming model extensions are the programming model benefits you gain by purchasing this product. They represent leading edge technology to enhance application capability and performance, and make programming and deployment faster and more productive.

In addition, now 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 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 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. For more information about this feature, see "Application extension registry" on page 97.

The various WebSphere programming model extensions, and the corresponding application services that support them in the application server runtime, can be considered in three groups: Business Object Model extensions, Business Process Model extensions, and extensions for producing Next Generation Applications.

Extensions pertaining to the Business Object Model

Business object model extensions operate with business objects, such as enterprise bean (EJB) applications.

Application profiling
> Application profiling is a WebSphere extension for defining strategies to dynamically control concurrency, prefetch, and read-ahead.
>
> Application profiling and access intent provide a flexible method to fine-tune application performance for enterprise beans without impacting source code. Different enterprise beans, and even different methods in one enterprise bean, can have their own intent to access resources. Profiling the components based on their access intent increases performance in the application server runtime.

Dynamic query
> Dynamic query is a WebSphere programming extension for unprecedented application flexibility. It lets you dynamically build and submit queries that select, sort, join, and perform calculations on application data at runtime. Dynamic Query service provides the ability to pass in and process EJB query language queries at runtime, eliminating the need to hard-code required queries into deployment descriptors during application development.
>
> Dynamic query improves enterprise beans by enabling the client to run custom queries on EJB components during runtime. Until now, EJB lookups and field mappings were implemented at development time and required further development or reassembly in order to be changed.

Dynamic cache

The dynamic cache service improves performance by caching the output of servlets, commands, and JSP files. This service within the application server intercepts calls to cacheable objects and either stores the output of the object or serves the content of the object from the dynamic cache.

Because J2EE applications have high read-write ratios and can tolerate small degrees of latency in the currency of their data, the dynamic cache can create opportunity for significant gains in server response time, throughput, and scalability.

Features include cache replication among clusters, cache disk offload, Edge side include caching, and external caching - the ability to control caches outside of the application server, such as that of your Web server.

Extensions pertaining to the Business Process Model

Business process model extensions provide process, workflow functionality, and services for the application server. Use them in conjunction with business integration capabilities.

ActivitySessions

ActivitySessions are a WebSphere extension for reducing the complexity of dealing with commitment rules and limitations associated with one-phase commit resources.

ActivitySessions provide the ability to extend the scope of multiple local transactions, and to group them. This enables them to be committed based on deployment criteria or through explicit program logic.

Web services

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. They implement a services oriented architecture (SOA), which supports the connecting or sharing of resources and data in a very flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

Extensions for creating next generation applications

Next generation applications can be used in applications that need the specific extensions. These enable next generation development by leveraging the latest innovations that build on today's J2EE standards. This provides greater control over application development, execution, and performance than was ever possible before.

Asynchronous beans

Asynchronous beans offer performance enhancements for resource-intensive tasks by enabling single tasks to run as multiple tasks. Asynchronous scheduling facilities can also be used to process parallel processing requests in ″batch mode″ at a designated time. The product provides full support for asynchronous execution and invocation of threads and components within the application server. The application server provides execution and security context for the components, making them an integral part of the application.

Startup beans

Startup beans allow the automatic execution of business logic when the application server starts or stops. For example, they might be used to pre-fill application-specific caches, initialize application-level connection pools, or perform other application-specific initialization and termination procedures.

Object pools

Object pools provide an effective means of improving application performance at runtime, by allowing multiple instances of objects to be reused. This reuse reduces the overhead associated with instantiating, initializing, and garbage-collecting the objects. Creating an object pool allows an application to obtain an instance of a Java object and return the instance to the pool when it has finished using it.

Internationalization

The internationalization service is a WebSphere extension for improving developer productivity. It allows you to automatically recognize the time zone and location information of the calling client, so that your application can act appropriately. The technology enables you to deliver each user, around the world, the right date and time information, the appropriate currencies and languages, and the correct date and decimal formats.

Scheduler

The scheduler service is a WebSphere programming extension responsible for starting actions at specific times or intervals. It helps minimize IT costs and increase application speed and responsiveness by maximizing utilization of existing computing resources. The scheduler service provides the ability to process workloads using parallel processing, set specific transactions as high priority, and schedule less time-sensitive tasks to process during low traffic off-hours.

Work areas

Work areas are a WebSphere extension for improving developer productivity. Work areas provide a capability much like that of ″global variables.″ They provide a solution for passing and propagating contextual information between application components.

Work areas enable efficient sharing of information across a distributed application. For example, you might want to add profile information as each customer enters your application. By placing this information in a work area, it will be available throughout your application, eliminating the need to hand-code a solution or to read and write information to a database.

To delve deeper into learning about any of the extensions, see "Learn about WebSphere programming extensions" on page 1119.

Accessing the Samples (Samples Gallery)

The Samples Gallery offers a set of Samples that demonstrate common enterprise application tasks. The Gallery also contains descriptions of where to find additional Samples and coding examples.

You can upgrade the Samples Gallery, including the server samples and client samples. For detailed information, refer to *app_server_root*/`samples/readme.html`.

Quick start - Accessing the Samples Gallery
Your application server must be running. The Samples must be installed. ▶ Windows On Windows systems, click (for example) **Start > Programs > IBM WebSphere > Application Server v6.1 > Profiles > profile_name > Samples Gallery**. ▶ Windows On Windows systems, click (for example) **Start > Programs > IBM WebSphere > Application Server v6.0 > Profiles > profile_name > Samples Gallery**. In the following examples, *hostname* is a variable. ▶ Windows `http://hostname:9080/WSsamples` ▶ Linux `http://hostname:9080/WSsamples` Client Samples: `install_root/samples/index.html` (local file system)

The Samples are for demonstration purposes only. See the following limitations for details.

First time here? Read the following information. A little setup is involved.

Samples Gallery

- Samples Gallery contents
- Installing and accessing the Samples Gallery
- Changing the Samples Gallery port number and troubleshooting

Client Samples Gallery
- Client Samples Gallery contents
- Installing and viewing the Client Samples Gallery

General information
- Limitations of the Samples
- Additional Samples and examples

Samples Gallery contents

The Samples Gallery includes the following materials:

Plants by WebSphere application
> This application demonstrates several Java 2 Platform, Enterprise Edition (J2EE) functions, using an online store that specializes in plant and garden tool sales.

Faces Client Tutorial - Sample Portfolio
> Sample Portfolio is a sample application that demonstrates the use of faces client components. The Hello world sample demonstrates how the faces client framework keeps a data model consistent in the browser.

Technology Samples
> These Samples demonstrate various core components in J2EE applications.

Web Services Samples
> These Samples demonstrate J2EE beans and JavaBeans components that are available as Web services.

Service Data Objects (SDO) Sample
> This Sample demonstrates data access to a relational database through Service Data Objects (SDO) and Java DataBase Connectivity (JDBC) Mediator technologies.

BeenThere workload management Sample
> This Sample demonstrates the workload management and clustering capabilities of the product, and is the only Sample that is supported in a multi-server, clustered environment.

JACL scripts
> These scripts enable you to configure resources and install the Sample applications.

Programming model extensions Samples in the Samples Gallery
> These Samples demonstrate WebSphere programming model extension features such as dynamic query service, work area service, internationalization service, ActivitySessions service, application profiling, Java Transaction API (JTA) extensions, asynchronous beans, and scheduler.

Installing and accessing the Samples Gallery

Follow these steps to install and configure the Samples Gallery.
1. Install the product.

   Select to install the Application Server Samples package.

   The Samples are installed in the *app_server_root*/samples directory.
2. Create a stand-alone application server profile.

If you already created a profile without the Samples, and you want to create additional profiles that contain the Samples, you have to use the graphical profile creation interface.

The Samples are installed in the directory *profile_root*/installedApps/*cell_name*. The Sample source code is placed in the directory *profile_root*/installedApps/*cell_name*/samples.

You can build or modify the Samples source code to support your project.

To run the Samples in a distributed WebSphere Application Server Network Deployment environment, install and configure the Samples in a stand-alone application server profile installation, and then use **addNode** to federate the node into the deployment manager cell. When running the **addNode** command, use the dmgr_host argument, and -includeapps and -includebuses options. For a complete list of the **addNode** parameters and their usage, see addNode command.

For example:

```
addNode.sh/bat dmgr_hostname -includeapps -includebuses
```

3. Start the application server.
4. Access the Samples Gallery.

   The Plants by WebSphere application is installed on the application server by default. That is, you need not deploy the PlantsByWebSphere.ear file onto the application server. This application is listed as an installed Sample in the Samples Gallery.

   Try it out! See the quick start instructions on this page.

   If you have difficulty accessing the Samples Gallery, check the port number.

5. Install additional Samples.

   Additional Samples are initially listed as installable Samples in the Samples Gallery.

   To deploy them to the application server, use the install script in the *app_server_root*/samples/samples/bin directory. See the Samples Gallery for more detailed instructions.

For information about configuring security for Samples, see the Samples Gallery.

Changing the Samples Gallery port number and troubleshooting

If you are unable to access the Samples Gallery, verify the following items.
- Verify that the application server is running.
- Verify that the samples are installed. In the administrative console, expand **Applications** and click **Enterprise applications**. Confirm that the Samples Gallery is listed as an installed application.

  If the Samples Gallery is not listed as an installed application:
  – Create a new profile using the default profile template.
- Verify the port number.

  The default port is 9080, which is the default value of the HTTP transport of the application server. If the Samples Web address fails, confirm the port number.

  ▶ Windows ◀ You can change the port number in the **Properties** window by clicking **Start > Programs > IBM WebSphere > Application Server Network Deployment v6.0 > profiles > profile_name > Samples Gallery**.
- You must specify a Samples password when using the manageprofiles command to create a profile for which administrative security is enabled. Otherwise, you will be allowed to create the profile successfully, but when you run the application server containing the Samples, exceptions and failures will be thrown to the server system out log.

Client Samples Gallery contents

The Client Samples Gallery demonstrates the following clients.
- J2EE application client
- Java thin client

- Applet client
- ActiveX to EJB Bridge client

Installing and viewing the Client Samples Gallery

To view the Client Samples Gallery, install the WebSphere Application Server client on a distributed machine.

See "Installing Application Client for WebSphere Application Server" on page 326 for more information.

Access the Client Samples Gallery on your local file system at *app_server_root*/samples/samples/index.html.

Limitations of the Samples

- The Samples are for demonstration purposes only.

    The code that is provided is not intended to run in a secured production environment. The Samples support Java 2 Security, therefore the Samples implement policy-based access control that checks for permissions on protected system resources, such as file I/O. The Samples also support administrative security.

- The BeenThere workload management Sample is the only Sample that is supported in a multi-server, clustered environment. For information on the BeenThere Sample, see the `readme_BeenThere.html` file in the Samples directory.

- Many of the WebSphere Application Server Samples connect to a Cloudscape database using the Cloudscape-embedded framework. The Cloudscape-embedded framework has a limitation that only one Java virtual machine (JVM) can access a given database instance. As a result, in a clustered WebSphere environment, the second server in the node fails to start the Sample applications, because the first server (JVM) already holds a connection to that database instance.

    For applications that require multiple Java virtual machines to access the same Cloudscape instance, use the Cloudscape networkServer framework.

Additional Samples and examples

IBM Telephone Directory
> The IBM Telephone Directory business application is shipped separately from WebSphere Application Server. For information about obtaining and using the IBM Telephone Directory application, see IBM Telephone Directory V5.2 in the *e-business and Web serving* topic of the iSeries Information Center.

Samples on developerWorks
> Additional WebSphere Application Server Samples are available on WebSphere developerWorks

Samples in tutorials
> Many WebSphere Application Server tutorials rely on Sample code. To find tutorials that demonstrate specific technologies, browse the links in Tutorials.

Examples in the product documentation
> The product documentation contains many code snippets and examples. To locate these examples easily, see the developer examples in the **Reference** section of the information center navigation for the product edition that you are using.

Java Samples on the Sun Microsystems Web site
> Although they do not showcase the capabilities added by purchasing WebSphere Application Server, the Samples on the java.sun.com Web site demonstrate the basic functionality of various technologies.

Web resources for learning

This topic familiarizes you with the many Web sites containing technical information for understanding and using your WebSphere Application Server product. A wealth of online information is available to complement the product documentation.

Choose an area of interest.
- Learning and education
- Developer resources
- Architect, planner, installer, and administrator resources
- Partner resources
- Redbooks, white papers, and documentation
- Troubleshooting and support

Also, throughout the documentation, you will find additional resources for learning pages, each focused on a specific technology, such as Web services. The pages provide links to particular documents of interest.

Learning and education

IBM Education Assistant

Find tutorials, multimedia demonstrations, and presentations for WebSphere servers and Rational development tools.

Training and certification

It's easy to learn about WebSphere® software. IBM has several educational options available to you. From classroom courses to onsite assistance and Internet-based training, if you're ready to learn, we're ready to teach.

Developer resources

developerWorks - WebSphere Application Server zone

Use this page to search for information, download software including trial code and fixes, learn about the application server, and find support and migration information.

Samples Gallery (locally installed)

The Samples Gallery is a locally installable application that offers a set of Samples that demonstrate common Web application tasks.

Architect, planner, installer, and administrator resources

Detailed system requirements page

These pages describe the minimum product levels you should have installed before opening a problem report with the WebSphere Application Server support team.

Patterns for e-business

Patterns for e-business are a group of reusable assets that can help speed the process of developing Web-based applications. The Patterns leverage the experience of IBM architects to create solutions quickly, whether for a small local business or a large multinational enterprise.

Partner resources

PartnerWorld

Find product, business, and technical inforamtion. The PartnerWorld program is designed to offer IBM Business Partners benefits, technical support, education, marketing campaigns, sales tools and more to help you grow your business and drive profits.

Redbooks, white papers, and documentation

Redbooks - WebSphere

Find Redbooks pertaining to WebSphere, including the newest, latest, and most popular Redbooks and Redpapers in draft and published form.

White papers

This link performs a query for white papers that are relevant to WebSphere Application Server.

Library page

A new, improved Web page for finding product documentation, including the online information center, documentation plug-ins for offline viewing with the WebSphere help system, and PDF books. This page links to a variety of other kinds of product information, such as WebSphere Redbooks.

Troubleshooting and support

WebSphere Application Server - Support

This page provides a convenient starting point for querying technical documents, solving problems, downloading fixes, planning, learning, and communicating.

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/was/support/

Support - Recent updates

This document lists valuable resources and newly created content.

Support - Resource reference list

This document is an introduction to available documentation and educational resources.

Support - Quick links

This document provides a reference of direct links to available documentation and educational resources.

Notes

*   The WebSphere Application Server product documentation found in the information center and PDF books documents supported configurations. Many of the above sites could contain information that describes unsupported configurations.
*   Information residing on non-IBM sites is provided for your convenience. Its technical accuracy is controlled by the owner of the site. Use the information at your own risk.

What is new for developers

This version contains many new and changed features for application developers.

New in Version 6.1! indicates new features or changes implemented at the Version 6.1 level. Unmarked items are Version 6.0 improvements that apply also to Version 6.1, which should interest anyone migrating to Version 6.1 from Version 5.x.

Deprecated and removed features describes features that are being replaced or removed in this or future releases.

Web services

Web services

This product has been a leader in advocating support for Web services standards that allow more automated, less hand-coded cross-platform computing. Standards support includes WS-Security, which authenticates communications between web services, and WS-Transactions, which is designed to assure that Web Services transactions are consistently delivered. Additionally, the product supports the WS-I Basic Profile 1.1 for development of interoperable Web services supporting the integration of Web services solutions.

See "Implementing Web services applications" on page 371.

WS-Transaction affinity, routing, and authorization

New in Version 6.1! The implementation in this product version removes 6.0 limitations to provide Web services the same level of distributed transaction support as enterprise beans using CORBA:

- WS-AT contexts use virtual host names and can span firewalls
- Application requests with WS-AT contexts can place transactional affinity constraints on client-side workload management.
- WS-AT protocol messages can be secured.

This product implements a standards-based solution to allow Web services on disparate systems to take part in global transactions with ACID properties. Transactions can span between JTA and J2EE and WS-AT/Web services domains in a seamless manner, requiring no additional programming.

Web services on disparate systems can take part in a compensation model in which compensational scopes span J2EE components and WS-BA/Web services domains in a seamless manner, requiring no additional programming. Applications distributed between WebSphere Application Server and other vendor solutions, for example Microsoft .NET, can take part in the same global transaction.

See "Web Services Atomic Transaction support in WebSphere Application Server" on page 1101.

WS-Notification support – "pub/sub for Web services"

New in Version 6.1! The WS-Notification v1.3 specifications have been added to the WebSphere programming model. Informally described as 'pub/sub for web services,' this family of specifications define web service message exchanges (such as an application interface) to enable web service applications to utilize the 'publish and subscribe' messaging pattern. Traditionally, publish and subscribe messaging is used in message oriented middle ware scenarios to implement a one-to-many distribution pattern.

In the publish and subscribe pattern a producing application inserts (publishes) a message (event notification) into the messaging system, having marked it with a topic that indicates the subject area of the message. Consuming applications that have subscribed to the topic and have the appropriate authority, receive an independent copy of the message that was published by the producing application.

WS-Notification also allows interchange of event notification between WS-Notification applications and other clients of the service integration bus. By exploiting other service integration bus functionality, you can use this function to interchange messages with other IBM publish and subscribe brokers such as Event Broker or Message Broker.

For more information, search the information center for the text: tjwsn_ep

WS-Addressing support -- "protocol independent interoperability for Web services"	**New in Version 6.1!** The product offers support and interoperability with the latest WS-Addressing specifications from W3C, while maintaining interoperability with the pre-W3C specification. This family of specifications provides transport-neutral mechanisms to address Web services and to facilitate end-to-end addressing.

This product version provides a programming interface to support referencing and targeting of Web service endpoints that represent WS-Resource instances, as defined by the WS-Resource Framework specification. Additionally, this version introduces a programming interface to allow programmers to create, reason about and manipulate WS-Addressing artifacts. Programmers can specify the WS-Addressing Message Addressing Properties for outbound messages and also acquire WS-Addressing Message Properties from the incoming message at the receiving endpoint.

See "Web Services Addressing support" on page 466.

Enterprise beans can be invoked from a Web services client using RMI-IIOP	WebSphere Application Server Version 6.0.x supports directly accessing an enterprise JavaBean (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.

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 invoke Web services using protocols with the transport such as SOAP and Remote Method Invocation (RMI).

With Version 6.0.x, 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 J2EE programming model. When a Web service is implemented by an EJB, multiprotocol JAX-RPC permits the Web service invocation path to be optimized for WebSphere Java clients.

Using the RMI/IIOP protocol instead of a SOAP- based protocol yields better performance and enables you to get support for client transactions, which are not standard for Web services. Benefits include -- 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.

For more information, refer to "Using WSDL EJB bindings to invoke an EJB from a Web services client" on page 438.

New extensions to the JSR-101 and JSR-109 programming models	WebSphere Application Server Version 6.0.x provides extensions to the Java Specification Request JSR-101 and JSR-109 client programming models. These extensions include the following:

- The REQUEST_TRANSPORT_PROPERTIES property and RESPONSE_TRANSPORT_PROPERTIES property can be added to a Java API for XML-based RPC (JAX-RPC) client Stub to enable a Web services client to send or retrieve HTTP transport headers.
- Implementation-specific support for javax.xml.rpc.ServiceFactory.loadService() as described by the JSR-101 and JAX-RPC specifications. 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 public.javax.xml.rpc.Service loadService signatures.

For more information, refer to "Extensions to the JAX-RPC and Web Services for J2EE programming models" on page 399.

Updates to options used by the emitter tools Java2WSDL and WSDL2Java	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.
	New option: 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 be written to the output of the WSDL document. Review the Java2WSDL article for more information on using the -bindingTypes option.
	The WSDL2Java command is run against a Web Services Description Language (WSDL) file to create Java APIs and deployment descriptor templates. A 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 Java Specification Requirements (JSR) 109 1.1 specification defines deployment descriptors that deploy a Web service in a Java 2 Platform Enterprise Edition (J2EE) environment. The WSDL2Java command is run against the WSDL file to create Java APIs and deployment descriptor templates according to these specifications.
	For more information, refer to "Java2WSDL command" on page 415 and "WSDL2Java command" on page 419.
Additional HTTP transport properties for Web services applications	JVM custom properties are available to manage the connection pool for Web services HTTP outbound connections. Establishing a connection is an expensive operation. Connection pooling improves performance by avoiding the overhead of creating and disconnecting connections. When an application invokes a Web service over an HTTP transport, the HTTP outbound connector for the Web service locates and uses an existing connection from a pool of connections. When the response is received, the connector returns the connection to the connection pool for reuse. The overhead to create and disconnect the connection is avoided.
	See "Configuring additional HTTP transport properties using the JVM custom property panel in the administrative console" on page 429.

Additions to the programming model

J2EE 1.4 support	J2EE 1.4 specification support is the basis of this product's programming model. It enables you to take advantage of the latest Java technology, as described in Java 2 Platform, Enterprise Edition (J2EE) specification.
WebSphere extensions	Several more WebSphere extensions are now available in this product edition. As a starting point for learning about each extension, see "Learn about WebSphere programming extensions" on page 1119. See also the WebSphere extensions section in "Learn about WebSphere applications: Overview and new features" on page 1.
Portlet application support (JSR 168)	**New in Version 6.1!** Developers can write portlets, in addition to servlets. Administrators can configure, manage, and run portlet applications. Users can access the portlets with URLs, as they do servlets.

Remote RequestDispatch calls are handled the same as local calls	**New in Version 6.1!** Now you can deploy portlets in web applications into separate application server instances with their own virtual machines, but call the portlets with the request dispatcher, just as you would call portlets in local Web applications. This supports application isolation to keep one portlet from bringing down the entire portal server. Complexity and differentiation between remote and local calls is minimized for the administrator.
	See "Remote request dispatcher" on page 88.
Real time collaboration features in applications (JSR 116)	**New in Version 6.1!** The application programming model has been extended to include Session Initiation Protocol (SIP) servlet applications. Developers can write SIP applications, which are Java programs that use at least one Session Initiation Protocol (SIP) servlet. SIP is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging.
	An IETF standard, the SIP protocol (JSR 116) supports clients registration, presence management, and media session negotiation. Media sessions can include such diverse media as text chat, IP audio/video, application sharing, and electronic whiteboards. Applications are growing rapidly, from telecoms and wireless providers, call centers, pervasive computing, and Customer Relationship Management (CRM). The SIP proxy can route SIP or HTTP with enterprise class availability.
Java 5 Software Development Kit (SDK)	**New in Version 6.1!** Developers can use many new API libraries, including generics, auto boxing of primitives, annotations, and enumerated types.
Reliable World Type and Devanagari font availability	**New in Version 6.1!** The World Type fonts and Devanagari font are available as an e-fix from the product Support site. This is to help mitigate the variance in font coverage among Linux distributions, especially the Asian language versions.
64-bit support can affect applications using JNI	WebSphere Application Server now supports 64-bit environments. For most J2EE applications, support for 64-bit environments is not a concern. However, it is a concern for applications that use Java Native Interface (JNI) code. An application that uses JNI code might not start after deployment. The JNI allows Java code running in a virtual machine to operate with applications and libraries written in other languages, such as C, C++, and assembly. If your J2EE application uses JNI in a 32-bit environment, you must recompile your code in a 64-bit environment. The JNI calls might be different after the compilation because JNI specifications can change from version to version.
Added serialVersionUID (SUID) to handle imposing explicit version control for serialized classes	Classes implementing the Serializable interface have added serialVersionUID (SUID) to impose explicit version control for Java serialization. A serialVersionUID identifies the unique original class version for which a class is capable of writing streams and also from which that class can be read.
	best-practices: As you develop your applications, it is recommended that your classes implementing the Serializable interface use serialVersionUID (SUID) to impose explicit version control for Java serialization.

IBM JSF widget library for improved Web pages	**New in Version 6.1!** IBM JSF Widget Library (JWL) is provided as an optional library in WAS. Applications can use the library when it is included in the Shared Library path. FacesClient Framework enabled Web pages are able to sustain longer interactions with the end user without requiring roundtrips back to the server. By creating what effectively is an MVC (Model View Controller) model inside the page, a developer is able to define a working set and a set of controls that dynamically bind to that data. The user can then interact with the working data set, using those controls, and until a roundtrip back to the server is really necessary (for example, to submit data), the user benefits from response times and a freedom to interact with the page that is uncommon in regular Web pages.

For an enterprise that deploys FacesClient Framework enabled Web pages, in addition to increased user satisfaction due to a more interactive and more responsive application, it also benefits in other areas such as lower consumption of server-side resources. Because of the lower amount of roundtrips, and smaller page size relatively speaking, the enterprise is able to scale its server infrastructure and bandwidth further, accommodating more users in the current setup. Applications overall are simpler to develop and maintain. By enabling an MVC-like model on the page, the FacesClient Framework enables a development model based on standards such as JSF (Java Server Faces).

See "JavaServer Faces widget library (JWL)" on page 80. |
| *Java Server Faces (JSF) 1.1 support* | **New in Version 6.1!** Version 6.1 introduces the ability to use JSF 1.1 (JSR 127) in your Java-based Web applications without including the JSF runtime libraries in your application, meaning you can produce smaller applications. The JSF 1.1 DTD will be provided as part of the application server runtime. The JSF specification provides migration instructions and does not list any deprecations. Your JSF 1.0 applications will continue to run without modification.

See "JavaServer Faces" on page 76. |

Data access resources

Service Data Objects (SDO)	As Introduction to Service Data Objects explains, the SDO framework makes the J2EE data programming model simpler, so you can focus on the business logic of your applications.

See "Data access with Service DataObjects" on page 717. |
| *Easier programming of disconnected data objects* | **New in Version 6.1!** An enhanced EJB Service Data Object (SDO) Mediator simplifies the programming model. Current techniques for implementing a disconnected data objects involve a combination of copy helper objects, session beans and EJB access beans. Using the EJB mediator reduces the amount of programming. Dynamic data objects provide flexibility and eliminate the need to define copy helper type objects. Increased performance can be achieved with optimized queries and having the EJB mediator read and write directly to the data store, bypassing the need to activate EJB instances. In addition, the EJB Mediator allows the EJB entity bean programming model and the EJB query language to provide services that can send or receive SDOs.

See "Enterprise JavaBeans Data Mediator Service" on page 733. |

| Cloudscape 10.1.x database support | **New in Version 6.1!** WebSphere Application Server supports Cloudscape v10.1.x as a test and development database. The new Cloudscape is a pure Java database server. The code base, which the open source community calls Derby, is a product of the Apache Software Foundation (ASF) open source relational database project. Cloudscape 10.1.x highlights include: |

* com.ibm.db2j.* becomes org.apache.derby.*:
* org.apache.derby.drda contains the networkServerControl to manipulate the NetworkServer process
* org.apache.derby.jdbc contains the JDBC classes
* org.apache.derby.tools contains the tools like ij and sysinfo dblook
* db2j.properties file becomes derby.properties
* db2j.system.home becomes derby.system.home
* db2j.drda.* becomes derby.drda.*

See the Cloudscape section of ibm.com: http://www-306.ibm.com/software/data/cloudscape/.

Messaging resources

| *Easier to configure access to WebSphere MQ queues from the service integration bus* | **New in Version 6.1!** The service integration bus offers improved, easier to configure connectivity to WebSphere MQ software. An application connected to the bus now can read messages directly from any z/OS WMQ queue, reducing your need to repeat MQ configuration details. |

Version 6.0 did not allow pulling messages directly from MQ queues, which could only be configured as "foreign destinations." In this version, a bus destination can act as a proxy for a z/OS WMQ Queue. JMS applications (including those using message driven beans) can access WMQ queues through such destinations. Requests to both send and receive messages against a destination that is acting as a proxy are delegated to the Queue Manager of the MQ queue, using MQ client protocols including XA flows. This capability enables queue access to be coordinated as part of a global transaction running in WebSphere Application Server.

As a starting point, see **Learn about WebSphere applications > Service integration** in the WebSphere Application Server information center navigation.

| *Flexibility in storage options* | **New in Version 6.1!** Storage options now include using the file system instead of a relational database. The message store component of a service integration bus messaging engine can be configured to use the file system for persistent storage as an alternative to using a relational database. New messaging engines are configured with a file-system-based message store by default. Options are provided in the console wizard and related scripting commands to specify directories and storage file sizes. Options also exist to select a relational-database-based message store. |

As a starting point, see **Learn about WebSphere applications > Service integration** in the WebSphere Application Server information center navigation.

Improved development and assembly tools

| *Easier deployment* | Deploying applications has never been easier -- particularly redeploying updated applications or modules. |

Updates to the Application Server Toolkit	The Application Server Toolkit has new capability:

The Application Server Toolkit has new capability:

- The server editor now has an option to optimize a WebSphere Application Server v6.x server for testing and developing. This option can reduce the startup time of the server. For details on the **Optimize server for testing and developing** check box, see *Reducing the startup time for WebSphere Application Server v6.0* in the online help.

- For EJB 2.x container managed persistence (CMP) entity beans, you can now use a partial operation to specify how you want to update the persistent attributes of the CMP bean to the database. Use the UPDATE_ONLY option for the partial operation to limit updates to the database to only persistent attributes of the CMP bean that have been modified. You can specify the partial operation as a persistent option at the bean-level in the access intent policy configured for the bean. For details on how to use the **Partial Operation** check box, see *Partial operation for container managed persistence* in the online help.

- You can now specify Derby v10 as a valid database vendor backend ID when generating EJB deployment code. See *The ejbdeploy command* in the online help.

- You can now specify the `-dbvendor` option for a mapped JAR file. In releases previous to v6.0.2, if the `-dbvendor` option is specified for mapped JAR files, the database vendor specification is ignored. Specifying the database vendor in the ejbdeploy command is used for generating new top-down maps. If omitted, then the ejbdeploy command uses a default value: `DB2UDB_V81`. For 2.x CMP beans, multiple mappings to different database vendors are supported. 1.1 CMP beans can only be mapped once. For details on the `-dbvendor` option, see the online help.

See "Assembly tools."

Assembly tools

WebSphere Application Server supports two tools that you can use to develop, assemble, and deploy J2EE modules: Application Server Toolkit (AST) and Rational Application Developer. These tools are referred to in this information center as the *assembly tools*.

The AST is available in your WebSphere Application Server CD-ROM package. Rational Application Developer is available only on a trial basis in the WebSphere Application Server CD-ROM package.

The assembly feature of the AST and Rational Application Developer products runs on Windows and Linux Intel platforms. Users of WebSphere Application Server on other platforms must assemble their modules using an assembly tool installed on Windows or Linux Intel platforms. To install an assembly tool, follow instructions available with the tool.

Although this information center refers to the AST and Rational Application Developer products as the *assembly tools*, you can use the products to do more than assemble modules. Rational Application Developer is an integrated development environment that provides development, testing, assembly and deployment capabilities. However, topics on application assembly in this information center focus on assembling J2EE modules using the J2EE Perspective of the assembly tools. Each assembly tool provides extensive online documentation; the topics on application assembly in this information center supplement that documentation. The **Application Server Toolkit** information center is available with this information center.

Enterprise (J2EE) applications

Enterprise applications (or J2EE applications) are applications that conform to the Java 2 Platform, Enterprise Edition, specification.

Enterprise applications can consist of the following:
- Zero or more EJB modules (packaged in JAR files)

- Zero or more Web modules (packaged in WAR files)
- Zero or more connector modules (packaged in RAR files)
- Zero or more Session Initiation Protocol (SIP) modules (packaged in SAR files)
- Zero or more application client modules
- Additional JAR files containing dependent classes or other components required by the application
- Any combination of the above

A J2EE application is represented by, and packaged in, an enterprise archive (EAR) file.

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:
- Specifications: 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:
- IBM Faces Component Catalog
- 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

Chapter 2. Designing applications

This topic highlights Web sites and other ideas for finding best practices for designing WebSphere applications, particularly in the realm of WebSphere extensions to the Java 2 Platform, Enterprise Edition (J2EE) specification.

When designing WebSphere applications, follow the example set by the Samples. Refer to the code in the Samples Gallery that is available with the product. In particular, the Samples Gallery highlights new and WebSphere-specific aspects of the programming model.

Use the following links to find relevant supplemental information about designing WebSphere applications. 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.

Web resources for learning

- The top 10 (more or less) J2EE best practices

  The authors, who are IBM consultants and performance experts, describe this document in the following way: Over the last five years, a lot has been written about J2EE best practices. There now are probably 10 or more books along with dozens of articles that provide insight into how J2EE applications should be written. In fact, there are so many resources, often with contradictory recommendations, navigating the maze has become an obstacle to adopting J2EE itself. To provide some simple guidance for customers entering this maze, we set out to compile the following ″top 10″ list of what we feel are the most important best practices for J2EE.

- IBM Patterns for e-Business

  Patterns for e-business are a group of reusable assets that can help speed the process of developing Web-based applications. The patterns leverage the experience of IBM architects to create solutions quickly, whether for a small local business or a large multinational enterprise.

- WebSphere Best Practices and Performance Considerations

  This document is older (2001), but its focus on the fundamentals of Web and Enterprise JavaBeans (EJB) application programming helps it stand the test of time.

- Best practices for using XSLT in WebSphere Application Server applications

  The author states: In this article I explore the reasons why some WebSphere Application Server applications use XSL for HTML production instead of JavaServer Pages (JSP) files. I will compare the performance of XSLT for HTML/XHTML production against JSP files and browser formatting. I will then provide guidance on how to improve XSLT performance in WebSphere Application Server should you decide to go this route. While this article focuses on the use of XSLT for the production of HTML, the performance best practices are directly applicable to other WebSphere Application Server uses of XSLT, such as XML-to-XML transformations and XML-to-text transformations.

- Rational on developerWorks

  This page provides quick links to technical resources and best practices for Rational software. Browse information by product or by technology. Find resources for learning, support, and developer communities.

- developerWorks site

  developerWorks is IBM's technical resource for developers, providing a wide range of tools, code, and education on DB2, eServer, Lotus, Rational, Tivoli, and WebSphere as well as on open standards technology such as Web services, Wireless, Linux, XML, Java technologies, and more. By providing focused and relevant technical information for developers, developerWorks offers choices you can apply

to building and deploying applications across heterogeneous systems. Using developerWorks, you can take full advantage of open standards and the IBM Software Development Platform in an on demand world.

- Resource reference list

  WebSphere Application Server has a large amount of existing documentation. Use the following user communities and other non-IBM sites that gather knowledge about using WebSphere products as a guideline to find the documentation that you require.
  - http://www.websphere-world.com/
  - http://www.websphere.org/
  - http://www.webspherepro.com/wphome/
  - http://www.sys-con.com/websphere/
  - http://websphereadvisor.com/

See also the documentation for the type of application that you are developing, such as Web applications, EJB applications, Web services applications, or applications that use messaging. Many sections contain *Web resources for learning* topics that bring attention to specific documents that become available.

Reference: Generated API documentation

The generated API documentation provides the details of the supported WebSphere Application Server application programming interfaces (APIs).

The generated API documentation is available in the information center table of contents:

- **Reference > Developer > API documentation** for developing J2EE applications to deploy on the application server
- **Reference > Administrator > API documentation** for extending the administrative infrastructure

To open the information center table of contents to the location of this reference information, click the **Show in Table of Contents** button ( ) on your information center border.

API documentation is organized by the package and class name, for easy lookup.

The API documentation is displayed in the content frame of the information center. If you would like more room to view the API documentation, double-click the gray bar located above the content area. This will expand the content area, while hiding the navigation area (the area containing the table of contents or search results list). Double-click the gray bar again when you are ready to restore the navigation area.

Chapter 3. Obtaining an integrated development environment (IDE)

This topic describes obtaining an integrated development environment (IDE). Use Rational products from IBM to design, construct, and manage changes to applications for deployment on your WebSphere Application Server products.

- See Packaging for a description of the Rational Application Developer Trial that is shipped with the product.

  Insert the disc and use the documentation and the installation program on the disc to install and set up the trial development environment.

- See Assembly tools for a description of the Application Server Toolkit that is shipped with product.

- Refer to these Web resources for learning.

  Rational software pages on ibm.com
  > Browse IBM's portfolio of software for requirements analysis and tracking, application design and construction, ensuring software quality, configuration and change management, and development project management.

  Rational on developerWorks
  > This page provides quick links to technical resources and best practices for Rational software. Browse information by product or by technology. Find resources for learning, support, and developer communities.

  developerWorks main page
  > This page is the entrance to IBM's resource for developers.

Chapter 4. Developing WebSphere applications

Web applications

Task overview: Developing and deploying Web applications

A developer creates the files comprising a Web application, and then assembles the Web application components into a Web module. Next, the deployer (typically the developer in a unit-testing environment or the administrator in a production environment) installs the Web application on the server.

1. **(Optional)** Migrate existing Web applications to run in the new version of WebSphere Application Server.

2. Design the Web application and develop its code artifacts: Servlets, JavaServer Pages (JSP) files, and static files, as for example, images and Hyper Text Markup Language (HTML) files. See the "Web applications: Resources for learning" on page 66 topic for links to design documentation.

   JavaServer Pages programming tips:
   - Disable session state of JavaServer Pages files using `<%@ page language="java" contentType="text/html" session="false" %>` instead of `<%@ page language="java" contentType="text/html" %>`
   - Replace setProperties calls in your JavaServer Pages files with direct calls to the appropriate setxxx methods.

3. Develop the Web application, using WebSphere Application Server extensions to enhance its functionality.

4. Assemble the Web application into a Web module using an assembly tool. Web module assembly properties might include the ability to:
   - Configure servlet page lists.
   - Configure servlet filters.
   - Serve servlets by class name.
   - Enable file serving.

5. Deploy the Web module or application module that contains the Web application.

   Following deployment, you might find it handy to use the tool that enables batch compiling of the JSP files for quicker initial response times.

6. **(Optional)** Troubleshoot your Web application.

7. **(Optional)** Modify the default Web container configuration in the application server in which you deployed the Web module or application module containing the Web application.

8. **(Optional)** Manage the deployed Web application.

Web applications

A Web application is comprised of one or more related servlets, JavaServer Pages technology (JSP files), and Hyper Text Markup Language (HTML) files that you can manage as a unit.

The files in a Web application are related in that they work together to perform a business logic function. For example, one of the WebSphere Application Server samples is a *Simple Greeting* Web application. This application, comprised of a servlet and Web pages, greets new users when they access the application.

The Web application is a concept supported by the Java Servlet Specification. Web applications are typically packaged as `.war` files.

web.xml file

The web.xml file provides configuration and deployment information for the Web components that comprise a Web application. Examples of Web components are servlet parameters, servlet and JavaServer Pages (JSP) definitions, and Uniform Resource Locators (URL) mappings.

The Java Servlet 2.4 specification defines the web.xml deployment descriptor file in terms of an XML schema document. For backwards compatibility of applications written to the Java Servlet 2.2 Specification, Web containers are also required to support the Java Servlet 2.2 specification. For backwards compatibility of applications written to the Java Servlet 2.3 specification, Web containers are also required to support the Java Servlet 2.3 specification.

If you use Rational Application Developer version 6 to create your portlets, you must remove the following reference to the std-portlet.tld from the web.xml file:

```
<taglib id="PortletTLD">
   <taglib-uri>http://java.sun.com/portlet</taglib-uri>
   <taglib-location>/WEB-INF/tld/std-portlet.tld</taglib-location>
  </taglib>
```

Location

The web.xml file must reside in the `WEB-INF` directory under the context of the hierarchy of directories that exist for a Web application.

For example, if the application is client.war, then the web.xml file is placed in the *install_root*/*client war*/`WEB-INF` directory.

Usage notes

*   Is this file read-only?

    No
*   Is this file updated by a product component?

    This file is updated by the Application Server Toolkit.
*   If so, what triggers its update?

    The Application Server Toolkit 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.

Sample file entry

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" 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>Servlet 2.4 application</display-name>
 <filter>
  <filter-name>ServletMappedDoFilter_Filter</filter-name>
  <filter-class>tests.Filter.DoFilter_Filter</filter-class>
  <init-param>
   <param-name>attribute</param-name>
   <param-value>tests.Filter.DoFilter_Filter.SERVLET_MAPPED</param-value>
   </init-param>
 </filter>
 <filter-mapping>
  <filter-name>ServletMappedDoFilter_Filter</filter-name>
  <url-patter>/DoFilterTest</url-pattern>
  <dispatcher>REQUEST</dispatcher>
 </filter-mapping>
 <filter-mapping>
```

```
 <filter-name>ServletMappedDoFilter_Filter</filter-name>
 <url-patter>/IncludedServlet</url-pattern>
 <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
<filter-mapping>
 <filter-name>ServletMappedDoFilter_Filter</filter-name>
 <url-patter>ForwardedServlet</url-pattern>
 <dispatcher>FORWARD</dispatcher>
</filter-mapping>
<listener>
 <listener-class>tests.ContextListener</listener-class>
</listener>
<listener>
 <listener-class>tests.ServletRequestListener.RequestListener</listener-class>
</listener>
<servlet>
 <servlet-name>welcome</servlet-name>
 <servlet-class>WelcomeServlet</servlet-class>
</servlet>
<servlet>
 <servlet-name>ServletErrorPage</servlet-name>
 <servlet-class>tests.Error.ServletErrorPage</servlet-class>
</servlet>
<servlet>
 <servlet-name>IncludedServlet</servlet-name>
 <servlet-class>tests.Filter.IncludedServlet</servlet-class>
</servlet>
<servlet>
 <servlet-name>ForwardedServlet</servlet-name>
 <servlet-class>tests.Filter.ForwardedServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>welcome</servlet-name>
 <url-pattern>/hello.welcome</url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>ServletErrorPage</servlet-name>
 <url-pattern>/ServletErrorPage</url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>IncludedServlet</servlet-name>
 <url-pattern>/IncludedServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>ForwardedServlet</servlet-name>
 <url-pattern>/ForwardedServlet</url-pattern>
</servlet-mapping>
<welcome-file-list>
 <welcome-file>hello.welcome</welcome-file>
</welcome-file-list>
<error-page>
 <exception-type>java.lang.ArrayIndexOutOfBoundsException</exception-type>
 <location>/ServletErrorPage</location>
</error-page>
</web-app>
```

Default Application

WebSphere Application Server provides a default configuration that allows administrators to easily verify
that the Application Server is running. When the product is installed, it includes an application server called
server1 and an enterprise application called *Default Application*.

Default Application contains a Web module called *DefaultWebApplication* and an enterprise bean Java
archive (JAR) file called *Increment*. The *Default Application* provides a number of servlets, described
below. These servlets are available in the product.

For additional code examples, visit the Samples Gallery. Learn how to locate and install the Samples Gallery by viewing the Samples Gallery reference page.

Snoop servlet

Use the Snoop servlet to retrieve information about a servlet request. This servlet returns the following information:
- Servlet initialization parameters
- Servlet context initialization parameters
- URL invocation request parameters
- Preferred client locale
- Context path
- User principal
- Request headers and their values
- Request parameter names and their values
- HTTPS protocol information
- Servlet request attributes and their values
- HTTP session information
- Session attributes and their values

The Snoop servlet includes security configuration so that when WebSphere Security is enabled, clients must supply a user ID and password to initiate the servlet.

The URL for the Snoop servlet is: `http://localhost:9080/snoop/`.

HelloHTML servlet

Use the HelloHTML pervasive servlet to exercise the PageList support provided by the WebSphere Web container. This servlet extends the PageListServlet, which provides APIs that allow servlets to call other Web resources by name or, when using the *Client Type detection* support, by type.

You can invoke the Hello servlet from an HTML browser, speech client, or most Wireless Application Protocol (WAP) enabled browsers using the URL: `http://localhost:9080/HelloHTML.jsp`.

transition: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 6.1 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.

HitCount application

Use the HitCount demonstration application to demonstrate how to increment a counter using a variety of methods, including:
- A servlet instance variable
- An HTTP session
- An enterprise bean

You can instruct the servlet to execute any of these methods within a transaction that you can commit or roll back. If the transaction is committed, the counter is incremented. If the transaction is rolled back, the counter is not incremented.

The enterprise bean method uses a container-managed persistence enterprise bean that persists the counter value to a Cloudscape database. This enterprise bean is configured to use the Default Datasource, which is set to the DefaultDB database.

When using the enterprise bean method, you can instruct the servlet to look up the enterprise bean, either in the WebSphere global namespace, or in the namespace local to the application.

The URL for the HitCount application is: `http://localhost:9080/HitCount.jsp`.

Servlets

Servlets are Java programs that use the Java Servlet Application Programming Interface (API). You must package servlets in a Web archive (WAR) file or Web module for deployment to the 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.

Servlets can support dynamic Web page content, provide database access, serve multiple clients at one time, and filter data.

For the purposes of WebSphere Application Server, discussions of servlets focus on Hyper Text Transfer Protocol (HTTP) servlets, which serve Web-based clients.

With the introduction of Java Servlet 2.4 specification, you can define servlets as welcome files. Non-servlet resources are served only when the FileServingEnabled attribute is set to true. Serving welcome files is connected to serving static content, therefore fileServing enabled is set in the Web module.

JavaServer Pages

JavaServer Pages (JSP) are application components coded to the JavaServer Pages Specification. JavaServer Pages enable the separation of the Hypertext Markup Language (HTML) code from the business logic in Web pages so that HTML programmers and Java programmers can more easily collaborate in creating and maintaining pages.

JSP files support a division of roles:

HTML authors
  Develop JSP files that access databases and reusable Java components, such as servlets and beans.
Java programmers
  Create the reusable Java components and provide the HTML authors with the component names and attributes.
Database administrators
  Provide the HTML authors with the name of the database access and table information.

WebSphere Application Server Version 6.1 supports the JSP 2.0 specification. The sub-topics below discuss WebSphere Application Server's JSP 2.0 implementation, focusing on configuration, tools and extensions.

JSP class file generation: At runtime, the WebSphere Application Server JavaServer Pages (JSP) engine loads JSP class files from either the WebSphere Application Server `temp` directory or a Web module's `WEB-INF/classes` directory. The WebSphere Application Server `temp` directory is typically *WAS_INSTALL_ROOT*`/AppServer/profiles/default/temp/`*node_name/server_name*. The JSP engine first searches for a class file in the `temp` directory and then it searches in the Web module's `WEB-INF/classes` directory. Figure 1 shows the processing logic of the JSP engine at runtime.
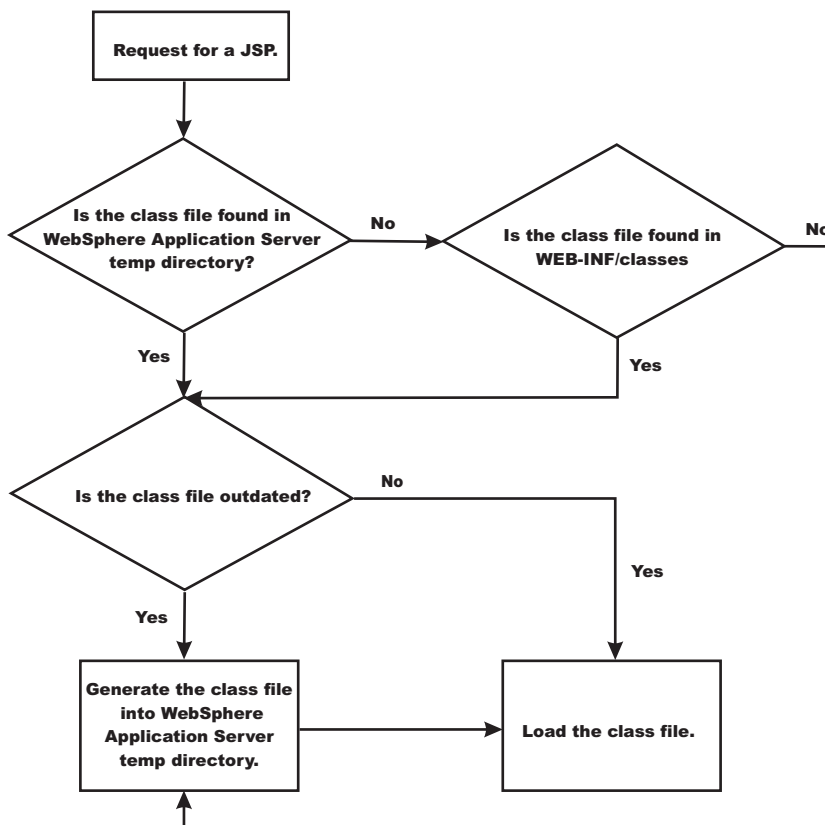
Request for a JSP.

Is the class file found in WebSphere Application Server temp directory?

No

Is the class file found in WEB-INF/classes

No

Yes

Yes

Is the class file outdated?

No

Yes

Yes

Generate the class file into WebSphere Application Server temp directory.

Load the class file.

The batch compiler supports the generation of class files in both the WebSphere Application Server `temp` directory and a Web module's `WEB-INF/classes` directory, depending on the type of batch compiler target. In addition, the batch compiler enables the generation of class files into any directory on the filesystem, outside of the target application. Generating class files into a Web module's `WEB-INF/classes` directory enables you to deploy the Web module as a self-contained Web archive (WAR) file, or a WAR file inside an enterprise archive (EAR) file. The following table shows the batch compiler's behavior when compiling class files.

	ear.path or war.path supplied	enterpriseApp.name supplied
compileToDir not supplied; *compileToWebInf* not supplied, or is true	The class files are compiled into the Web module's `WEB-INF/classes` directory.	The class files are compiled into the Web module's `WEB-INF/classes` directory.
compileToDir not supplied; *compileToWebInf* is false	The class files are compiled into the Web module's `WEB-INF/classes` directory.	The class files are compiled into the WebSphere Application Server temp directory, usually `{WAS_ROOT}/profiles/profilename/temp`
compileToDir is supplied; *compileToWebInf* not supplied, or is either true or false	The class files are compiled into the directory indicated by *compileToDir*.	The class files are compiled into the directory indicated by *compileToDir*.

Packages and directories for generated .java and .class files:

By default, the .java files for all JavaServer Pages (JSP) files are generated with the package statement, `package com.ibm._jsp;`. The JSP engine's class loader knows how to load JSP classes when they are all in the same package. The .java files are located in the filesystem within a directory structure mirroring the JSP source directory structure.

If the JSP engine configuration parameter **useFullPackageNames** is set to true, the .java files are generated with the package statement

```
Package _ibmjsp.<directory structure in which the jsp is located>;
```

The usage of full package names enables the configuration of a JSP as a servlet in the `web.xml` file. See "JSP class loading" on page 36 for more information. The table below gives examples of packages and directory structures for generated .java and .class files.

	Java package		Location of .java or .class files in file system	
JSP file	**default**	**useFullPackageNames=true**	**default**	**useFullPackageNames=true**
/myJsp.jsp	com.ibm._jsp	_ibmjsp	/	/_ibmjsp
/jspFiles/ jspOne.jsp	com.ibm._jsp	_ibmjsp.jspFiles	/jspFiles	/_ibmjsp/jspFiles
/dir with spaces/jspTwo.jsp	com.ibm._jsp	_ibmjsp.dir_20_with_20_spaces	/dir with spaces	/_ibmjsp/ dir_20_with_20_spaces

Generated .java files: When the JSP engine's **keepgenerated** configuration parameter is set to true, the `.java` file that is generated for JavaServer Pages (JSP) is retained. This file contains information that is useful in debugging.

Dependency information

In the `.java` file, immediately following the class declaration, an array of dependent files is defined, if the source JSP has any dependencies. There are three types of files that are tracked as dependencies:

1. Files that are statically included in the JSP
2. Tag files that are used by the JSP, but only tag files that are not in Java Archive (JAR) files
3. TLD files that are used by the JSP, but only TLDs that are not in JAR files

This array is always generated, but the JSP engine uses it, in determining whether a JSP needs to be recompiled, only when the trackDependencies parameter is set to true.

In the example below, three JSP fragments, one TLD and one tag file are dependencies of the JSP jsp1.jsp. There are three parts to each array entry:

1. The path to the dependency, relative to the Web module's context root. For example:
   /dir1/frag1.jspf
2. The long value representing the time the file was last modified. For example: 1082407108000
3. The String representation of the long value. For example: Mon Apr 19 16:38:28 EDT 2004

```
public final class _jsp1 extends com.ibm.ws.jsp.runtime.HttpJspBase
  implements com.ibm.ws.jsp.runtime.JspClassInformation {

  private static String[] _jspx_dependants;
  static {
  _jspx_dependants = new String[5];
  _jspx_dependants[0] = "/Banner.jspf^1082407108000^Mon Apr 19 16:38:28 EDT 2004";
  _jspx_dependants[1] = "/Footer.jspf^1077657462000^Tue Feb 24 16:17:42 EST 2004";
  _jspx_dependants[2] = "/dir1/frag1.jspf^1035396680000^Wed Oct 23 14:11:20 EDT 2002";
  _jspx_dependants[3] = "/utility.tld^1080069938000^Tue Mar 23 14:25:38 EST 2004";
  _jspx_dependants[4] = "/WEB-INF/tags/top.tag^1065440490000^Mon Oct 06 07:41:30 EDT 2003";
  }
```

Version, JSP engine options, and WEB.XML information

The generated .java source contains a comment that lists information about the file which is located at the bottom of the generated file. This information includes:

- The date and time the `.java` file was generated
- The version, build number and build date of the WebSphere Application Server on which the `.java` file was generated
- The values of the JSP engine configuration parameters that were in effect when the file was generated
- The values of any `<jsp-config>` elements in the `web.xml` file that pertained to the source JSP file.

```
/*
profile_root/AppSrv01/installedApps/MyCell/sampleApp.ear/examples.war/WEB-INF/classes/_ibmjsp/_jsp1.java
was generated @ Wed May 03 10:05:56 EDT 2006IBM WebSphere Application Server - ND, 6.1.0.0
    Build Number: o0441.04
    Build Date: 05/01/06*********************************************************
The JSP engine configuration parameters were set as follows:

classDebugInfo =          [false]
debugEnabled =            [false]
deprecation =            [false]
compileWithAssert =       [false]
jdkSourceLevel =          [13]disableJspRuntimeCompilation =[false]
extendedDocumentRoot =    [null]
ieClassId =               [clsid:8AD9C840-044E-11D1-B3E9-00805F499D93]
keepGenerated =           [true]




outputDir =               [C:/WebSphere_6.0/AppServer/profiles/AppSrv01/installedApps/MyCell/
                           sampleApp.ear/examples.war/WEB-INF/classes]
reloadEnabled =           [true]
reloadEnabledSet =        [true]
reloadInterval =          [5000]
trackDependencies =       [false]
usePageTagPool =          [false]
useThreadTagPool =        [true]
useImplicitTagLibs =      [true]
verbose =                 [false]
looseLibMap =             [null]
useJikes =                [false]
useFullPackageNames =     [true]
translationContextClass = [null]
extensionProcessorClass = [null]
javaEncoding =            [UTF-8]
autoResponseEncoding =    [false]

*********************************************************
The following JSP Configuration Parameters were obtained from web.xml:

prelude list = [[]]
coda list = [[]]
elIgnored = [false]
pageEncoding = [null]
isXML = [false]
scriptingInvalid = [false]
*/
```

JSP class loading:

You can configure a JavaServer Pages (JSP) class to be loaded by either the JSP engine's class loader or by the Web module's class loader.

By default, a JSP class is loaded by a unique instance of the JSP engine's class loader. The JSP engine's class loader enables reloading at runtime of a JSP class when the JSP source or one of its dependents is modified. This allows you to reload a single JSP class when necessary, without affecting any other loaded JSP classes.

JSP classes are loaded by the Web module's class loader under either of the following scenarios.

1. 1. The JSP engine configuration parameter useFullPackageNames is set to true, and the JSP file is configured as a servlet in the `web.xml` file using the <servlet-class> scenario in the table below.

2. 2. The JSP engine configuration parameters useFullPackageNames and disableJspRuntimeCompilation are both set to true. In this case, you do not need to configure a JSP file does as a servlet in the `web.xml` file.

Configuring JSP files as Servlets

You can configure a JSP file as a servlet in the `web.xml` file. There are two ways to do this. They are described in the table below.

Before you configure a JSP file as a servlet, consider the following.

1. Reloading capability - If runtime reloading of JavaServer Pages files is desired, requests for JavaServer Pages files must be handled by the JSP engine. The <servlet-class> scenario in the table below disables runtime JSP file reloading, while the <jsp-file> scenario is compatible with reloading.

2. Reducing the number of class loaders - If you do not require runtime reloading of modified JSP pages and you want to reduce the number of class loader instances, then you can use the <servlet-class> scenario in the table below. Similarly, scenario 2 in section 1 above can be used without having to configure a JSP file as a servlet.

Scenario	Example	compatible with runtime reloading	multiple class loaders used?	useFullPackageNames
<jsp-file>	<servlet> <servlet-name>jspOne</servlet-name> <jsp-file>jspOne.jsp</jsp-file> </servlet>	Yes	Yes	Can be true or false
<servlet-class>	<servlet> <servlet-name>jspTwo</servlet-name> <servlet-class>_ibmjsp.jspTwo</servlet-class> </servlet>	No	No	Must be true

The JSP batch compiler tool helps you configure JavaServer Pages files as servlets. When useFullPackageNames is true, the JSP batch compiler generates <servlet> and <servlet-mapping> elements for each JSP file that it successfully translates and compiles. The elements are written to a `web.xml` fragment file named `generated_web.xml` which is located in the binaries `WEB-INF` directory of a Web module processed by the JSP file batch compiler (this directory is located within the deployed application's ear file). You can copy and paste all or some of these elements into the `web.xml` file to configure JavaServer Pages files as servlets.

Take note of the location of the `web.xml` that is used by the application server. The application specific configuration is obtained from either the application binaries (the application's ear file) or from the configuration repository. If an application is deployed into WebSphere Application Server with the flag Use Binary Configuration set to true, then the `WEB-INF/web.xml` file is looked for in a Web module's binaries directory, not in the configuration repository. Below are examples of these two locations.

- An example of a configuration repository directory is `{WAS_ROOT}/profiles/profilename/config/cells/`*cellname*`/applications/`*enterpriseappname*`/deployments/`*deployedname*`/webmodulename*
- An example of an application binaries directory is: `{WAS_ROOT}/profiles/`*profilename*`/installedApps/`*nodename*`/EnterpriseAppName/`*WebModuleName*`/`

If the JSP batch compiler is executed on a pre-deployed application then the `web.xml` file is in the Web module's `WEB-INF` directory.

Configuring JSP run time reloading: JSP files can be translated and compiled at run time when the JSP file or its dependencies are modified. This is known as JSP reloading. JSP reloading is enabled through the **reloadEnabled** JSP engine parameter in the `WEB-INF/ibm-web-ext.xmi` file:

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

The following table contains the recommended reload settings for production and development environments.

Configuration Attribute	Recommended settings	
	Production Environment	Development Environment
reloadEnabled	false	true
reloadInterval	n/a (ignored if reloadEnabled is false)	approximately 5 seconds
trackDependencies	n/a (ignored if reloadEnabled is false)	true Alternatively, set this to false to improve response time if dependencies are not changing
disableJspRuntimeCompilation	true - Alternatively, set this to false if JSP files are not pre-compiled and therefore need to be compiled on the first request.	false

If the **reloadEnabled** parameter is set to true, a JSP file is reloaded at run time if the JSP file and its class file do not have the same timestamp. In addition, if **trackDependencies** is set to true then the JSP file is reloaded if the timestamp of any of its dependencies has changed since the JSP class file was last generated. If the **reloadEnabled** parameter is set to false, a JSP file is still compiled if necessary on the first request to it unless the parameter **disableJspRuntimeCompilation** is true. For example, when **disableJspRuntimeCompilation** is false and **reloadEnabled** is false, a JSP file is compiled on the first request if the class file is outdated. It would not compiled on subsequent requests even if the JSP source file is modified or the class file is deleted unless **reloadEnabled** is true

Reload interval

The reload interval is set through the **reloadInterval** JSP engine parameter:

`<jspAttributes xmi:id="JSPAttribute_1" name="reloadInterval" value="5"/>`

If reloading is enabled, the **reloadInterval** parameter value 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 five seconds prior to the current request for the JSP file. Once the **reloadInterval** is exceeded, reload checking is performed and the reload interval

timer is reset to 0 for that JSP file. The larger the **reloadInterval**, the less frequently the JSP engine checks for the need to reload a JSP file.

Dependency tracking

Dependency tracking is set through the **trackDependencies** JSP engine parameter:

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

If reloading is enabled, the **trackDependencies** parameter value determines whether the JSP engine tracks modifications to the requested JSP file dependencies as well as to the JSP file itself. The three types of dependencies tracked by the JSP engine are:

- files statically included in the JSP file
- tag files that are referenced in the JSP file (excluding tag files that are in JAR files)
- TLDs that are referenced in the JSP file (excluding TLDs that are in JAR files)

Dependency tracking information is always included in the generated class file even if **trackDependencies** is false. The information is not used by the JSP engine or batch compiler unless the **trackDependencies** parameter is true. This means that you can enable dependency tracking without having to recompile JSP files.

For example, the `toplevel.jsp` file statically includes the `footer.jspf` file. When the `toplevel.jsp` file is compiled, the path to the `footer.jspf` file and its timestamp are stored in the `toplevel.jsp`'s class file. As a result, the `footer.jspf` file is modified and the `toplevel.jsp` file is requested. Now that the reload interval for the `toplevel.jsp` file has been exceeded, the JSP engine compares the timestamp stored in the class file with the `footer.jspf` file timestamp on disk. Because the timestamps are different, the `toplevel.jsp` file is compiled, picking up the modification to the `footer.jspf` file. In order for dependency tracking to work, the **trackDependencies** value must be set to true at the time a JSP file is requested at run time or is processed by the batch compiler.

Disabling compilation

Disablement of run time compilation of JavaServer Pages is set via the disableJspRuntimeCompilation JSP engine parameter:

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

If the **disableJspRuntimeCompilation** parameter is set to true, the JSP engine at run time 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 for the class files to be loaded. With this option set to true, an application can be installed without JSP source, but must have precompiled class files. There is a Web container custom property of the same name that can be 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. Setting the **disableJspRuntimeCompilation** parameter to true automatically sets **reloadEnabled** to false.

Reload processing sequence

The processing sequence pertaining to JSP file reloading when **trackDependencies** is false is shown in Figure 1.
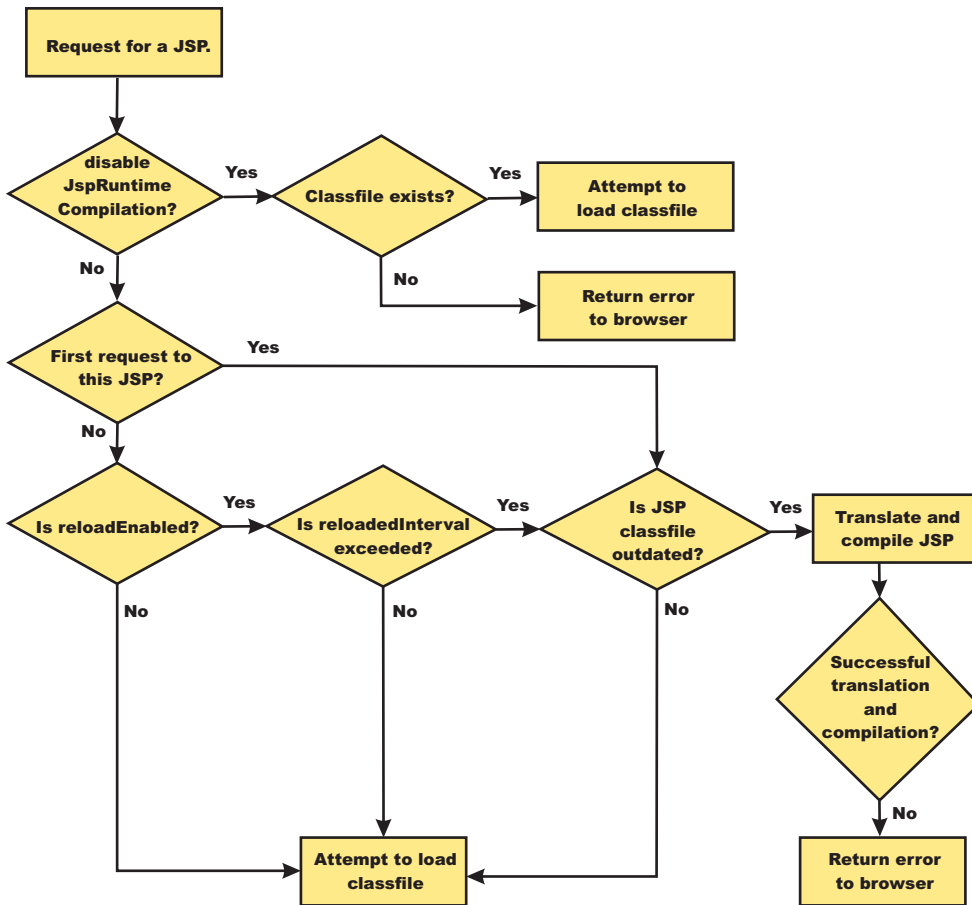
Figure 1. Reload processing sequence when **trackDependencies** *is false.*

When **trackDependencies** is true, the JSP engine does additional file system processing to determine if any of a JSP file's dependencies have changed since the JSP file was last translated and compiled. Figure 2 shows the additional processes that are performed on the 'No' path of flow chart labeled "is JSP class file outdated?". You can see that the path taken when disableJspRuntimeCompilation is true is the most efficient path.
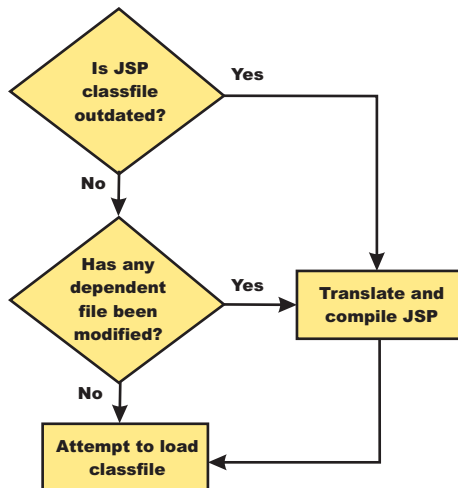
Figure 2. Additional reload processing performed when **trackDependencies** *is true.*

JSP reload options for Web modules settings:

Use this panel to configure the class reloading of Web modules such as JavaServer Pages (JSP) files

To view this administrative console panel, click **Applications > Enterprise Applications >** *application_name* **> JSP reload options for Web modules**. This panel is the same as the **Provide JSP reloading options for Web modules** panel on the application installation and update wizards.

Web module:

Specifies the name of a JSP file in the application.

URI:

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

JSP enable class reloading:

Specifies whether to enable class reloading when JSP files are updated.

A Web container reloads JSP files only when the IBM extension jspReloadingEnabled in the jspAttributes of the `ibm-web-ext.xmi` file is set to `true`.

JSP reload interval in seconds:

Specifies the number of seconds to scan the application's file system for updated JSP files. The default is the value of the reloading interval attribute in the IBM extension (`META-INF/ibm-application-ext.xmi`) file of the EAR file.

To enable reloading, specify a value greater than zero (for example, 1 to 2147483647). To disable reloading, specify zero (0). The range is from 0 to 2147483647.

The reloading interval attribute takes effect only if class reloading is enabled.

Disabling JavaServer Pages run time compilation: By default, the JavaServer Pages (JSP) engine translates a requested JSP file, compiles the `.java` file, and loads the compiled servlet into the run time environment. You can change the JSP engine default behaviour by indicating a JSP file should never be translated or compiled at run time, even when a `.class` file does not exist.

If run time compilation is disabled, you must precompile the JSP files, which provides the following advantages:

- Reduces compilation related disk operations.
- Minimizes disk storage requirements necessary for handling temporary `.java` files generated during a run time compilation.
- Allows you to not include the JSP source files in the application.
- Allows verification that a JSP file compiled successfully before deploying and installing the application in WebSphere Application Server.

You can disable run time JSP file compilation on a global or an individual Web application basis:

- To disable the translation and compilation of JSP files for all Web applications, set the Web container custom property `disableJspRuntimeCompilation` to `true`.

  Set this property through the Web container `Custom properties` panel in the administrative console. To view this administrative console page, click:

  Servers > **Application servers** > *server_name* > **Web container settings** >
    Web container > **Custom properties** > *property_name*

  Valid values for this setting are `true` or `false`. If this property is set to `true`, then translation and compilation of the JSP files is disabled at run time for all Web applications.
- To disable the translation and compilation of JSP files for a specific Web application, set the JSP engine initialization parameter `disableJspRuntimeCompilation` to `true`. This setting, if enabled, determines the run time behavior of the JSP engine and overrides the Web container custom property setting.

  Set this parameter through the `JavaServer Pages attribute assembly settings` panel in the Chapter 7, "Assembling applications," on page 1345.

  Valid values for this setting are `true` or `false`. If this parameter is set to `true`, then, for that specific Web application, translation and compilation of the JSP files is disabled at run time, and the JSP engine only loads precompiled files.
- If neither the Web container custom property nor the JSP parameter is set, the first request for a JSP file results in the translation and compilation of the JSP file when the `.class` file does not exist or is outdated. Subsequent requests for the file also result in translations and compilations, but only if the following conditions are met:
  - Translations are required because the `.class` file is outdated.
  - Reloading is enabled for the Web module.
  - Reload interval is exceeded.

If you disable run time compilation and a request arrives for a JSP file that does not have a matching `.class` file, the JSP engine returns HTTP error 500 (Internal server error) to the browser. In this case, an exception is written to the System Out (SYSOUT) and First Failure Data Capture (FFDC) logs.

If a JSP file has a matching `.class` file but that file is out of date, the JSP engine still loads the `.class` file into memory.

Provide options to compile JavaServer Pages settings:

Use this panel to specify options to be used by the JavaServer Pages (JSP) compiler.

This administrative console panel is a step in the application installation and update wizards. To view this panel, you must select **Precompile JavaServer Pages files** on the **Select Installation options** panel. Thus, to view this panel, click **Applications > Install New Application >** *application_path* **> Show me all installation options and parameters > Next > Next > Precompile JavaServer Pages files > Next > Step: Provide options to compile JSPs**.

You can specify the JSP compiler options on this panel only when installing or updating an application that contains Web modules. After the application is installed, you must edit the JSP engine configuration parameters of a Web module's `WEB-INF/ibm-web-ext.xmi` file to change its JSP compiler options.

Web module:

Specifies the name of a module within the application.

URI:

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

JSP class path:

Specifies a temporary class path for the JSP compiler to use when compiling JSP files during application installation. This class path is not saved when the application installation is complete and is not used when the application is running. This class path is used only to identify resources outside of the application which are necessary for JSP compilation and which will be identified by other means (such as shared libraries) after the application is installed. In network deployment configurations, this class path is specific to the deployment manager machine.

To specify that multiple Web modules use the same class path:
1. In the list of Web modules, select the **Select** check box beside each Web module that you want to use a particular class path.
2. Expand **Apply Multiple Mappings**.
3. Specify the desired class path.
4. Click **Apply**.

Use full package names:

Specifies whether the JSP engine generates and loads JSP classes using full package names.

When full package names are used, precompiled JSP class files can be configured as servlets in the `web.xml` file, without having to use the `jsp-file` attribute. When full package names are not used, all JSP classes are generated in the same package, which has the benefit of smaller file-system paths.

When the options 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.

This option is the same as the useFullPackageNames JSP engine parameter.

JDK source level:

Specifies the source level at which the Java compiler compiles JSP Java sources. Valid values are `13`, `14`, and `15`. The default value is `13`, which specifies source level 1.3.

Disable JSP runtime compilation:

Specifies whether a JSP file should never be translated or compiled at run time, even when a `.class` file does not exist.

When this option is set to `true`, the JSP engine does not translate and compile JSP files at run time; the JSP engine loads only precompiled class files. JSP source files do not need to be present in order to load class files. You can install an application without JSP source, but the application must have precompiled class files.

For a single Web application class loader to load all JSP classes, this compiler option and the **Use full package names** option both must be set to `true`.

This option is the same as the disableJspRuntimeCompilation JSP engine parameter.

JSP batch compilation:

As an IBM enhancement to JavaServer Pages (JSP) support, IBM WebSphere Application Server provides a batch JSP compiler that allows JSP page compilation before application deployment. The batch compiler validates the syntax of JSP pages, translates the JSP pages into Java source files, and compiles the Java source files into Java servlet class files. The batch compiler also validates tag files and generates their Java implementation classes.

Batch compilation of JSP pages in a predeployed application simplifies the deployment process and improves the runtime performance of JSP page by eliminating first-request compilations. The batch compiler also operates on enterprise applications that have been deployed into WebSphere Application Server.

The JSP batch compiler works on Web modules that support Servlet 2.2 and up through Servlet 2.4 The batch compiler works on JSP pages written to the JSP 2.0 specification or previous specifications back to JSP 1.0. It recognizes a Servlet 2.4 deployment descriptor, `web.xml`, and can use any jsp-config elements that it may contain. In a Servlet 2.3 (JSP 1.2) or Servlet 2.2 (JSP 1.1) deployment descriptor the batch compiler recognizes and uses any taglib elements that the descriptor may contain.

Batch compiling makes the first request for a JSP page much faster because the JSP page is already translated and compiled into a servlet. Batch compiling is also useful as a fast way to resynchronize all of the JSP pages for an application.

The batch compiler supports the generation of class files in both the WebSphere Application Server `temp` directory and a Web module's `WEB-INF/classes` directory, depending on the type of batch compiler target. In addition, the batch compiler enables generation of class files into any directory on the filesystem, outside the target application. Generating class files into a Web module's `WEB-INF/classes` directory enables the Web module to be deployed as a self-contained WAR file, or a WAR inside an EAR.

Also, you can use shared libraries with the JSP batch complier. When you use the JSP batch compiler, you must either add the JAR to the WAR in the <WEB-INF>/lib directory, or add the JAR to the JVM class path to use shared libraries.

JSP batch compiler tool: The batch compiler validates the syntax of JSP pages, translates the JSP pages into Java source files, and compiles the Java source files into Java Servlet class files. The batch compiler also validates tag files and generates their Java implementation classes. Use this function to batch compile your JSP files and thereby enable faster responses to the initial client requests for the JSP files on your production Web server.

The batch compiler can be executed against compressed or expanded enterprise archive (EAR) files and Web application archive (WAR) files, as well as enterprise applications and Web modules that have been deployed into WebSphere Application Server. When the target is a deployed enterprise application, the server does not need to be running to execute the batch compiler. If the batch compiler is executed while the target sever is running, the server is not aware of an updated class file and does not load that class file unless the enterprise application is restarted. When the target is a compressed EAR file or WAR file, the batch compiler must expand it before executing.

Processing of Web modules

The batch compiler operates on one Web module at a time. If the target is either an EAR file or an installed enterprise application that contains more than one Web module, the batch compiler operates on each Web module individually. This is done because JSP pages are configured on a Web module basis, through the Web module's web.xml deployment descriptor file. Within a Web module, the batch compiler processes one directory at a time. It validates and translates each JSP page individually, and then invokes

the Java compiler for the entire group of generated Java sources files in that directory. If one JSP page fails during the Java compilation phase, the Java compiler might not create class files for most or all of the JSP pages that successfully compiled in that directory.

JSP file extensions

The batch compiler uses four things to determine what file extensions it should process:
1. Standard JSP file extensions
   - *.jsp
   - *.jspx
   - *.jsw
   - *.jsv
2. The url-pattern property of the jsp-property-group elements in the deployment descriptor file in Servlet 2.4 Web modules
3. The **jsp.file.extensions** JSP engine configuration parameter (for pre-Servlet 2.4 Web modules)
4. The batch compiler configuration parameter **jsp.file.extensions**

The standard extensions are always used by the batch compiler. If the Web module contains a Servlet 2.4 deployment descriptor, the batch compiler also processes any url-patterns found within the jsp-config element. If the batch compiler target contains the JSP engine configuration parameter **jsp.file.extensions**, then those extensions are also processed. If the batch compiler configuration parameter **jsp.file.extensions** is present, the extensions given are also processed and will override the JSP engine configuration parameter **jsp.file.extensions**.

It is a good idea to give JSP 'fragments' an extension that is not processed by the batch compiler. Statically-included fragments that do not stand alone generate translation or compilation errors if processed. The JSP 2.0 Specification suggests that you use the extension .jspf for such files.

Batch compiler command

Both a Windows batch file, JspBatchCompiler.bat and UNIX shell script JspBatchCompiler.sh for running the batch compiler from the command line are found in the {WAS_ROOT}/bin directory. An Ant task is also available for executing the batch compiler using Ant. See the topic, Batch Compiler Ant Task for additional information.

The batch compiler target is the only required parameter. The target is one of -ear.path, -war.path or -enterpriseapp.name.

```
JspBatchCompiler -ear.path | -war.path | -enterpriseapp.name <name>
    [-response.file <filename>]
    [-webmodule.name <name>]
    [-filename <jsp name | directory name>
    [-recurse <true | false>]
    [-config.root <path>]
    [-cell.name <name>]
    [-node.name <name>]
    [-server.name <name>]
    [-profileName <name>]
    [-extractToDir <path>]
    [-compileToDir <path>]
    [-compileToWebInf <true | false>]
    [-translate <true | false>]
    [-compile <true | false>]
    [-removeTempDir <true | false>]
    [-forceCompilation <true | false>]
    [-useFullPackageNames <true | false>]
    [-trackDependencies <true | false>]
    [-createDebugClassfiles <true | false>]
```

```
[-keepgenerated <true | false>]
[-keepGeneratedclassfiles <true | false>]
[-usePageTagPool <true | false>]
[-useThreadTagPool <true | false>]
[-classloader.parentFirst <true | false>]
[-classloader.singleWarClassloader <true | false>]
[-additional.classpath <classpath to additional JAR files and classes>]

[-verbose <true | false>]
[-deprecation <true | false>]
[-javaEncoding <encoding>
[-jdkSourceLevel <13 | 14 | 15>]
[-compilerOptions <space-separated list of java compiler options>]
[-useJikes <true | false>]
[-jsp.file.extensions <file extensions to process>]
[-log.level <SEVERE | WARNING | INFO | CONFIG | FINE | FINER | FINEST | OFF>]

*** See batchcompiler.properties.default in WAS_ROOT/bin for more information. ***
*** See JspCBuild.xml in WAS_ROOT/bin for information about the public WebSphere Ant task JspC. ***
```

The batch compiler is aware of three groups of configuration parameters:

1. JSP engine configuration parameters for a Web module.

   See the topic, JSP engine configuration parameters.

2. Batch compiler response file configuration parameters.

   These are the parameters that are found in a batch compiler response file. See -response.file, below.

3. Batch compiler command line configuration parameters.

   These are the parameters entered on the command line when running the batch compiler.

The batch compiler looks at all three groups of configuration parameters in order to determine which value for a parameter is used when compiling JSP pages. When resolving the value for a given parameter, the precedence is:

1. If the parameter is found on the command line, its value is used.

2. If the parameter is not found on the command line, the batch compiler looks for the parameter in a response file named on the command line.

3. If no response file is named, or if the parameter is not found therein, the batch compiler looks for the parameter in the Web module's JSP engine configuration parameters.

If a configuration parameter is not found among these three groups, then a default value is used. The default values for the configuration parameters are given below along with the description of the parameters.

With one exception, these parameters are not case sensitive; -profileName is case sensitive. If the values specified for these arguments are comprised of two or more words separated by spaces, you must add quotation marks around the values.

The batch compiler does not create, or set the values of, equivalent JSP engine parameters. This means that if a JSP page in a deployed Web module is modified and is recompiled by the JSP engine at run time, the JSP engine's configuration parameters will determine the engine's behavior. For example, if you use the batch compiler to compile a Web module and you use the -useFullPackageNames true option, the JSP files will be compiled to support that option. But the JSP engine parameter useFullPackageNames must also be set to true in order for the JSP runtime to be able to load the compiled JSP pages. If JSP pages are modified in a deployed Web module, then the engine's parameters should be set to the same values used in batch compilation.

To use the JSP batch compiler, enter the following command on a single line at an operating system command prompt.

- **ear.path | war.path | enterpriseapp.name**

Represents the full path to a single compressed or expanded enterprise application archive (EAR) file or Web application archive (WAR) file, or the name of the deployed enterprise application that you want to compile. For example:

- – `JspBatchCompiler -ear.path c:\myproject\sampleApp.ear`
- – `JspBatchCompiler -war.path c:\myWars\examples.war`
- – `JspBatchCompiler -enterpriseapp.name myEnterpriseApp -webmodule.name my.war -filename aDir/main.jsp`

- **response.file**

  Specifies the path to a file that contains configuration parameters used by the batch compiler. The *response.file* is used only if it is given on the command line; it is ignored if it is present in a response file.

  In a default installation, the template response file, batchcompiler.properties.default, is found in the `{WAS_ROOT}/bin` directory. Copy this template to create your own response files containing defaults for the parameters in which you are interested. All the required and optional parameters (except response.file) can be configured in a response file. **For example:** JspBatchCompiler -response.file c:\myproject\batchc.props

  Default : null

- **webmodule.name**

  Represents the name of the specific Web module that you want to batch compile. If this argument is not set, all Web modules in the enterprise application are compiled. This parameter is used only when *ear.path* or *enterpriseapp.name* is given. This parameter is useful when JSP pages in a specific Web module within a deployed enterprise application need to be regenerated, because all shared library dependencies are picked up.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -webmodule.name myWebModule.war

  Default: All Web modules in an EAR file or enterprise application are compiled if this parameter is not given.

- **filename**

  Represents the name of a single JSP file that you want to compile. If this argument is not set, all files in the Web module are compiled. Alternatively, if *filename* is set to the name of a directory, only the JSP files in that directory and that directory's child directories are complied. The name is relative to the context root of the Web module.

  Example 1: If you want to compile the file, `myTest.jsp`, and it is found in `/subdir/myJSPs`, you would enter `-filename /subdir/myJSPs/myTest.jsp`.

  Example 2: If you want to compile all JSP files in `/subdir/myJSPs` and its child directories, you would enter `-filename subdir/myJSPs`.

  Default: All JSP files in the Web module are compiled. Entering `-filename /` is equivalent to the default.

- **recurse**

  Determines whether subdirectories beneath the target directory are processed. This parameter is used only when the *filename* parameter is given. Set value to `false` to process only the directory named *filename* parameter; and not its subdirectories.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -filename /subdir1 -recurse false.

  Default: true; All directories beneath the target directory are processed.

- **config.root**

  Specifies the location of the WebSpehere Application Server configuration directory. This parameter is used only when *enterpriseapp.name* is given.

  Default: `{WAS_ROOT}/profiles/profilename/config`

- **cell.name**

  Specifies the name of the cell in which the application is deployed. This parameter is used only when *enterpriseapp.name* is given.

  Default: The default is obtained from the profile script that is used. The symbolic name of this variable is WAS_CELL.

- **node.name**

  Specifies the name of the node in which the application is deployed. This parameter is used only when *enterpriseapp.name* is given.

  Default: The default is obtained from the profile script that is used. The symbolic name of this variable is WAS_NODE.

- **server.name**

  Represents the name of the server in which the application is deployed. This parameter is used only when *enterpriseapp.name* is given.

  Default: server1

- **profileName**

  Specifies the name of the profile you want to use. This parameter is used only when `enterpriseapp.name` is given.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -profileName AppServer-3

  Default: The default profile is used. This is obtained from the file `setupCmdLine` script in the `install_root`/bin directory. The symbolic name is DEFAULT_PROFILE_SCRIPT.

- **extractToDir**

  Specifies the directory into which predeployed enterprise archive (EAR) files and Web application archive (WAR) files will be extracted before the batch compiler operates on them. This parameter is ignored when *enterpriseapp.name* is given. The extractToDir parameter is used as described in the table below.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -extractToDir c:\myTempDir.

  Use-case: You must extract a compressed archive before it is batch compiled. You can also extract an expanded archive to a new directory as well. In both cases, extraction leaves the original archive untouched, which may be useful while development is underway.

  Default values:

	Expanded archive	Compressed archive
extractToDir supplied	The batch compiler extracts the archive to extractToDir before operating on it. If a file or directory of the same name as the archive already exists in the extractToDir, the batch compiler removes the archive completely before extracting that archive. If the batch compiler exits with no errors, it compresses the archive in place in the extractToDir, even if the original EAR file or WAR file was expanded. If errors are encountered during compilation, the EAR file or WAR file is left in the expanded state even if the original EAR file or WAR file was compressed.	
extractToDir not supplied	The batch compiler operates on the EAR file or WAR file in place (does not extract it to another directory) and the archive remains expanded after the batch compiler finishes.	The batch compiler extracts the archive to the directory returned by the JVM property ″java.io.tmpdir″. The rest of the behavior described above, when extractToDir is supplied, is the same in this case.

  The default is `server1`.

- **compileToDir**

  Specifies the directory into which JSP pages are translated into Java source files and compiled into class files. This directory can be anywhere on the filesystem, but the batch compiler's default behavior is usually adequate. The batch compiler's behavior when compiling class files is described in the table below

  Example:: JspBatchCompiler -enterpriseApp.name sampleApp -compileToDir c:\myTargetDir

  Use-case: This parameter enables you to generate the Java and class files into a directory outside of the target, which may be useful if you want to compare the newly generated files with their previous versions which remain untouched within the target.

Default values:

	ear.path or war.path supplied	enterpriseApp.name supplied
compileToDir not supplied; compileToWebInf not supplied, or is true	The class files are compiled into the Web module's `WEB-INF/classes` directory	The class files are compiled into the Web module's `WEB-INF/classes` directory.
compileToDir not supplied; compileToWebInf is false	The class files are compiled into the Web module's `WEB-INF/classes` directory.	The class files are compiled into the WebSphere Application Server temp directory (usually `{WAS_ROOT}/temp`).
compileToDir is supplied; compileToWebInf not supplied, or is either true or false	The class files are compiled into the directory indicated by compileToDir.	The class files are compiled into the directory indicated by compileToDir.

- **compileToWebInf**

  Specifies whether the target directory for the compiled JSP class files should be the Web module's `WEB-INF/classes` directory. This parameter is used only when *enterpriseApp.name* is given, and it is overridden by *compileToDir* if *compileToDir* is given.

  The batch compiler's default behavior is to compile to the Web module's `WEB-INF/classes` directory. The batch compiler's behavior when compiling class files is described in the table above.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -compileToWebInf false.

  Use-case: Set this parameter to false when enterpriseApp.name is supplied and you want the class files to be compiled to the WebSphere Application Server temp directory instead of the Web module's `WEB-INF/classes` directory. Recommendation: if this parameter is set to false, set forceCompilation to true if there are any JSP class files in the `WEB-INF/classes` directory.

  Default: true; see the table above.

- **forceCompilation**

  Specifies whether the batch compiler is forced to recompile all JSP resources regardless or whether the JSP page is outdated.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -forceCompilation true.

  Use-case: Especially useful when creating an archive for deployment, to make sure all JSP classes are up to date.

  Default: false

- **useFullPackageNames**

  Specifies whether the batch compiler generates full package names for JSP classes. The default is to generate all JSP classes in the same package. The JSP engine's class loader knows how to load JSP classes when they are all in the same package. The default has the benefit of generating smaller file-system paths. Full package names have the benefit of enabling the configuration of precompiled JSP class files as servlets in the `web.xml` file without use of the `jsp-file` attribute, resulting in a single class loader (the Web application's class loader) being 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 pages 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 page that is successfully translated and compiled. The information can optionally be copied into the Web module's `web.xml` file so that the JSP pages are loaded as servlets by the Web container. Note that if a JSP page is configured as a servlet in this way, no reloading of the JSP page is done at run time if the JSP page is modified. This is because the JSP page is treated as a regular servlet and requests for it do not pass through the JSP engine.

  Example: JspBatchCompiler –enterpriseApp.name sampleApp –useFullPackageNames true

  Use-case: Enables JSP classes to be loaded by a single class loader.

  Default: false

- **removeTempDir**

  Specifies whether the Web module's temp directory is removed. The batch compiler by default generates JSP class files into a Web module's `WEB-INF/classes` directory. JSP class files are generated into the `temp` directory at run time if a JSP page is modified and JSP reloading is enabled. By batch compiling all the JSP pages in a Web module and also removing the `temp` directory, disk resources are preserved. You can only use the removeTempDir parameter when -enterpriseApp.name is given.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -removeTempDir true.

  Use-case: Free up disk space by clearing out a Web application's `temp` directory.

  Default: false
- **translate**

  Specifies whether JSP pages are translated and compiled. Set translate to false if you do not want JSP pages to be translated and compiled. You must use this option in conjunction with `-removeTempDir` to tell the batch compiler to remove only the `temp` directory and to do no further processing.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -translate false -removeTempDir true.

  Use-case: Free up disk space by clearing out a Web application's `temp` directory, without invoking JSP processing.

  Default: true
- **compile**

  Specifies whether JSP pages go through the Java compilation phase. Set compile to `false` if you do not want JSP pages to go through the Java compilation phase.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -compile false

  Use-case: If you only want JSP pages to be syntax-checked, set -compile to false. You can set -keepgenerated to true if you want to see the `.java` files that are generated during the translation phase.

  Default: true
- **trackDependencies**

  Specifies whether the batch compiler recompiles a JSP page when any of its dependencies have changed, even if the JSP page itself has not changed. Tracking dependencies incurs a significant runtime performance penalty because the JSP Engine checks the filesystem on every request to a JSP page to see if any of its dependencies have changed. The dependencies tracked by WebSphere Application Server are :
  1. Files statically included in the JSP page
  2. Tag files used by the JSP page (excluding tag files that are in JAR files)
  3. TLD files used by the JSP page (excluding TLD files that are in JAR files)

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -trackDependencies true.

  Use-case: Useful in a development environment.

  Default: false
- **createDebugClassfiles**

  Specifies whether the batch compiler generates class files that contain SMAP information, as per JSR 45, **Debugging support for Other Languages**.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -createDebugClassfiles true

  Use-case: Use this parameter when you want to be able to debug JSP pages in your JSR 45-compliant IDE.

  Default: false
- **keepgenerated**

  Specifies whether the batch compiler saves or erases the generated Java source files created during the translation phase.

  If set to `true`, WebSphere Application Server saves the generated `.java` files used for compilation on your server. By default, this argument is set to `false` and the `.java` files are erased after the class files have compiled.

Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -keepgenerated true

Use-case: Use this parameter when you want to review the Java code generated by the batch compiler.

Default: false

- **keepGeneratedclassfiles**

  Specifies whether the batch compiler saves or erases the class files generated during the compilation of Java source files.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -keepGeneratedclassfiles false -keepgenerated false

  Use-case: Set this parameter to false if you only want to see if there are any translation or compilation errors in your JSP pages. If paired with -keepgenerated false, this parameter results in all generated files being removed before the batch compiler completes.

  Default: true

- **usePageTagPool**

  Enables or disables the reuse of custom tag handlers on an individual JSP page basis.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -usePageTagPool true

  Use-case: Use this parameter to enable JSP-page-based reuse of tag handlers.

  Default: false

- **useThreadTagPool**

  Enables or disables the reuse of custom tag handlers on a per request thread basis per Web module.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -useThreadTagPool true

  Use-case: Use this parameter to enable Web module-based reuse of tag handlers.

  Default: false

- **classloader.parentFirst**

  Specifies the search order for loading classes by instructing the batch compiler to search the parent class loader prior to application class loader. This parameter is only used when *ear.path* or *enterpriseApp.name* is given.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -classloader.parentFirst false

  Use-case: Set this parameter to false when your Web module contains a JAR file that is also found in the server lib directory, and you want your Web module's JAR file to be picked up first.

  Default: true

- **classloader.singleWarClassloader**

  Specifies whether to use one class loader per enterprise archive (EAR) file or one class loader per Web application archive (WAR) file. Used only when *ear.path* or *enterpriseApp.name* is given.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -classloader.singleWarClassloader true

  Use-case: Set this parameter to true when a Web module depends on JAR files and classes in another Web module in the same enterprise application.

  Default: false; One class loader is created per WAR file with no visibility of classes in other Web modules.

- **additional.classpath**

  Specifies additional class path entries to be used when parsing and compiling JSP pages. This parameter is used only when `war.path` is given. When `war.path` is the target, WebSphere Shared Libraries are not picked up by the batch compiler. Therefore, if your WAR file relies on, for example, a JAR file that is configured in WebSphere Application Server as a shared library, then use this option to point to that JAR file. In addition, if you give `war.path` and also use the `-extractToDir` parameter, then any JAR files that are in the WAR file's manifest class-path is not added to the class path (since the WAR file has now been extracted by itself outside the EAR file in which it resides). Use `-additional.classpath` in this case to point to the necessary JAR files. Add the full path to needed resources, separated by your system-dependent path separator.

Example: JspBatchCompiler -war.path c:\myproject\examples.war -additional.classpath c:\myJars\someJar.jar;c:\myClasses

Use-case: Use this parameter to add to the class path JAR files and classes outside of your WAR file. At run time, these same JAR files and classes have to be made available through the standard WebSphere Application Server configuration mechanisms.

Default: null

- **verbose**

  Specifies whether the batch compiler should generate verbose output while compiling the generated sources.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -verbose true

  Use-case: Set this parameter to true when you want to see Java compiler class loading and other messages.

  Default: false

- **deprecation**

  Indicates the compiler should generate deprecation warnings while compiling the generated sources.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -deprecation true

  Use-case: Set this parameter to true when you want to see Java compiler deprecation messages.

  Default: false

- **javaEncoding**

  Specifies the encoding that will be used when the .java file is generated, and when it is compiled by the Java compiler. When -javaEncoding is set, that encoding is passed to the java compiler via the -encoding argument. Note that encoding is not supported by Jikes.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -javaEncoding Shift-JIS

  Use-case: Set this parameter when the page encoding of your JSP pages is not UTF-8 compatible.

  Default value: UTF-8.

- **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.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -jdkSourceLevel 14

  Use-case: Set this parameter when you want to enable or disable the language features of JDK 1.4 and JDK 5.0

  Default value: 13

- **compilerOptions**

  Specifies a list of strings to be passed on the Java compiler command. This is a space-separated list of the form ″arg1 arg2 argn″.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -compilerOptions ″ -bootclasspath <path>″

  Use-case: Use this parameter if you need Java compiler arguments other than verbose, deprecation and Assert facility support.

  Default: null

- **useJikes**

Specifies whether Jikes should be used for compiling Java sources. NOTE: Jikes is not shipped with WebSphere Application Server.

Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -useJikes true

Use-case: Set this parameter to true in order for the batch compiler to use Jikes as the Java compiler.

Default value: false

- **jsp.file.extensions**

    Specifies the file extensions to be processed by the batch compiler. This is a semicolon- or colon-separated list of the form ″*.ext1;*.ext2:*.extn″. Note that this parameter is not necessary for Servlet 2.4 Web applications because the url-pattern property of the jsp-property-group elements in the deployment descriptor can be used to identify extensions that should be treated as JSP pages.

    Example: JspBatchCompiler -enterpriseApp.name sampleApp -jsp.file.extensions *jspz;*.jspt

    Use-case: Use this parameter to add additional extensions to the be processed by the batch compiler.

    Default: null. See section, ″JSP file extensions″, in this topic for additional information.

- **log.level**

    Specifies the level of logging that is directed to the console during batch compilation. Values are SEVERE | WARNING | INFO | CONFIG | FINE | FINER | FINEST | OFF

    Example: JspBatchCompiler -enterpriseApp.name sampleApp -log.level FINEST

    Use-case: Set this parameter higher or lower to control logging output. FINEST generates the most output useful for debugging.

    Default: CONFIG

Batch compiler ant task:

The ant task **JspC** exposes all the batch compiler configuration options. It executes the batch compiler under the covers. It is backward compatible with the WebSphere Application Server 5.x version of the **JspC** ant task. The following table lists all the ant task attribute and their batch compiler equivalents.

JspC attribute	Equivalent batch complier parameter
earPath	-ear.path
warPath	-war.path
src Same as warPath, for backward compatiblity	-war.path
enterpriseAppName	-enterpriseapp.name
responseFile	-response.file
webmoduleName	-webmodule.name
fileName	-filename -config.root
configRoot	-config.root
cellName	-cell.name
nodeName	-node.name
serverName	-server.name
profileName	-profileName
extractToDir	-extractToDir
compileToDir same as compileToDir, for backward compatibility	-compileToDir -compileToDir
compileToWebInf	-compileToWebInf
compilerOptions	-compilerOptions

recurse	-recurse
removeTempDir	-removeTempDir
translate	-translate
compile	-compile
forceCompilation	-forceCompilation
useFullPackageNames	-useFullPackageNames
trackDependencies	-trackDependencies
createDebugClassfiles	-createDebugClassfiles
keepgenerated	-keepgenerated
keepGeneratedclassfiles	-keepGeneratedclassfiles
usePageTagPool	-usePageTagPool
useThreadTagPool	-useThreadTagPool
classloaderParentFirst	-classloader.parentFirst
classloaderSingleWarClassloader	-classloader.singleWarClassloader
additionalClasspath	-additional.classpath
classpath same as additionalClasspath, for backward compatibility	-additional.classpath
verbose	-verbose
deprecation	-deprecation
javaEncoding	-javaEncoding
compileWithAssert	-compileWithAssert
useJikes	-useJikes
jspFileExtensions	-jsp.file.extensions
logLevel	-log.level
wasHome	none
Classpathref	none
jdkSourceLevel	-jdkSourceLevel

Below is an example of a build script with multiple targets, each with different attributes. The following commands are used to execute the script:

On Windows:

```
ws_ant -Dwas.home=%WAS_HOME% -Dear.path=%EAR_PATH% -Dextract.dir=%EXTRACT_DIR%
ws_ant jspc2 -Dwas.home=%WAS_HOME% -Dapp.name=%APP_NAME% -Dwebmodule.name=%MOD_NAME%
ws_ant jspc3 -Dwas.home=%WAS_HOME% -Dapp.name=%APP_NAME% -Dwebmodule.name=%MOD_NAME% -Ddir.name=%DIR_NAME%
```

On UNIX or i5/OS:

```
ws_ant -Dwas.home=$WAS_HOME -Dear.path=$EAR_PATH -Dextract.dir=$EXTRACT_DIR
ws_ant jspc2 -Dwas.home=$WAS_HOME -Dapp.name=$APP_NAME -Dwebmodule.name=$MOD_NAME
ws_ant jspc3 -Dwas.home=$WAS_HOME -Dapp.name=$APP_NAME -Dwebmodule.name=$MOD_NAME -Ddir.name=$DIR_NAME
```

Example build.xml Using the JspC Task

```
<project name="JSP Precompile" default="jspc1" basedir=".">
 <taskdef name="wsjspc" classname="com.ibm.websphere.ant.tasks.JspC"/>
 <target name="jspc1" description="example using  a path to an EAR, and extracting the EAR to a directory">
  <wsjspc wasHome="${was.home}"
          earpath="${ear.path}"
```

```
                forcecompilation="true"
                extractToDir="${extract.dir}"
                useThreadTagPool="true"
                keepgenerated="true"

 />
 </target>
 <target name="jspc2" description="example using an enterprise app and webmodule">
  <wsjspc wasHome="${was.home}"
                enterpriseAppName="${app.name}"
                webmoduleName="${webmodule.name}"
                removeTempDir="true"
                forcecompilation="true"
                keepgenerated="true"

 />
 </target>
 <target name="jspc3" description="example using an enterprise app, webmodule and specific directory">
  <wsjspc wasHome="${was.home}"
                enterpriseAppName="${app.name}"
                webmoduleName="${webmodule.name}"
                fileName="${dir.name}"
                recurse="false"
                forcecompilation="true"
                keepgenerated="true"

 />
 </target>
</project>
```

Batch compiler class path:

The batch compiler builds its class path as shown in the table below. When the batch compiler target is a Web archive (WAR) file and `war.path` is supplied, the configuration *additional.classpath* parameter is used to give extra class path information.

	Batch compiler target		
Location added to class path	enterpriseapp.name	ear.path	war.path
WebSphere Application Server JAR files and classes	yes	yes	yes
JAR files listed in manifest class path for a Web module	yes	yes	yes, when the target WAR is inside an EAR and *–extractToDir* is not used; otherwise, no.
Shared libraries	yes	no	no
Web module JAR files and classes	yes	yes	yes
additional.classpath parameter to batch compiler	no	no	yes

Global tag libraries:

JavaServer Pages (JSP) tag libraries contain classes for common tasks such as processing forms and accessing databases from JSP files.

Tag libraries encapsulate, as simple tags, core functionality common to many Web applications. The Java Standard Tag Library (JSTL) supports common programming tasks such as iteration and conditional processing, and provides tags for:
• manipulating XML documents
• supporting internationalization
• using Structured Query Language (SQL)

Tag libraries also introduce the concept of an expression language to simplify page development, and include a version of the JSP expression language.

A tag library has two parts - a Tag Library Descriptor (TLD) file and a Java archive (JAR) file.

tsx:dbconnect tag JavaServer Pages syntax (deprecated):

Use the <tsx:dbconnect> tag to specify information needed to make a connection to a database through Java DataBase Connectivity (JDBC) or Open Database Connectivity (ODBC) technology.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The <tsx:dbconnect> syntax does not establish the connection. Use the <tsx:dbquery> and <tsx:dbmodify> syntax instead to reference a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file to establish the connection.

When the JSP file compiles into a servlet, the Java processor adds the Java coding for the <tsx:dbconnect> syntax to the servlet service() method, which means a new database connection is created for each request for the JSP file.

This section describes the syntax of the <tsx:dbconnect> tag.

```
<tsx:dbconnect id="connection_id"
    userid="db_user" passwd="user_password"
    url="jdbc:subprotocol:database"
    driver="database_driver_name"
    jndiname="JNDI_context/logical_name">
</tsx:dbconnect>
```

where:
* **id**

  Represents a required identifier. The scope is the JSP file. This identifier is referenced by the connection attribute of a <tsx:dbquery> tag.
* **userid**

  Represents an optional attribute that specifies a valid user ID for the database that you want to access. Specify this attribute to add the attribute and its value to the request object.

  Although the userid attribute is optional, you must provide the user ID. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this information in the JSP file.
* **passwd**

  Represents an optional attribute that specifies the user password for the userid attribute. (This attribute is not optional if the userid attribute is specified.) If you specify this attribute, the attribute and its value are added to the request object.

  Although the passwd attribute is optional, you must provide the password. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this attribute in the JSP file.
* **url** and **driver**

  Respresents a required attribute if you want to establish a database connection. You must provide the URL and driver.

  The application server supports connection to JDBC databases and ODBC databases.
  – For a JDBC database, the URL consists of the following colon-separated elements: jdbc, the subprotocol name, and the name of the database to access. An example for a connection to the Sample database included with IBM DB2 is:

    ```
    url="jdbc:db2:sample"
    driver="com.ibm.db2.jdbc.app.DB2Driver"
    ```

- For an ODBC database, use the Sun JDBC-to-ODBC bridge driver included in their Java2 Software Developers Kit (SDK) or another vendor's ODBC driver.

  The url attribute specifies the location of the database. The driver attribute specifies the name of the driver to use in establishing the database connection.

  If the database is an ODBC database, you can use an ODBC driver or the Sun JDBC-to-ODBC bridge. If you want to use an ODBC driver, refer to the driver documentation for instructions on specifying the database location with the url attribute and the driver name.

  If you use the bridge, the url syntax is `jdbc:odbc:database`. An example follows:

  ```
  url="jdbc:odbc:autos"
  driver="sun.jdbc.odbc.JdbcOdbcDriver"
  ```

  Note: To enable the application server to access the ODBC database, use the ODBC Data Source Administrator to add the ODBC data source to the System DSN configuration. To access the ODBC Administrator, click the ODBC icon on the Windows NT Control Panel.

- **jndiname**

  Represents an optional attribute that identifies a valid context in the application server Java Naming and Directory Interface (JNDI) naming context and the logical name of the data source in that context. The Web administrator configures the context using an administrative client such as the WebSphere Administrative Console.

  If you specify the jndiname attribute, the JSP processor ignores the driver and url attributes on the <tsx:dbconnect> tag.

An empty element (such as <url></url>) is valid.

dbquery tag JavaServer Pages syntax (deprecated):

Use the <tsx:dbquery> tag to establish a connection to a database, submit database queries, and return the results set.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The <tsx:dbquery> tag does the following:
1. References a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file and uses the information the tag provides to determine the database URL and driver. You can also obtain the user ID and password from the <tsx:dbconnect> tag if those values are provided in the <tsx:dbconnect> tag.
2. Establishes a new connection
3. Retrieves and caches data in the results object.
4. Closes the connection and releases the connection resource.

This section describes the syntax of the <tsx:dbquery> tag.

```
<%-- SELECT commands and (optional) JSP syntax can be placed within the tsx:dbquery. --%>
<%-- Any other syntax, including HTML comments, are not valid. --%>
<tsx:dbquery id="query_id" connection="connection_id" limit="value" >
</tsx:dbquery>
```

where:
- **id**

  Represents the identifier of this query. The scope is the JSP file. Use `id` to reference the query. For example, from the <tsx:getProperty> tag, use `id` to display the query results.

  The `id` is a tsx reference to the bean and can be used to retrieve the bean from the page context. For example, if id is named mySingleDBBean, instead of using:
  - if (mySingleDBBean.getValue(″UISEAM″,0).startsWith(″N″))

  use:

– com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults bean =
(com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults)pageContext. findAttribute(″mySingleDBBean″); if
(bean.getValue(″UISEAM″,0).startsWith(″N″)). . .

The bean properties are dynamic and the property names are the names of the columns in the results
set. If you want different column names, use the SQL keyword for specifying an alias on the SELECT
command. In the following example, the database table contains columns named FNAME and LNAME,
but the SELECT statement uses the AS keyword to map those column names to FirstName and
LastName in the results set:

```
Select FNAME, LNAME AS FirstName, LastName from Employee where FNAME='Jim'
```

* **connection**

  Represents the identifier of a <tsx:dbconnect> tag in this JSP file. The <tsx:dbconnect> tag provides the
  database URL, driver name, and optionally, the user ID and password for the connection.
* **limit**

  Represents an optional attribute that constrains the maximum number of records returned by a query. If
  this attribute is not specified, no limit is used. In such a case, the effective limit is determined by the
  number of records and the system caching capability.
* SELECT command and JSP syntax

  Represents the only valid SQL command, SELECT. The <tsx:dbquery> tag must return a results set.
  Refer to your database documentation for information about the SELECT command. See other articles
  in this section for a description of JSP syntax for variable data and inline Java code.

dbmodify tag JavaServer Pages syntax (deprecated):

The <tsx:dbmodify> tag establishes a connection to a database and then adds records to a database
table.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application
Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer
Pages Standard Tag Library (JSTL).

The <tsx:dbmodify> tag does the following:
1. References a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file and uses the information
   provided by that tag to determine the database URL and driver.

   Note: You can also obtain the user ID and password from the <tsx:dbconnect> tag if those values are
   provided in the <tsx:dbconnect> tag.
2. Establishes a new connection.
3. Updates a table in the database.
4. Closes the connection and releases the connection resource.

This section describes the syntax of the <tsx:dbmodify> tag.

```
<%-- Any valid database update commands can be placed within the DBMODIFY tag. -->
<%-- Any other syntax, including HTML comments, are not valid. -->
<tsx:dbmodify connection="connection_id">
</tsx:dbmodify>
```

where:
* **connection**

  Represents the identifier of a <tsx:dbconnect> tag in this JSP file. The <tsx:dbconnect> tag provides the
  database URL, driver name, and (optionally) the user ID and password for the connection.
* Database commands

  Represents valid database commands. Refer to your database documentation for details

tsx:getProperty tag JavaServer Pages syntax and examples (deprecated):

The `<tsx:getProperty>` tag gets the value of a bean to display in a JavaServer Pages (JSP) file.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

This IBM extension of the Sun JSP <jsp:getProperty> tag implements all of the <jsp:getProperty> function and adds the ability to introspect a database bean created using the IBM extension <tsx:dbquery> or <tsx:dbmodify>.

Note: You cannot assign the value from this tag to a variable. The value, generated as output from this tag, displays in the browser window.

This section describes the syntax of the `<tsx:getProperty>` tag:

```
<tsx:getProperty name="bean_name"
  property="property_name" />
```

where:
- **name**

  Represents the name of the bean declared by the `id` attribute of a <tsx:dbquery> syntax within the JSP file. See <tsx:dbquery> for an explanation. The value of this attribute is case-sensitive.
- **property**

  Represents the property of the bean to access for substitution. The value of the attribute is case-sensitive and is the locale-independent name of the property.

Tag example:

```
<tsx:getProperty name="userProfile" property="username" />
```

tsx:userid and tsx:passwd tag JavaServer Pages syntax (deprecated):

With the <tsx:userid> and <tsx:passwd> tags, you do not have to hard code a user ID and password in the <tsx:dbconnect> tag.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

Use the <tsx:userid> and <tsx:passwd> tags to accept user input for the values and then add that data to the request object. You can access the request object with a JavaServer Pages (JSP) file, such as the *JSPEmployee.jsp* example that requests the database connection.

You must use <tsx:userid> and <tsx:passwd> tags within a <tsx:dbconnect> tag.

This section describes the syntax of the <tsx:userid> and <tsx:passwd> tags.

```
<tsx:dbconnect id="connection_id"
    <font color="red"><userid></font>
    <tsx:getProperty name="request" property=request.getParameter("userid") />
    <font color="red"></userid></font>
    <font color="red"><passwd></font>
    <tsx:getProperty name="request" property=request.getParameter("passwd") />
    <font color="red"></passwd></font>
    url="protocol:database_name:database_table"
    driver="JDBC_driver_name">
</tsx:dbconnect>
```

where:
- **<tsx:getProperty>**

Represents the syntax as a mechanism for embedding variable data.

- **userid**

  Represents a reference to the request parameter that contains the user ID. You must add the parameter to the request object that passes to this JSP file. You can set the attribute and its value in the request object, using an HTML form or a URL query string to pass the user-specified request parameters.

- **passwd**

  Represents a reference to the request parameter that contains the password. Add the parameter to the request object that passes to this JSP file. You can set the attribute and its value in the request object, using an HTML form or a URL query string, to pass user-specified values.

tsx:repeat tag JavaServer Pages syntax (deprecated):

The <tsx:getProperty> tag repeats a block of HTML tagging.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

Use the <tsx:repeat> syntax to iterate over a database query results set. The <tsx:repeat> syntax iterates from the start value to the end value until one of the following conditions is met:
- The end value is reached.
- An exception is thrown.

If an exception of the types **ArrayIndexOutOfBoundsException** or **NoSuchElementException** is created before a block completes, output is written only for the iterations up to and not including the iteration during which the exception was created. All other exceptions results in no output being written for that tag instance.

This section describes the syntax of the `<tsx:repeat>` tag:

```
<tsx:repeat index="name" start="starting_index" end="ending_index">
</tsx:repeat>
```

where:
- **index**

  Represents an optional name used to identify the index of this repeat block. The scope of the index is NESTED. Its type must be integer.

- **start**

  Represents an optional starting index value for this repeat block. The default is `0`.

- **end**

  Represents an optional ending index value for this repeat block. The maximum value is `2,147,483,647`.

  If the value of the end attribute is less than the value of the start attribute, the end attribute is ignored.

Example: Combining tsx:repeat and tsx:getProperty JavaServer Pages tags (deprecated): Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The following code snippet shows you how to code these tags:

```
<tsx:repeat>
<tr>
   <td><tsx:getProperty name="empqs" property="EMPNO" />
   <tsx:getProperty name="empqs" property="FIRSTNME" />
   <tsx:getProperty name="empqs" property="WORKDEPT" />
```

```
    <tsx:getProperty name="empqs" property="EDLEVEL" />
    </td>
</tr>
</tsx:repeat>
```

Example: tsx:dbmodify tag syntax (deprecated): Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JavaServer Pages (JSP) file and referenced in the database commands using the <tsx:getProperty> tag.

```
<tsx:dbmodify connection="conn" >
insert into EMPLOYEE
    (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)
values
('<tsx:getProperty name="request" property=request.getParameter("EMPNO") />',
'<tsx:getProperty name="request" property=request.getParameter("FIRSTNME") />',
'<tsx:getProperty name="request" property=request.getParameter("MIDINIT") />',
'<tsx:getProperty name="request" property=request.getParameter("LASTNAME") />',
'<tsx:getProperty name="request" property=request.getParameter("WORKDEPT") />',
<tsx:getProperty name="request" property=request.getParameter("EDLEVEL") />)
</tsx:dbmodify>
```

Example: Using tsx:repeat JavaServer Pages tag to iterate over a results set (deprecated): Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The <tsx:repeat> tag iterates over a results set. The results set is contained within a bean. The bean can be a static bean, for example, a bean created by using the IBM WebSphere Studio database wizard, or a dynamically generated bean, for example, a bean generated by the <tsx:dbquery> syntax. The following table is a graphic representation of the contents of a bean called, *myBean*:

	col1	col2	col3
row0	friends	Romans	countrymen
row1	bacon	lettuce	tomato
row2	May	June	July

Some observations about the bean:
- The column names in the database table become the property names of the bean. The <tsx:dbquery> section describes a technique for mapping the column names to different property names.
- The bean properties are indexed. For example, `myBean.get(Col1(row2))` returns `May`.
- The query results are in the rows. The <tsx:repeat> tag iterates over the rows, beginning at the start row.

The following table compares using the <tsx:repeat> tag to iterate over a static bean, versus a dynamically generated bean:

Static Bean Example	<tsx:repeat> Bean Example

myBean.class
 	
``
// Code to get a connection
 	
// Code to get the data
   Select * from myTable;
 	
// Code to close the connection
``
 	
JSP file
 	
``
<tsx:repeat index=abc>
  <tsx:getProperty name="myBean"
    property="col1(abc)" />
</tsx:repeat>
``
 	
Notes:
• The bean (myBean.class) is a static bean.
• The method to access the bean properties is myBean.get(*property*(*index*)).
• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.
• The <tsx:repeat> tag iterates over the bean properties row by row, beginning with the start row.
	JSP file
	
	``
	<tsx:dbconnect id="conn"
	userid="alice"passwd="test"
	url="jdbc:db2:sample"
	driver="COM.ibm.db2.jdbc.app.DB2Driver">
	</tsx:dbconnect >
	
	<tsx:dbquery id="dynamic"
	 connection="conn" >
	  Select * from myTable;
	</tsx:dbquery>
	
	<tsx:repeat index=abc>
	  <tsx:getProperty name="dynamic"
	    property="col1(abc)" />
	</tsx:repeat>
	``
	
	Notes:
	• The bean (dynamic) is generated by the <tsx:dbquery> tag and does not exist until the syntax executes.
	• The method to access the bean properties is dynamic.getValue(″*property*″, *index*).
	• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.
	• The <tsx:repeat> tag syntax iterates over the bean properties row by row, beginning with the start row.

Implicit and explicit indexing

Examples 1, 2, and 3 show how to use the <tsx:repeat> tag. The examples produce the same output if all indexed properties have 300 or fewer elements. If there are more than 300 elements, Examples 1 and 2 display all elements, while Example 3 shows only the first 300 elements.

Example 1 shows *implicit indexing* with the default start and default end index. The bean with the smallest number of indexed properties restricts the number of times the loop repeats.

```
<table>
<tsx:repeat>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="address" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="telephone" />
  </tr></td>
</tsx:repeat>
</table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table>
<tsx:repeat index=myIndex start=0 end=2147483647>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=city(myIndex) />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=address(myIndex) />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=telephone(myIndex) />
</tr></td>
</tsx:repeat>
</table>
```

Example 3 shows *explicit indexing* and ending index with implicit starting index. Although the index attribute is specified, you can still implicitly index the indexed property city because the (`myIndex`) tag is not required.

```
<table>
<tsx:repeat index=myIndex end=299>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" /t>
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="address(myIndex)" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="telephone(myIndex)" />
  </tr></td>
</tsx:repeat>
</table>
```

Nesting <tsx:repeat> blocks

You can nest <tsx:repeat> blocks. Each block is separately indexed. This capability is useful for interleaving properties on two beans, or properties that have subproperties. In the example, two <tsx:repeat> blocks are nested to display the list of songs on each compact disc in the user's shopping cart.

```
<tsx:repeat index=cdindex>
  <h1><tsx:getProperty name="shoppingCart" property=cds.title /></h1>
  <table>
  <tsx:repeat>
    <tr><td><tsx:getProperty name="shoppingCart" property=cds(cdindex).playlist />
    </td></tr>
  </tsx:repeat>
  </table>
 </tsx:repeat>
```

JavaServer Pages migration best practices and considerations:

The standard JavaServer Pages (JSP) tags from JSP 1.1 such as jsp:include, jsp:useBean, and <%@ page %>, a will migrate successfully to JSP 2.0. However, there are several areas that must be considered when migrating JavaServer Pages. This topic discusses the areas that you must consider when migrating JavaServer Pages.

Classes from the unnamed or default package

As of JSP 2.0, referring to any classes from the unnamed or default package is not allowed. This can result in a translation error on some containers, specifically those that run in a JDK 1.4 or greater environment which will also break compatibility with some older JSP applications. However, as of JDK 1.4, importing classes from the unnamed package is not valid. See Java 2 Platform, Standard Edition Version 1.4.2 Compatibility with Previous Releases for details. Therefore, for forwards compatibility, applications must not rely on the unnamed package. This restriction also applies for all other cases where classes are referenced, such as when specifying the class name for a tag in a Tag Library Descriptor (TLD) file.

Page encoding for JSP documents

There have been noticeable differences in internationalization behavior on some containers as a result of ambiguity in the JSP 1.2 specification. However, steps were taken to minimize the impact on backwards compatibility and overall, the internationalization abilities of JSP files have been greatly improved.

In JSP specification versions prior to JSP 2.0, JSP pages in XML syntax, JSP documents, and those in standard syntax determined their page encoding in the same fashion, by examining the pageEncoding or contentType attributes of their page directive, defaulting to ISO-8859-1 if neither was present.

As of JSP 2.0, the page encoding for JSP documents is determined as described in section 4.3.3 and appendix F.1 of the XML specification, and the pageEncoding attribute of those pages is only checked to

make sure it is consistent with the page encoding determined as per the XML specification. As a result of this change, JSP documents that rely on their page encoding to be determined from their pageEncoding attribute are no longer decoded correctly. These JSP documents must be changed to include an appropriate XML encoding declaration.

Additionally, in JSP 1.2, page encodings are determined on translation unit basis whereas in JSP 2.0, page encodings are determined on the basis of each file. Therefore, if the a.jsp file statically includes the b.jsp file, and a page encoding is specified in the a.jsp file but not in the b.jsp file, in JSP 1.2 the encoding for the a.jsp file is used for the b.jsp file, but in JSP 2.0, the default encoding is used for the b.jsp file.

web.xml file version

The JSP container uses the version of the web.xml file to determine whether you are running a JSP 1.2 application or a JSP 2.0 application. Various features can behave differently depending on the version of the web.xml file. The following is a list of things JSP developers should be aware of when upgrading their web.xml file from version Servlet 2.3 to version Servlet 2.4:

1. EL expressions are ignored by default in JSP 1.2 applications. When you upgrade a Web application to JSP 2.0, EL expressions are interpreted by default. You can use the escape sequence \$ to escape EL expressions that should not be interpreted by the container. Alternatively, you can use the isELIgnored page directive attribute, or the <el-ignored> configuration element to deactivate EL for entire translation units. Users of JSTL 1.0 must upgrade their taglib imports to the JSTL 1.1 uris or use the _rt versions of the tags, for example, use c_rt instead of c or fmt_rt instead of fmt.

2. Web applications that contain files with an extension of .jspx will have those files interpreted as JSP documents, by default. You can use the JSP configuration element <is-xml> to treat .jspx files as regular JSP pages, but there is no way to disassociate .jspx from the JSP container.

3. The escape sequence \$ was not reserved in JSP 1.2. The output for any template text or attribute value that appeared as \$ in JSP 1.2 was \$, however, the output now is just $.

jsp:useBean tag

WebSphere Application Server version 5.1 and later enforces more strict adherence to the specification for the jsp:useBean tag: with type and class attributes. Specifically, you should use the type attribute should be used to specify a Java type that cannot be instantiated as a JavaBean. For example, a Java type that is an abstract class, interface, or a class with no public no-args constructor. If the class attribute is used for a Java type that cannot be instantiated as a JavaBean, the WebSphere Application Server JSP container produces a unrecoverable translation error at translation time.

Generated packages for JSP classes

Any reliance on generated packages for JSP classes will result in non-portable JSP files. Packages for generated classes are implementation-specific and therefore you should not rely on these packages.

JspServlet class

Any reliance on the existence of a JspServlet class will cause unrecoverable error problems. WebSphere Application Server version 6.0 and later no longer uses a JspServlet class.

Web modules

A Web module represents a Web application. A Web module is created by assembling servlets, JavaServer Pages (JSP) files, and static content such as Hypertext Markup Language (HTML) pages into a single deployable unit. Web modules are stored in Web archive (WAR) files, which are standard Java archive files.

A Web module contains:
- One or more servlets, JSP files, and HTML files.

- A deployment descriptor, stored in an Extensible Markup Language (XML) file.

  The file, named `web.xml`, declares the contents of the module. It contains information about the structure and external dependencies of Web components in the module and describes how the components are used at run time.

You can create Web modules as stand-alone applications, or you can combine Web modules with other modules to create Java 2 Platform, Enterprise Edition (J2EE) applications. You install and run a Web module in the Web container of an application server.

Troubleshooting tips for Web application deployment

Deployment of a Web application is successful if you can access the application by typing a Uniform Resource Locator (URL) in a browser, or if you can access the application by following a link.

If you cannot access your application, follow these steps to eliminate some common errors that can occur during migration or deployment.

Web module does not run in WebSphere Application Server Version 5.x or 6.x

Symptom	Your Web module does not run when you migrate it to Version 5.x or 6.x
Problem	In Version 4.x, the classpath setting that affected visibility was *Module Visibility Mode*. In Versions 5.x and 6.x, you must use class loader policies to set visibility.
Recommended response	Reassemble an existing module, or change the visibility settings in the class loader policies. See "Class loaders" on page 1351 and Chapter 8, "Class loading," on page 1351 for more information.

Welcome page is not visible.

Symptom	You cannot access an application with a Web path of: `/webapp/myapp`
Problem	The default welcome page for a Web application is assumed to be *index.html*. You cannot access the default page of the *myapp* application unless it is named *index.html*.
Recommended response	To identify a different welcome page, modify the properties of the Web module during assembly. See the article "Assembling Web applications" on page 81 for more information.

HTML files are not found.

Symptom	Your Web application ran successfully on prior versions, but now you encounter errors that the welcome page (typically *index.html*), or referenced HTML files are not found:
	
	`Error 404: File not found: Banner.html`
	`Error 404: File not found: HomeContent.html`

Problem	For security and consistency reasons, Web application URLs are now case-sensitive on all operating systems.
	Suppose the content of the index page is as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 5.0 Frameset//EN">
<HTML>
<TITLE>
Insurance Home Page
</TITLE>
    <frameset   rows="18,80">
    <frame      src="Banner.html"          name="BannerFrame"  SCROLLING=NO>
    <frame      src="HomeContent.html"      name="HomeContentFrame">
    </frameset>
</HTML>
```

However the actual file names in the \WebSphere\AppServer\installedApps\... directory where the application is deployed are:

```
banner.html
homecontent.html
```

Recommended response	To correct this problem, modify the *index.html* file to change the names *Banner.html* and *HomeContent.html* to *banner.html* and *homecontent.html* to match the names of the files in the deployed application.

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.

Web applications: Resources for learning

Use the following links to find relevant supplemental information about Web applications. 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
* J2EE BluePrints for Web applications
* Redbook on the design and implementation of Servlets, JSP files, and enterprise beans

Programming instructions and examples
* WebSphere Studio Application Developer Programming Guide
* Sun's Java™ Tutorial on Servlets and JavaServer Pages
* Web delivered samples in the Samples Gallery

Programming specifications
* Java 2 Software Development Kit (SDK)
* Servlet 2.4 Specification
* JavaServer Pages 2.0 Specification
* Differences between JavaScript and ECMAScript
* ISO 8859 Specifications
* Java 2 Platform, Standard Edition (J2SE)

Developing servlets with WebSphere Application Server extensions

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.

   See Resources for learning for links to coding specifications and examples.

   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 a default installation.

   All the public WebSphere Application Server APIs are located in the `com.ibm.websphere...` packages.

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.

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

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 precedes 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: 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)
    {
    }
}
```

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.

transition: 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.

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 > 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.

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

Example: com.ibm.websphere.LoggingFilter.java

The following example shows how to implement a filter:

```
package com.ibm.websphere;

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

public class LoggingFilter implements Filter
{
    File _loggingFile = null;
```

```
    // implement the required init method
    public void init(FilterConfig fc)
    {
        // create the logging file
        xxx;
    }

    // implement the required doFilter method...this is where most of
   the work is done
    public void doFilter(ServletRequest request,
     ServletResponse response, FilterChain chain)
    {
        try
        {
            // add request info to the log file
            synchronized(_loggingFile)
            {
                xxx;
            }

            // pass the request on to the next resource in the chain
            chain.doFilter(request, response);
        }
        catch (Throwable t)
        {
            // handle problem...
        }
    }

    // implement the required destroy method
    public void destroy()
    {
        // make sure logging file is closed
        _loggingFile.close();
    }
}
```

Configuring page list servlet client configurations

Note: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 6.1 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.

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.

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.

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 6.1 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 6.1 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 6.1 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 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>
```

Example: Extending PageListServlet:

Note: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 6.1 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 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);
        }
    }
}
```

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.

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

Review the `autoRequestEncoding` and `autoResponseEncoding` encoding examples for a description of Web container behavior when these values are set to `true`.

Examples: autoRequestEncoding and autoResponseEncoding encoding examples

The default value of the `autoRequestEncoding` and `autoResponseEncoding` extensions is `false`, which means that both the request and response character encoding is set to the Servlet 2.3 Specification default of ISO-8859-1. 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. Also, If the value is set to `false` for a response, the Web container cannot set a response content type.

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.

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.

Developing Web applications

Design a Web application and the components that it needs.

For general Web application design information, see ″Resources for learning.″

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. The IBM WebSphere Application Developer product is the recommended IDE. For more information, see the documentation for that product.
- 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 Software Development Kit (SDK) 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`

   ▶ Windows   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.

Assemble the application components in one or more Web modules.

JavaServer Faces
JavaServer Faces (JSF) is a user interface framework or API that eases the development of Java based Web applications. WebSphere Application Server version 6.1 supports JavaServer Faces 1.1 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
- Allows custom user interface components to be easily build and reused

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.
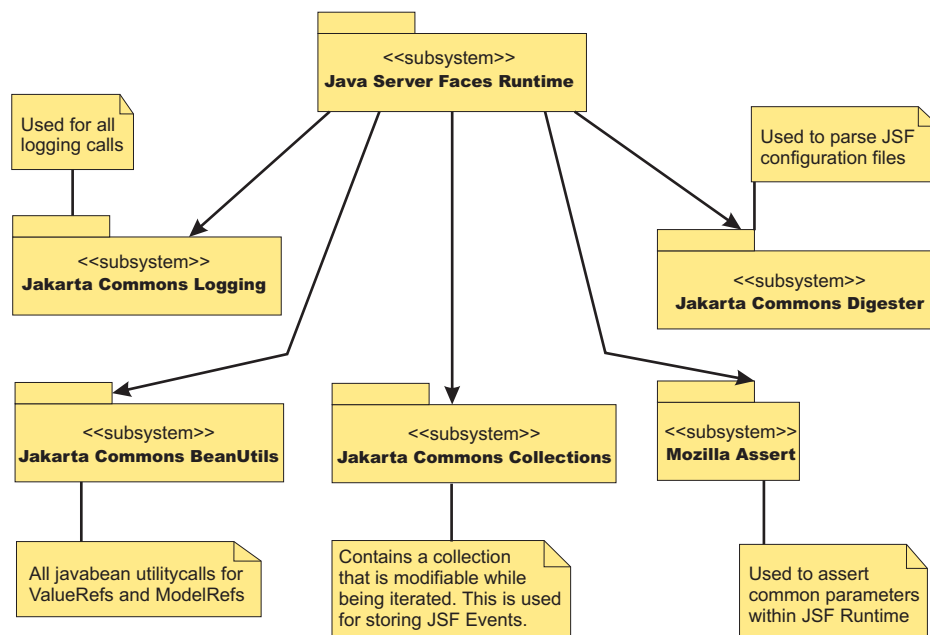
Figure 3. Current external API dependencies from the Sun based JSF runtime

Figure 4. Replacement APIs
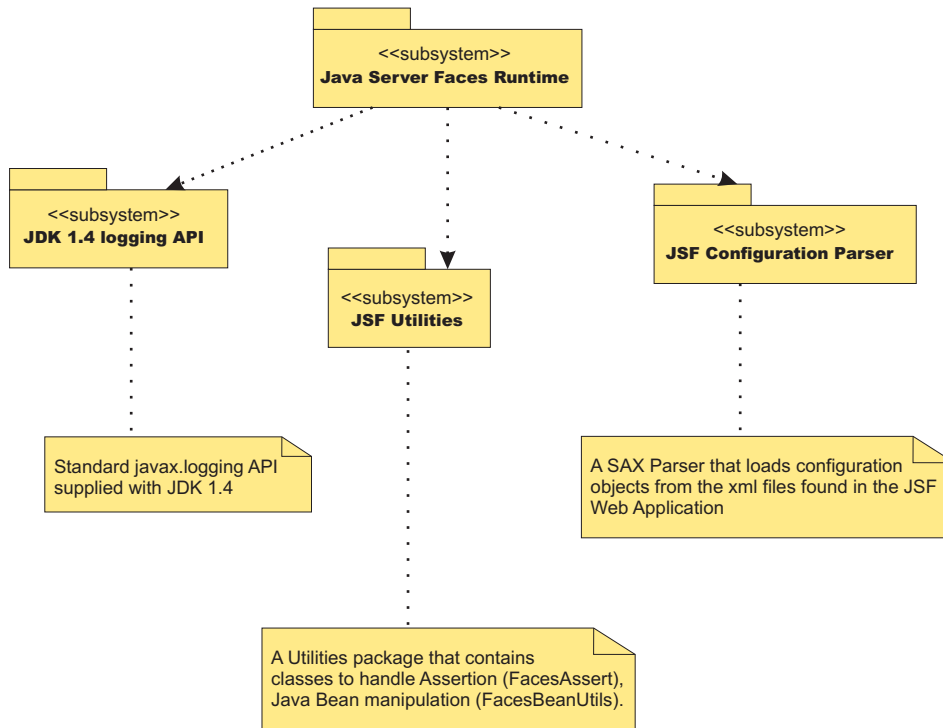
The specification related classes (javax.faces.*) for JSF and the IBM modified version of the JSF Sun reference implementation are packaged in the Application Server runtime.

Typically Web applications that leverage this API/Framework embed the JSF API and implementation JAR files within their WAR file. This is not required when these Web applications are deployed and run within WebSphere Application Server. Only the removal of these jars along with any JSTL JAR files from the WAR file is required.

If a Web application requires the use of its own version of JSF or JSTL embedded within it, you can change the class loader mode of the Web application. By default this is set to PARENT_FIRST mode. Changing this value to PARENT_LAST allows the Web application version of the JSF or JSTL classes to load before the WebSphere Application Server.

FacesAssert class

The Sun Reference implementation uses a utility class from Mozilla to perform assertion style calls to method parameters. The faces assert class provides equivalent functionality. The option of leveraging the assertion functionality available in JDK 1.4 is not possible due to the requirement of providing JVM level parameters to turn on assertion code support. The FacesAssert class only contains static method and has no life cycle.

FacesAssert
+ notEmpty ([in] str : String) : boolean
+ nonNull ([in] isNull : Object) : boolean
+ wsAssert ([in] message : String) : boolean
+ wsAssert ([in] argument : boolean , [in] message : String) : boolean

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

Faces configuration parser

The Sun Reference Implementation of JavaServer Faces use the Jakarta Commons Digester API to parse Faces configuration files. An XML SAX based parser is provided for the Application Server . The Digester code uses reflection code to perform its parsing. This has been found to be quite slow when large configuration files are parsed. The FaceConfigParser class in the diagram below is custom written for the Faces Configuration DTD and therefore parses large configuration files more quickly.

Figure 5. Faces configuration parser

JavaServer Faces widget library (JWL)

FacesClient framework

JavaServer Faces widget library (JWL) is a IBM JSF-based Web widget library that integrates widgets from a number of sources. It includes the JSF components from Rational Web Developer (RWD), with the exception of the base JSF components which are already included in the Application Server runtime. This includes the IBM extended JSF components and the extended Odyssey components.

JWL also extends JSF with client-side features for rich Browser-based experiences in the form of the Odyssey Browser Framework (OBF).

JWL Java archive files

JWL is packaged into two jar files, odc-jsf.jar and jsf-ibm.jar files, 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 article, Using installed optional packages.

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.

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.

Use the Application Server Toolkit (AST) or Rational Application Developer 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 J2EE specification level.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE 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.

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**.

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.

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.

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.

   For more information, see ″Importing Web archive (WAR) files″ and ″Importing an enterprise application EAR file″ in the Application Server Toolkit documentation.

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.

   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 or Web client and place the client certificate in the server trust keyring file, if `ClientCert` is selected.

9. Close the deployment descriptor editor and, when prompted, click **Yes** to save the changes.

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.

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.

Context parameters

A servlet context defines a server's view of the Web application within which the servlet is running. The context also allows a servlet to access resources available to it.

Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use. These properties declare a Web application's parameters for its context. They convey setup information, such as a webmaster's e-mail address or the name of a system that holds critical data.

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.
- 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 enabling the **Enable application security** option on the Secure administration, applications, and infrastructure panel.

Note that:

- Global settings apply to existing and future applications and cannot be customized.
- Default settings apply only to future applications and can be customized.

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 2 Platform, Enterprise Edition (J2EE) 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.

Servlet mappings

A servlet mapping is a correspondence between a client request and a servlet.

Web containers use URL paths to map client requests to servlets, and follow the URL path-mapping rules as specified in the Java Servlet specification. The container uses the URI from the request, minus the context path, as the path to map to a servlet. The container chooses the longest matching available context path from the list of Web applications that it hosts.

Serving of servlets by name or class name

This behavior is triggered by setting the serveServletsbyClassnameEnabled property within IBM extensions.

The attribute is used to specify the enablement of the serving of servlets by name or classname

Example attributes:

invoker.patterns
> This attribute allows you to specify the patterns that trigger invocation of the server component and allows the serving of servlets by name or by class name. This value is a list separated by either a space, colon, or semicolon.

File serving

File serving allows a Web application to serve static file types, such as HTML. File-serving attributes are used by the servlet that implements file-serving behavior.

This 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

Initialization parameters

Initialization parameters are sent to a servlet in its HttpConfig object when the servlet is first started.

Servlet caching

You can use dynamic caching to improve the performance of a servlet and JavaServer Pages (JSP) files by serving requests from an in-memory cache. Cache entries contain the servlet's output and metadata.

Web components

A Web component is a servlet, JavaServer Pages (JSP) file, or HTML file. One or more Web components make up a Web module.

Web property extensions

Web property extensions are IBM extensions to the standard deployment descriptors for Web applications. These extensions include mime filtering and servlet caching.

Web resource collections

A Web resource collection defines a set of URL patterns (resources) and HTTP methods belonging to the resource.

HTTP methods handle HTTP-based requests, such as GET, POST, PUT, and DELETE. A URL pattern is a partial Uniform Resource Locator that acts as a template for matching the pattern with existing full URLs in an attempt to find a valid file.

Welcome files

A Welcome file is an entry point file (for example, `index.html`) for a group of related HTML files.

Welcome files are located by using a group of partial URIs. The Web container uses the partial URIs to find a valid file when the initial URI is not found.

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.

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 92 for details.

Some statement about the process used to arrive here.

1. "Installing application files with the console" on page 1368

2. Configure the sending of include requests between the application and remote resources.
    *   "Configuring Web applications to dispatch remote includes" on page 89
    *   "Configuring Web applications to service remote includes" on page 89

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.

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.

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 92 for details.

You can also configure this property when you install the application installation. See "Installing application files with the console" on page 1368

1. In the administrative console, click **Applications > Enterprise Applications >** *application_name* **> Remote Request Dispatcher Properties**.
2. Select **Enable remote resources to receive include requests**.

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

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.

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 92 for details.

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 application files with the console" on page 1368

1. In the administrative console, click **Applications > Enterprise Applications >** *application_name* **> Remote Request Dispatcher Properties**.
2. Select **Enable remote resources to service include requests**.

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

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 89.

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.

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.

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 request dispatch 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 > Enterprise Applications > **application_name** > Remote request dispatch 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

Example: Locating resources 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 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>
```

Example: Locating resources 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);
}
```

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" on page 94
- "Extension handlers" on page 95
- "Extension delegators" on page 95
- "Custom EMF packages" on page 96

For more information on the com.ibm.wsspi.rrd.extension package, see "Reference: Generated API documentation" on page 26 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 "Reference: Generated API documentation" on page 26 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 "Reference: Generated API documentation" on page 26 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.

Defining an extension for the registry filter

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

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

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 become extensible as soon as they define an 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 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?

  The Application Server Toolkit 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>
```

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.

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.

To change the size of the invocation cache:

1. In the administrative console, click **Servers > Application servers** and select the application server you are tuning.
2. Click **Process Definition** under Additional Properties.
3. Click **Java Virtual Machine** under Additional Properties.
4. Click **Custom Properties** under Additional Properties.
5. 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.
6. Click **Apply** and then **Save** to save your changes.
7. Stop and restart the application server.

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

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 below.

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 2.4 API.

1. Plan your approach to session management, which could include session tracking and session recovery.
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.

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.

When you migrate between releases of WebSphere Application Server Version 5.x and later and you are using a database for session persistence, you can share the session database table between releases. For example, if you are accessing applications that are on WebSphere Application Server version 5.x you can share the session id with applications running on Version 6.x.

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 6.1 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*.
* In another WebSphere Application Server instance. This storage option is known as *memory-to-memory 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

There are several options for session tracking, depending on what sort of tracking method you want to use:
*   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: 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

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.

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 being activated in another.

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. Use session recovery support under the following conditions:
- When the user's session data must be maintained across a server restart
- 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, click **Servers** > **Application servers** > *server_name* > **Web container settings** > **Session management** > **Distributed environment settings**.

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

WebSphere Application Server supports session replication to another WebSphere Application Server instance. This support is referred to as *memory-to-memory session replication*. In this mode, sessions can replicate to one or more WebSphere Application Server instances 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.

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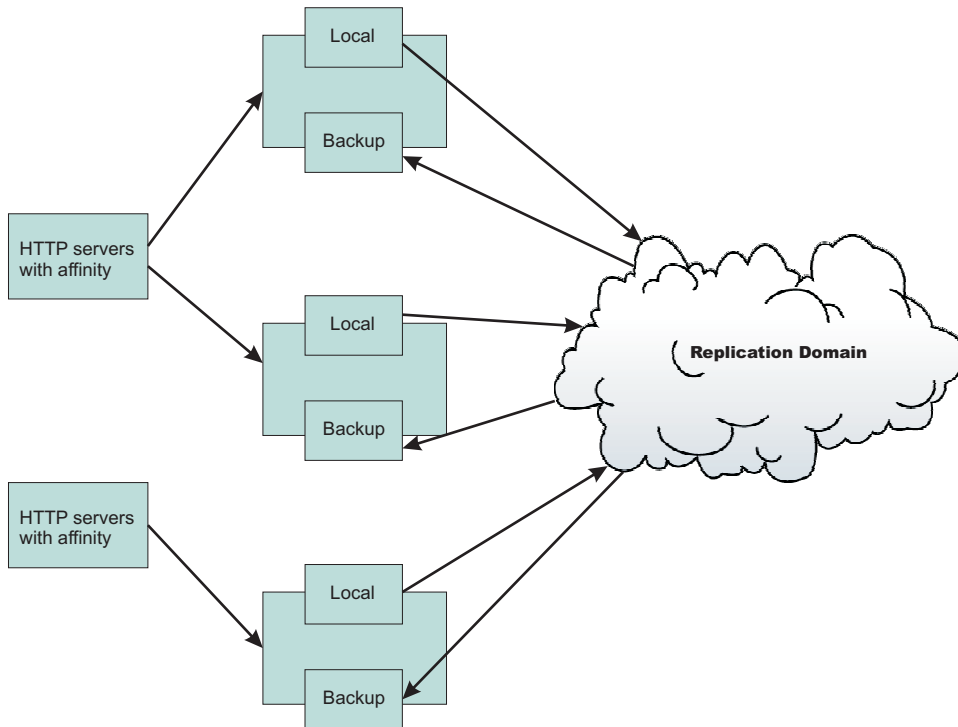
In this basic peer-to-peer topology, each server Java Virtual Machine (JVM) can:

- 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.

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.

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 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.

WebSphere Application Server
servers including HTTP sessions
with local tables.

WebSphere Application Server
servers including HTTP sessions
with backup tables.

Local

Local

Backup

HTTP servers
with affinity

Replication Domain

HTTP servers
with affinity
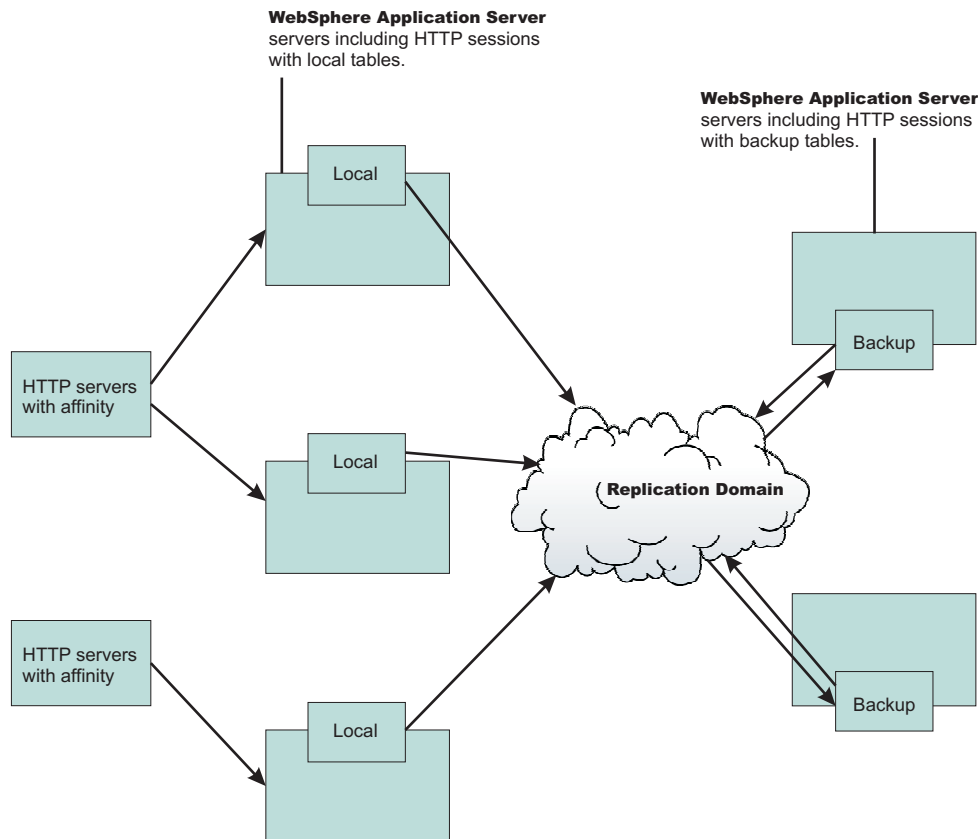
Local

Backup

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 table summarizes the features, including whether they apply to sessions tracked in memory, in a database, with memory-to-memory replication, or all. Click a feature for details about the feature. 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)

Base in-memory session pool size: The base in-memory session pool size number has different meanings, depending on session support configuration:
• With in-memory sessions, session access is optimized for up to this number of sessions.
• With distributed sessions (meaning, when sessions are stored in a database or in another WebSphere Application Server instance); it 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.

Tuning parameter settings:

Use this page to set tuning parameters for distributed sessions.

To view this administrative console page, click **Servers > Application servers >** *server_name* **> Web container settings > 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** > **Application servers** > *server_name* **Web container settings** > **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

best-practices: 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:
- See HTTP session aren't getting created or are getting dropped to see if your specific problem is discussed.
- View the JVM 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 Best practices for using HTTP Sessions.
- To dynamically view the number of sessions as a Web application is running, enable performance monitoring for HTTP sessions. This will give you an indication as to whether sessions are actually being created.
- To learn how to view the http session counters as the application runs, see Monitoring performance with Tivoli Performance Viewer (formerly Resource Analyzer).
- Alternatively, a special servlet can be invoked that displays the current configuration and statistics related to session tracking. This servlet has all the counters that are in performance monitor tool and has some additional counters.
  - 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.
- Enable tracing for the HTTP Session Manager component:
  - Use the trace specification **com.ibm.ws.webcontainer.httpsession.*=all=enabled**. Follow the instructions for dumping and browsing the trace output to narrow the origin of the problem.
  - If you are using persistent sessions based on memory replication, also enable trace for **com.ibm.ws.drs.***.
- 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 Errors accessing a datasource or connection pool

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.

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.

Problems creating or using HTTP sessions

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.

If your problem is not described here, or none of these steps fixes the problem:
- Review "HTTP session manager troubleshooting tips" on page 117 for general steps on debugging session-manager related problems.
- Review "Task overview: Managing HTTP sessions" on page 101 for information on how to configure the session manager, and best practices for using it.
- 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.

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 will ″expire″ 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 will be unpredictable, and will depend 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. On the server, enable SessionManager trace. Enable tracing for the HTTP session manager component, by using the trace specification ″com.ibm.ws.webcontainer.httpsession.*=all=enabled″. After trace is enabled, exercise your session-using servlet or jsp, then follow the instructions for dumping and browsing the trace output .
  3. Access the session servlet from the browser.
  4. The browser will prompt for the cookie; note the jsessionid.
  5. Reload the servlet, note down the cookie if a new cookie is sent.
  6. 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 **releaseSesson(..)** 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.
  - Ensure that your servlets and JSP files are implementing URL rewriting correctly. For details and an example see "Session tracking options" on page 104.
- 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.
  - Review "Session tracking options" on page 104.
- 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**.
  - Persistence could also be set to **Memory-to-Memory Replication**.
  - 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 JVM 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 "Memory-to-memory replication" on page 106.
  - 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" %>
```

Session data intended for one client is seen by another client

In rare situations, usually due to application errors, session data intended for one client might be seen by another client. This situation is referred to as session data crossover. When the *DebugSessionCrossover* custom property is set to true, code is enabled to detect and log instances of session data crossover. Checks are performed to verify that only the session associated with the request is accessed or referenced. Messages are logged if any discrepancies are detected. These messages provide a starting point for debugging this problem. This additional checking is only performed when running on the WebSphere-managed dispatch thread, not on any user-created threads.

For additional information on how to set this property, see article, Web container custom properties.

A ClassCastException error occurs during failover of a session that contains an Enterprise JavaBeans (EJB) reference

If you run WebSphere® Application Server for z/OS® Version 6.0.1 and configure a session manager to replicate EJB references, a session failover might trigger display of the following exception in the server region job log:

```
java.lang.ClassCastException: cannot cast class
        org.omg.stub.java.rmi._Remote_Stub to interface javax.ejb.EJBObject
```

The log also displays a null pointer exception. The problem results from the session outbound request, where WebSphere Application Server for z/OS issued a CORBA::COMM_FAILURE exception with a C9C21355 minor code. This behavior occurs because your application server contains all of the following configurations:

1. SAF is both the local operating system, as well as the user registry

2. Attribute propagation is enabled

3. An unauthenticated user initiated the session outbound request

To correct this problem apply the APAR PK06777 fix to WebSphere Application Server for z/OS V6.0.1. You can retain the previously mentioned server configurations.

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/was/support/

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
- Persistent client state HTTP cookies specification

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

Developing session management in servlets

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.

   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: SessionSample.java

```
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

In accordance with the Servlet 2.3 API specification, by default the Session Management facility supports session scoping by Web module. Only servlets in the same Web module can access the data associated with a particular session. WebSphere Application Server provides an option that you can use to extend the scope of the session attributes to an enterprise application. Therefore, you can share session attributes across all the Web modules in an enterprise application. This option is provided as an IBM extension.

Restriction: To use this option, you must install all the Web modules in the enterprise application on a given server. You cannot split up Web modules in the enterprise application by servers. For example, with an enterprise application containing two Web modules, you cannot use this option when one Web module is installed on one server and second Web module 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 option 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.

Servlet API Behavior

Note: If shared HttpSession context is turned on in an enterprise application, HttpSession listeners defined in all the Web modules inside the enterprise application are invoked for session events. The order of listener invocation is not guaranteed.

Do the following to share session data across Web modules in an enterprise application:

1. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
2. In the assembly tool, right-click the application (EAR file) you want to share and click **Open With > Deployment Descriptor Editor**.
3. 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.
4. 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.

Portlet applications

Task overview: Managing portlets

You can use this task to manage deployed portlet applications.

Before you begin this task, you must have a portlet application installed. See "Installing application files" on page 1364 for additional information.

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 131.
  - Access multiple portlets using "Portlet aggregation using JavaServer Pages" on page 125.
  - Access portlets using "Remote request dispatcher" on page 88.
- Change the location of "Portlet preferences" on page 132. 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 JSR168 Portlet Container. 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 J2EE 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. A JSR 168 compliant portlet application 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 168 Portlet Specification, in which portlets are instantiated, used, and finally destroyed. The JSR 168 Portlet API provides standard interfaces for portlets. Portlets based on this JSR 168 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 131 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.

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** > **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" 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.

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 allows you to directly render a portlet 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 125 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" on page 134, 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 132, 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] [/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 the version '1.0' is allowed. Any differing 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.

The following list includes examples of valid 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

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;
      }
}
```

Portlet deployment descriptor extensions

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 portlet container's default values are used.

The default for the portletServingEnabled attribute is true. The following is an example of how to configure 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>
```

Converting portlet fragments to an HTML document

A portlet only delivers fragment output whereas a servlet typically delivers document output. However, you can use the PortletServingServlet servlet, which is similar to the FileServingServlet servlet, to address portlets like servlets. A default document servlet filter, the DefaultFilter filter, is applied to the PortletServingServlet servlet to return the portlet's content inside of a document. This filter only applies to requests, not to includes or forwards using the RequestDispatcher method. A servlet filter that is used to embed the portlet's content into a document is called the document servlet filter. You can define additional document servlet filters in a .xml file.

The FilterRequestHelper attribute within com.ibm.wsspi.portletcontainer.util is provided to assist the document servlet filters in analyzing a request regarding filter chain and portlet information. It is used in supporting dynamic portlet titles, as a marker for redirection for document servlet filters and to ensure that document conversion is completed once.

Adding a new document servlet filter

The filter capability is a server feature, therefore all filters must be installed into the server to use the filter capability of the server. The filters need to be available in any classes or library directory on a server level. You must also register the filter in a plugin.xml file within the root of a Java archive (JAR) file. The following is an example of how to register the filter in a plugin.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin id="sample.plugin" name="Customer_Plugin" provider-name="Customer" version="1.0.0">
  <extension point="com.ibm.ws.portletcontainer.portlet-document-filter-config">
    <portlet-document-filter class-name="sample.filter.CustomFilter" order="200" />
  </extension>
</plugin>
```

Dynamic portlet titles

The PortletServingServlet servlet supports dynamic portlet titles by providing the dynamic title as a request attribute, FilterRequestHelper.DYNAMIC_TITLE. This attribute returns the dynamic portlet title if it has been set by the portlet, otherwise it returns the static portlet title of the portlet.xml file if defined.

The FilterRequestHelper is used to assist in controling of the dynamic portlet title. The following constant is defined, `DYNAMIC_TITLE = 'javax.portlet.title'`

The DefaultFilter uses this request attribute to set the document title while converting the fragment into a document. If the filter wants to support browser caching or dynamic portlet titles, the complete portlet content must be cached

Redirection for document servlet filters

A document servlet filter can set a marker as request attribute, FilterRequestHelper.REDIRECT. This marker ensures that the portlet container returns to the document servlet filter after the portlet action has been called prior to any render calls. The following constants are defined, `REDIRECT = 'com.ibm.websphere.portlet.action'` and `REDIRECT_VALUE = 'redirect'`. The DefaultFilter uses this request attribute to provide special cache handling for the portlet rendering call to support dynamic title.

Document conversion

The conversion of the portlet's fragment into a valid document must be completed only once. Therefore each document servlet filter must ensure that the fragment has not yet been converted to a document previously. If the document servlet filter converts the fragment to a document, the request attribute FilterRequestHelper.DOCUMENT must be set to FilterRequestHelper.DOCUMENT_VALUE. This request attribute marks whether the conversion still needs to be completed. The following constants are defined, `DOCUMENT = 'com.ibm.websphere.portlet.filter'` and `DOCUMENT_VALUE = 'document'`. The DefaultFilter uses this request attribute to check whether it should convert the fragment to an Hypertext Markup Language (HTML) document. For example, this allows another document servlet filter in front to convert the fragment into a valid Wireless Markup Language (WML) document instead.

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. Thus for MBeans of type portletapplication, the Web module name concatenated with the string ″_portletapplication″ has been chosen as the MBean name. The name chosen for the MBean of type portlet is the name of the MBean of type portletapplication the portlet belongs to, concatenated with the porlet name. A full stop separates the preceding Web module name from the portlet name. For more information about the Portlet and PortletApplication MBean types in the Generated API documentation. 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 ″Bookmark″.

```
String myPortletApplicationName = "Bookmark_war_portletapplication";
This name is composed by the Web module name concatenated with the substring "_portletapplication"


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 attribute webApplicationContextRoot is retrieved on that MBean or the first MBean that is found and the result is stored in the variable ctxRoot. This variable now contains the context root of the Web application that contains the portlet application that was searched. For example, this may be something like, ″/bookmark″.

The next code example demonstrates how to invoke the MBean of type portlet for a portlet with the name ″BookmarkPortlet″.

```
String myPortletName = "Bookmark_war_portletapplication.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.

SIP applications

Providing real time collaboration with SIP applications

Follow these procedures for creating SIP applications and configuring the SIP container.

Session Initiation Protocol (SIP) is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging. 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.

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 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

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 specification (JSR 116) requires all SIP elements to support both UDP and TCP, and optionally TLS, SCTP, and potentially other transports.

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.

Developing SIP applications

This section explains how to develop a SIP application for WebSphere Application Server.

When you develop Session Initiation Protocol (SIP) applications, you should be aware of certain considerations and of the SIP status codes.

Compliance with industry SIP standards

This product complies with various industry standards for the Session Initiation Protocol (SIP)

The standards bodies for these standards are as follows:

IETF Internet Engineering Task Force
JCP   Java Community Process
3GPP Third Generation Partnership Project

SIP and SIP proxy

This product complies with the SIP standards of IETF and JCP shown in Table 1.

Table 1. Compliance with SIP and SIP proxy standards

Standard	Body	Description	Support	Notes
JSR 116	JCP	SIP servlets API	Full	Application composition according to the cascaded services model is supported. Converged applications are supported in environments where session failover is disabled. **Note:** application composition is underspecified in JSR 116. WebSphere Application Server's SIP application composition details can be found in "SIP application composition" on page 143.
RFC 3261	IETF	SIP core protocol	Full	Supersedes RFC 2543 (SIP base protocol). Backward compatible.
RFC 3262	IETF	Reliability of provisional responses SIP	Full	
RFC 3263	IETF	Locating SIP servers	Full	
RFC 3515	IETF	SIP REFER method	Full	

SIP presence server

SIP Server complies with the SIP presence server standards shown in Table 2.

Table 2. Compliance with SIP presence server standards

Standard	Body	Description	Support	Notes
RFC 3265	IETF	Specific event notification	Full	This is a base protocol for the presence server.
RFC 3842	IETF	Message waiting indicator	Partial	This can be supported as part of the presence server as another event package.
RFC 3856	IETF	Presence event package for SIP	Full	This is a base protocol for the presence server.
RFC 3863	IETF	Presence Information Data Format (PIDF)	Full	
RFC 3903	IETF	SIP extension for event state publication	Full	

Other SIP applications

SIP Server complies with standards for other SIP applications as shown in Table 3 on page 139. 3GPP is the Third Generation Partnership Project.

Table 3. Compliance with standards for other SIP applications

Standard	Body	Description	Support	Notes
RFC 2976	IETF	The SIP INFO method	Full	SIP Server contains no explicit support in SLSP or the container for INFO methods, but an application is free to implement and act upon them.
RFC 3326	IETF	The Reason header field	Full	This field enables a SIP container to indicate a reason for a given SIP message. The Reason header is moved to the application for processing.
RFC 3327	IETF	Extension (Path) header field for registering nonadjacent contacts	Full	3GPP requires that registrar and location server applications implement and use the Path header.
RFC 3428	IETF	SIP extension for instant messaging	Full	MESSAGE methods extend SIP and are processed by the application.
RFC 3455	IETF	Private header extensions to SIP	Full	This is an informational RFC on 3GPP private headers. The container passes the headers to the application.
RFC 3725	IETF	Best current practices for third-party call control (3pcc)	Full	Applications running in the SIP container implement these practices.

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 4 on page 140.

Table 4. 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.

Developing a custom trust association interceptor

When you develop Session Initiation Protocol (SIP) applications, you can create a custom trust association interceptor (TAI).

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.

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). The 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 (sent reliably) or provisional (typically not sent reliably). For cases where you need to send a provisional response reliably, you can use the PRACK (Provisional Response ACKnowledgement) method.

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 sendReliabily() method instead of the send() method to send the response.

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.
  - Within your application's doPrack(...) method, prepare the application to generate and send a final response to an incoming PRACK request.
- 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.

SIP application composition

Reference materials for the WebSphere Application Server implementation of JSR 116 standards

Application composition specification

The JSR 116 standard states in Section 2.4 that multiple applications may be invoked for the same SIP request.

This standard requires that implementations must obey a cascaded services model, stating: "Triggering of service applications on the same host, shall be performed in the same sequence as if triggering had occurred on different hosts." This means that responses will flow upstream, that they will hit applications in the reverse order of their corresponding requests.

JSR 116 does not specify how to implement this when developing SIP applications, thus there are many ways to comply with this standard.

Application composition in the WebSphere Application Server environment

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. Upon finding the nth match, the container then invokes the corresponding servlet.
- If the servlet needs to proxy the request, the container re-scans the rules searching for additional matches. Upon finding the $(n+1)^{th}$ match, the container invokes the corresponding servlet.
- Matching the request excludes any servlet in the same application as the previously invoked servlet. As stated in the standards, no servlet will be invoked twice for the same SIP request.

Deploying SIP applications

Use the administrative console to customize your Session Initiation Protocol (SIP) application installation

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 2 Platform Enterprise Edition (J2EE) 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.

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**.

   Remember: 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.

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.

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 J2EE application.

1. Design a J2EE application and the enterprise beans that it needs. For links to design information that is specific to enterprise beans, see "Data access: Resources for learning" on page 688.
2. Develop any enterprise beans that your application will use.
3. Prepare for assembly. For your EJB 2.x-compliant entity beans, decide on an appropriate access intent policy.
4. 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.
5. Assemble the modules into a J2EE application using the assembly tool.
6. 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.
7. Deploy the application in an application server.
8. Test the modules.
   - As needed, debug problems with the container.
   - Debug access problems.
9. Assemble the production application using one of the assembly tools
10. Deploy the application to a production environment.
11. Manage the application:
    a. Manage installed EJB modules. After an application has been installed, you can manage its EJB modules individually through the Assembly Service Toolkit.
    b. Manage other aspects of the J2EE application.
12. Update the module and redeploy it using one of the assembly tools.
13. 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 J2EE applications. There are three types of enterprise beans, *entity* beans, *session* beans, and *message-driven* beans.

All beans reside in EJB containers, which provide an interface between the beans and the application server on which they reside.

Entity beans 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.

Session beans typically contain the high-level and mid-level business logic for an application. Each method on a session bean typically 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.

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() 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.

For more information about enterprise beans, see "Enterprise beans: Resources for learning" on page 148.

EJB modules
An 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.

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 J2EE application. An EJB module is installed and run in an enterprise bean container.

For more information about EJB modules, see "Enterprise beans: Resources for learning" on page 148

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.

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, WebSphere Application Server 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 155.
  – 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.

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 they must exist before messages are posted for them), Startup Beans (which must be processed at server startup time), and those EJB types that you specify to initialize at server start. For more information about disabling quick start for EJB types, see Changing enterprise bean types to initialize at application start time using the Application Server Toolkit.

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.

For more information about EJB containers, see "Enterprise beans: Resources for learning" on page 148.

Enterprise beans: Resources for learning

Use the following links to find relevant supplemental information about enterprise beans. 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.

Planning, business scenarios, and IT architecture

* Mastering Enterprise JavaBeans

  A comprehensive treatment of Enterprise JavaBeans (EJB) programming in nonprintable form (PDF). One must be registered to download the PDF, but registration is free. Information about purchasing a hardcopy is available on the Web site.
* *Enterprise JavaBeans* by Richard Monson-Haefel (O'Reilly and Associates, Inc.: Third Edition, 2001)

Programming model and decisions

* Read all about EJB 2.0

  A comprehensive overview of the 2.0 specification that is still relevant to users of EJB 2.1.
* The J2EE Tutorial

  This set of articles by Sun Microsystems covers several EJB-related topics, including the basic programming models, persistence, and EJB Query Language.

Programming instructions and examples

* WebSphere Application Server Development Best Practices for Performance and Scalability

  Programming practice for enterprise beans and other types of J2EE components.
* Optimistic Locking in IBM WebSphere Application Server 4.0.2

  Examples of the effect of optimistic concurrency on application behavior. Although the paper is based on a previous version of this product, the data access issues discussed in it are current.

  This paper does not seem to be available directly by URL. To view this paper, visit the specified URL and search on `"optimistic locking"`

Programming specifications

* Enterprise JavaBeans 2.1 Specification

  You can download the specification from this URL.
* Enterprise JavaBeans 3.0 Specification

  You can download the specification from this URL.
* Java^{TM} 2 Platform: Compatibility with Previous Releases

  This Sun Microsystems article includes both source and binary compatibility issues.

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 Java bean (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 queue, due to the use of unbounded threads.

The following illustration depicts the two queuing options of enterprise beans.

EJB Queuing
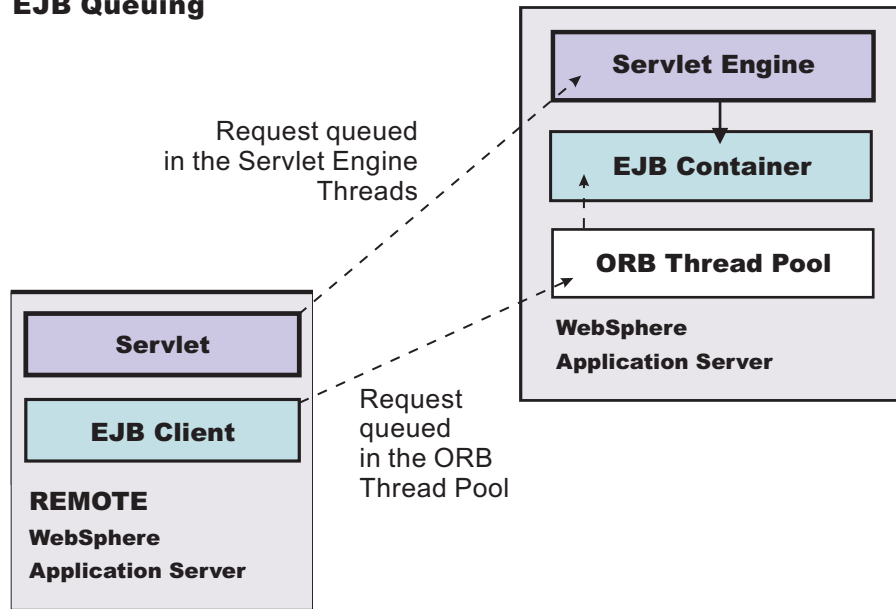
The following are two tips for queueing enterprise beans:
- **Analyze the calling patterns of the EJB client**.

  When configuring the thread pool, it is important to understand the calling patterns of the EJB client. If a servlet is making a small number of calls to remote enterprise beans and each method call is relatively quick, consider setting the number of threads in the ORB thread pool to a value lower than the Web container thread pool size value.

Short-lived EJB calls

Remote Call   Remote Call

Servlet service()   Servlet service()
BEGIN   **END**
Execution timeline

Longer-lived EJB calls

Remote Call   Remote Call

Servlet service()   Servlet service()
BEGIN   **END**
Execution timeline
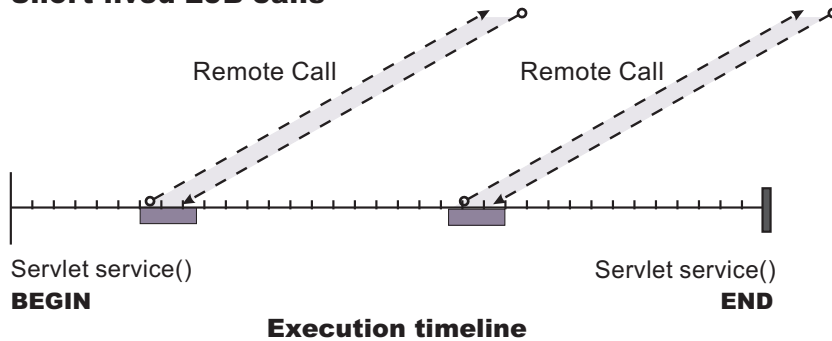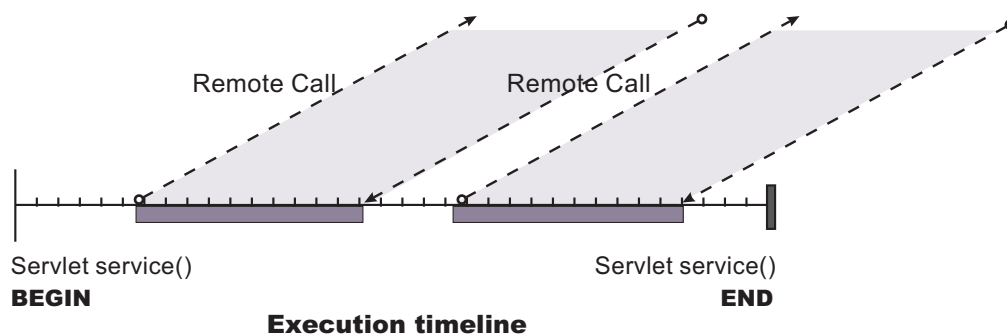
The degree to which the ORB thread pool value needs increasing is a function of the number of simultaneous servlets, that is, clients, calling enterprise beans and the duration of each method call. If the method calls are longer or the applications spend a lot of time in the ORB, consider making the ORB thread pool size equal to the Web container size. If the servlet makes only short-lived or quick calls to the ORB, servlets can potentially reuse the same ORB thread. In this case, the ORB thread pool can be small, perhaps even one-half of the thread pool size setting of the Web container.

- **Monitor the percentage of configured threads in use**.

    Tivoli Performance Viewer shows a metric called *percent maxed*, which is used to determine how often the configured threads are used. A value that is consistently in the double-digits, indicates a possible bottleneck a the ORB. Increase the number of threads.

See also Queuing network

Enterprise bean and EJB container troubleshooting tips

If you are having problems starting an 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 JVM log files for the application server which hosts the container. Look for the message **server** *server_name* **open for e-business** in the SystemOut.log . If it does not appear, or if you see the message **problems occurred during startup**, browse the SystemErr.log for details.
- Browse the system log files for the application server which hosts the container.
- Enable tracing for the EJB Container component, by using the following trace specification EJBContainer=all=enabled. Follow the instructions for dumping and browsing the trace output to narrow the origin of the problem.

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.

Error in client log: Missing jar file:

The following error message appears in the client log file because a 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.

Cannot access an enterprise bean from a servlet, a JSP file, a stand-alone program, or another client

This article provides 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 CMP EJB module fails with **javax.naming.NameNotFoundException: *dataSourceName***
* **Transaction [tran ID] has timed out after 120 seconds** error accessing EJB.

* Symptom: **CNTR0001W: A Stateful SessionBean could not be passivated**
* Symptom: **org.omg.CORBA.BAD_PARAM: Servant is not of the expected type. minor code: 4942F21E completed: No** returned to client program when attempting to execute an EJB method

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 JVM logs of the application server hosting the enterprise bean as well as log files of the client.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, perform these steps:
1. If the problem appears to be name-service related, which means that you see a NameNotFoundException, or a message ID beginning with NMSV, see these topics for more information:
   * Cannot look up an object hosted by WebSphere Application Server from a servlet, JSP file, or other client
   * Naming service troubleshooting tips

2. Check to see if the problem is identified and documented using the links in Diagnosing and fixing problems: Resources for learning.

If you still cannot fix your problem, seeTroubleshooting help from IBM for further assistance.

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 specification (available at `http://java.sun.com/products/ejb/docs.html`), section 7.6.2, Dealing with exceptions.

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:`. To obtain a dump of a server `local:` name space, use the name space dump utility described in the article ″Name space dump utility for `java:`, `local:` and `server` name spaces.″

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 could not successfully complete. When this exception is thrown, browse the JVM 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. Open the enterprise bean .jar file (or the EAR file containing the enterprise bean) and select the Session Beans or Entity Beans object in the component tree on the left. The Unresolved Action property is on the IBM Extensions tab of the container properties.

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 could not be bound into the namespace.
  - Look for additional naming warnings or errors in the status bar, and in the hosting application server JVM logs. Check any further naming-exception problems that you find by looking at the topic Cannot look up an object hosted by WebSphere Application Server from a servlet, JSP file, or other client.

Transaction [tran ID] has timed out after 120 seconds accessing an enterprise bean

This error can occur when a client executes a transaction on a CMP or BMP enterprise bean.
- 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, go to **Manage Application Servers > *server_name***, select the **Transaction Service properties** page, and look at the property **Total transaction lifetime timeout**. Increase this value if necessary and save the configuration.

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 JVM log for the EJB container that hosts the enterprise bean, and looks similar to:

```
[time EDT] <ThreadID> StatefulPassi W CNTR0001W:
A Stateful SessionBean could not be passivated: StatefulBean0
(BeanId(XXX#YYY.jar#ZZZZ, <ThreadID>),
state = PASSIVATING) com.ibm.ejs.container.passivator.StatefulPassivator@<ThreadID>
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, and *<ThreadID>* is the thread ID for that run.

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. See the enterprise bean specification mandating this requirement, specifically section 7.4 in the EJB specification Version 2.1. 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.

Symptom: org.omg.CORBA.BAD_PARAM: Servant is not of the expected type. minor code: 4942F21E completed: No

This error can be returned to a client program when the program attempts to execute an EJB method.

Typically this problem is caused by a mismatch between the interface definition and implementation of the client and server installations, respectively.

Another possible cause is when an application server is set up to use a single class loading scheme. If an application is uninstalled while the application server remains active, the classes of the uninstalled application are still loaded in the application server. If you change the application, redeploy and reinstall it on the application server, the previously loaded classes become back level. The back level classes cause a code version mismatch between the client and the server.

To correct this problem:
1. Change the application server class loading scheme to multiple.
2. Stop and restart the application server and try the operation again.
3. Make sure the client and server code version are the same.

Developing enterprise beans

In selecting a tool for developing enterprise beans, there are two basic approaches, with or without an IDE. The steps in this article explain development without an IDE.

Design a J2EE application and the enterprise beans that it needs.
- For general design information, see ″Resources for learning.″
- Before developing entity beans with container-managed persistence (CMP), read ″Concurrency control.″

There are two basic approaches to selecting tools for developing enterprise beans:
- You can use one of the available integrated development environments (IDEs). IDE tools automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. The IBM WebSphere Application Developer product is the recommended IDE. For more information, see the documentation for that product.
- 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.

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 Enterprise JavaBeans (EJB) specification.
2. Write and compile the components of the enterprise bean.
   - 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 define a local home interface as well.

     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.
   - Create a database schema for the entity bean's persistent data.
     - For entity beans with container-managed persistence (CMP), you must store the bean's persistent data in one of the supported databases. The Application Service Toolkit 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 the IBM Rational Application Developer product to generate code for the database tables.
     - 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, consult your database documentation.
   - **(CMP entity beans for EJB 2.x only)** 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 `NameBeanFinderHelper`, 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.

Assemble the beans in one or more EJB modules.

Developing read-only entity beans

In addition to the existing EJB caching options, you can develop read-only entity beans.

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 the application assembly tooling (either WebSphere Application Developer or the Application Server Toolkit).

1. Start your assembly tool.

2. Call up the parameter selection list as you normally do.

3. Set the **Activate At** parameter to *ONCE*. This is the same as for standard option A caching.

4. Set the **Load At** parameter to either *INTERVAL*, *DAILY*, or *WEEKLY.*

   INTERVAL
   > causes the bean to be reloaded if a certain time interval has been exceeded since the last time the bean was loaded.

   DAILY causes the bean to be reloaded on the first business method invocation that occurs after a specified time on the host computer's local time-of-day clock.

   WEEKLY
   > is similar to DAILY except it occurs once per week at a specified time.

5. Set the **Reload Interval** parameter to a nonnegative integer value. The meaning of this parameter depends on whether the Load At parameter is INTERVAL, DAILY, or WEEKLY.

   INTERVAL
   > the integer represents the number of minutes that can elapse (since the last time the bean was loaded) before the EJB container reloads the bean's state from persistent storage. A value of 0 is permissible and causes the container to never reload the state of the bean.

   DAILY the integer represents an absolute time each day that the reload is performed after, expressed in what is commonly called the 24-hour clock. That is, a whole number between 0000 and 2359, where 0000 represents midnight, 1200 represents noon, and 2359 represents one minute before midnight. Any leading zeroes on this number are optional. In the case of malformed values (for example, 1285), the resulting clock time is always computed by taking the minutes value from the two least-significant digits and adding that to the hour value taken from the digit or digits to the left of the two least-significant ones. Thus, a value of 1285 will be interpreted as 1325 (85 minutes after 1200). Any values exceeding 2359, as well as negative or nonnumeric values, are interpreted as 0000.

   WEEKLY
   > the integer is encoded in the same manner as daily, except it must be five digits in length, the first digit representing the day of the week. Sunday is encoded as **1** and Saturday is encoded as **7**. If the value is four digits or less, it is treated as if it were five digits long with the first digit being **1**.

   Reloading is performed only in response to a business method invocation on the bean. When a business method is invoked, the EJB container checks to see whether either the reload interval time has expired or the absolute clock time for that day has passed (depending on whether INTERVAL, DAILY, or WEEKLY was used). If so, the container reloads the bean state.

   When a read-only entity bean is invoked within a global transaction and the reload interval expires while the transaction is active, business method calls on the bean during that transaction continue to see the non-reloaded state of the bean for the duration of that transaction. That is, a snapshot of the bean state is effectively taken on the first business method invocation on that bean during a transaction, and that state continues to be in effect for that transaction until it completes. New invocations on that bean performed in a different transaction after the reload see the reloaded state.

Example: read-only entity bean:

A usage scenario and example for writing an 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 Version 2.1 of the Enterprise JavaBeans (EJB) specification is added for Version 6 of this product. Migration of enterprise beans deployed in Versions 4 or 5 of this product is not generally necessary; Versions 1.1 and 2.0 of the EJB specification are still supported.

Follow these steps as appropriate for your application deployment.
1. Modify enterprise bean code for changes in the specification.
    * For Version 1.0 beans, migrate at least to Version 1.1.
    * As stated previously, migration from Version 1.1 to Version 2.x of the EJB specification is not required for redeployment on this version of the product. However, if your application requires the capabilities of Version 2.x, migrate your Version 1.1-compliant code.

> **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 so 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. You might have to modify code for some EJB 1.1-compliant modules that were not migrated to Version
   2.x. Use the following information to help you decide.
   - Some stub classes for deployed enterprise beans have changed in the Java 2 Software
     Development Kit, Version 1.4.1.
   - The task of generating deployment code for enterprise beans changed significantly for EJB
     1.1-compliant modules relative to EJB 1.0-compliant modules.

3. Ensure WebSphere Application Server 32-bit to 64-bit compatibility. For a ″pure″ Java application this
   is not an issue. However, if your application code utilizes the Java Native Interface (JNI) code, you
   should be aware of the following: the JNI allows Java code running in a virtual machine to operate with
   applications and libraries written in other languages, such as C, C++, and assembly. So, if your J2EE
   application uses JNI in a 32-bit environment, your code must be recompiled in the 64-bit environment.
   It is possible that the JNI calls could be different after the compilation, as the JNI specifications can
   change from version to version.

4. Reassemble and redeploy all modules to incorporate migrated code.

Migrating enterprise bean code from Version 1.0 to Version 1.1:

The following information generally applies to any enterprise bean that currently complies with Version 1.0
of the Enterprise JavaBeans (EJB) specification.

For more information about migrating code for beans produced with Rational Application Developer, see
the documentation for that product. For more information about migrating code in general, see ″Resources
for learning.″

1. In session beans, replace all uses of javax.jts.UserTransaction with javax.transaction.UserTransaction.
   Entity beans may no longer use the UserTransaction interface at all.

2. In finder methods for entity beans, include FinderException in the `throws` clause.

3. Remove throws of java.rmi.RemoteException; throw javax.ejb.EJBException instead. However,
   continue to include RemoteException in the `throws` clause of home and remote interfaces as required
   by the use of Remote Method Invocation (RMI).

4. Remove uses of the finalize() method.

5. Replace calls to getCallerIdentity() with calls to getCallerPrincipal(). The use of getCallerIdentity() is
   deprecated.

6. Replace calls to isCallerInRole(Identity) with calls to isCallerinRole (String). The use of
   isCallerInRole(Identity) and java.security.Identity is deprecated.

7. Replace calls to getEnvironment() in favor of JNDI lookup. As an example, change the following code:

   ```
   String homeName =
      aLink.getEntityContext().getEnvironment().getProperty("TARGET_HOME_NAME");
   if (homeName == null) homeName = "TARGET_HOME_NAME";
   ```

   The updated code would look something like the following:

   ```
   Context env = (Context)(new InitialContext()).lookup("java:comp/env");
   String homeName = (String)env.lookup("ejb10-properties/TARGET_HOME_NAME");
   ```

8. In ejbCreate methods for an entity bean with container-managed persistence (CMP), return the bean's
   primary key class instead of `void`.

9. Add the getHomeHandle() method to home interfaces.

   ```
   public javax.ejb.HomeHandle getHomeHandle() {return null;}
   ```

Consider enhancements to match the following changes in the specification:
- Primary keys for entity beans can be of type java.lang.String.
- Finder methods for entity beans return java.util.Collection.
- Check the format of any JNDI names being used. Local name spaces are also supported.
- Security is defined by role, and isolation levels are defined at the method level rather than at the bean level.

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.

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. For more information about migrating code in general, see ″Resources for learning.″

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.

Note: In Application Developer Version 5, there is a J2EE Migration wizard to migrate the EJB beans within an EJB 2.1 project from 1.x into 2.1 (you cannot just migrate individually selected beans). The wizard performs migration steps #1 to #2 above. It also migrates EJB 1.1 (proprietary) relationships into EJB 2.1 (standard) relationships, and maintains EJB inheritance.

Adjusting exception handling for EJB wrappered applications migrating from version 5 to version 6:

Because of a change in the Java APIs for XML based Remote Procedure Call (JAX-RPC) specification, EJB applications that could be wrappered in WebSphere Application Server version 5.1 cannot be wrappered in version 6 unless you modify the code to the exception handling of the base EJB application.

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 6, 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 that IBM provides with this 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 (such 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 IBM Rational Application Developer product.

Optimistic concurrency control for container-managed persistence

This product supports optimistic concurrency control of data access. See "Concurrency control" on page 182 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 200 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 "Entity bean assembly settings."

Some enterprise beans created with the IBM Rational Application Developer product 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.

Best practices for developing enterprise beans

Use the following guidelines when designing and developing enterprise beans.
- Use a stateless session bean to act as the entry point for business logic. For more information about using session facades, see "Resources for learning."
- Entity beans should use container-managed persistence.
- In an Enterprise JavaBeans (EJB) Version 2.x environment, 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 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. 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 WebSphere Application Server enables you to make the same settings using the Application Server Toolkit (AST).

Deferred Create for container managed persistence:

The specification for Enterprise JavaBeans (EJB) 2.x states that for Container Managed Persistence (CMP) during the *ejbCreate*, the container can create the representation of the entity in the database immediately, or defer it to a later time.

The WebSphere Application Server versions 5.0.2 and later enable you to take advantage of this specification. 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 WebSphere Application Server enables you to make the same settings using the Application Server Toolkit (AST).

Partial column updates for container managed persistence:

Previously, the WebSphere Application Server implementation of the Container Managed Persistence (CMP) bean method *ejbStore* always 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.

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 169.

Affects on 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 i/o since there is less data to be transmitted.

- by saving any processing time for non-trivially mapped columns (if a column uses converters/composers/transformations), by partially injecting 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 performance in general, it can adversely affect performance too.
- If you enable partial update for a bean for which your application modifies several different combinations of columns during the same time span, then 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 again and again, decreasing performance for all CMP functions (not just limited to ejbStore()).
- 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 now 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, to execute the preparedStatement setXXX() calls, and so on).

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 (few columns) with simple data types and no update triggers. The cost to assemble the partial query dynamically would probably outweigh 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 might be 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 typically.
- 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 many different partial queries and fills up the prepared statement cache quickly. But if the bean does not have too many columns (four or less) and it has complex data types, then you might consider turning partial update on, with the option of increasing the statement cache size to allow for the 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 of 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 indeed increase performance when viewed across all the beans for which both updates are on.

So you should determine which situation gives the best performance for your application: batch update only or partial update only or both (with grouppartialupdate flag set to true).

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.
```

Explicit invalidation in the Persistence Manager cache:

Container managed persistence (CMP) entity beans can be configured as *long-lifetime* beans. A long-lifetime bean is one that is configured with *Lifetime In Cache Usage* equal to a value other than the default **OFF** . A value other than **OFF** means that data for this bean is cached beyond the end of the transaction in which the bean was obtained by a finder or other method. The *Lifetime In Cache Usage* and *Lifetime In Cache* values control the basic length of time the cached data remains valid. When the specified time runs out, the cached data is no longer valid. See the *LifetimeInCache* help sections of the Assembly Service Toolkit (AST) for more details.

However, there is also an API that lets the client application code explicitly invalidate the cached data of a bean on demand, superseding the basic lifetime of the cache data as controlled by the *Lifetime In Cache Usage* and *Lifetime In Cache* settings. This is useful where an application that does not use CMP beans modifies the data that underlies a CMP bean (for example, it updates a database table to which a CMP bean is mapped). Such an application can inform WebSphere Application Server that any cached version of this bean data is **stale** and no longer matches what is in the database. The data should be invalidated (in essence, discarded). For CMP beans that cannot tolerate stale data, this is an important feature.

Because the PM Cache Invalidation mechanism does consume resources in exchange for its benefits, it is not enabled by default. To enable it refer to Setting Persistence Manager Cache Invalidation .

Example: Explicit invalidation in the persistence manager cache:

The usage scenario for this feature begins with configuring one or more bean types to be long-lifetime beans and configuring the necessary Java Message Service (JMS) resources, which is described below. After this is done, the server is started.

Usage Scenario

The scenario continues as follows:
1. Assume that a CMP entity bean of type *Department* has been configured to be a long-lifetime bean.
2. Transaction 1 begins. Application code looks up *Department*'s home and calls a finder method (such as *findByPrimaryKey("dept01")*). As this is the first finder to return *Department dept01*, a trip is made to the database to obtain the data. Transaction 1 ends.
3. Transaction 2 begins. Application code calls *findByPrimaryKey("dept01")* again. Because this is not the first finder to return *Department dept01*, we get a cache hit and no database trip is made. Transaction 2 ends. So far this is current WebSphere Application Server behavior for long-lifetime beans.
4. Another application, which does not use the *Department* CMP bean, is executed. This application might or might not be run on WebSphere Application Server; it could be a legacy application. The application updates the database table that is mapped to the *Department* bean, altering the row for *dept01*. For example, the *budget* column in the table (mapped to a Java double CMP attribute in the *Department* bean) is changed from $10,000.00 to $50,000.00. This application was designed to

cooperate with WebSphere Application Server. After performing the update, the application sends an invalidate request message to invalidate the (now incorrect) cached data for *Department* bean *dept01*.

5. Transaction 3 begins. Application code looks up *Department*'s home and calls a finder method (such as *findByPrimaryKey("dept01")*). Because this is the first finder after *Department dept01* is invalidated, a new database trip is made to obtain the data. Transaction 3 ends.

Persistence Manager cache invalidation API

The PM cache invalidation API is in the form of a JMS message that the client sends to a specially named JMS topic using a connection from a specifically named JMS *TopicConnectionFactory*. The JMS message must be an *ObjectMessage* created by the client. The client code creates a *PMCacheInvalidationRequest* object that describes the bean data to invalidate. Client code places the *PMCacheInvalidationRequest* object in the *ObjectMessage* and publishes the *ObjectMessage* (for further details on the JMS objects and terms used here, please see the Java Message Service documentation).

The public class *PMCacheInvalidationRequest* is central to the API, so we include a portion of its code here for illustration purposes (if you see any differences between this illustration and the actual class, the class is to be considered correct):

```
packagecom.ibm.websphere.ejbpersistence;

/**
*An instance of this class represents a request to invalidate one or more
*CMP beans in the PM cache. When an invalidate occurs, cached data for this
*bean is removed from the cache; the next time an application tries to find
*this bean, a fresh copy of the bean data is obtained from the data store.
*
*The ability to invalidate a bean means that a CMP bean may be configured
*as a long-lifetime bean and thus be cached across transactions for much
*greater performance on future attempts to find this bean. Yet when some
*outside mechanism updates the bean data, sending an invalidation request
*will remove stale data from the PM cache so applications do not behave falsely
*based on stale data.
*/
public class PMCacheInvalidationRequestimplementsSerializable{

. . .

/**
 * Constructor used to invalidate a single bean
 * @param beanHomeJNDIName the JNDI name of the bean home. This is the same value
 * used to look up the bean home prior to calling findByPrimaryKey, for example.
 * @param beanKey the primary key of the bean to be invalidated. The actual
 * object type must be the primary key type for this bean type.
 */
public PMCacheInvalidationRequest(String beanHomeJNDIName, Object beanKey)
 throws IOException {
. . .
}
/**
 * Constructor used to invalidate a Collection of beans
 * @param beanHomeJNDIName java.lang.String the JNDI name of the bean home.
 * This is the same value used to look up the bean home prior to calling
 * findByPrimaryKey, for example.
 * @param beanKeys a Collection of the primary keys of the beans to be
 * invalidated. The actual type of each object in the Collection must be the
 * primary key type for this bean type.
*/
 public PMCacheInvalidationRequest(String beanHomeJNDIName, Collection beanKeys)
 throws IOException {
. . .
}
/**
 * Constructor used to invalidate all beans of a given type
```

```
 * @param beanHomeJNDIName java.lang.String the JNDI name of the bean home.
 * This is the same value used to look up the bean home prior to calling
 * findByPrimaryKey, for example.
 */
public PMCacheInvalidationRequest(String beanHomeJNDIName) {
. . .
}

}
```

If the client wants to perform the invalidation in a synchronous way, it can opt to receive an acknowledgement JMS message when the invalidation is complete. To ask for an acknowledgement (ACK) message, the client sets a *Topic* of its own choosing in the *JMSReplyTo* field of the *ObjectMessage* for the invalidation request (see the Java Message Service documentation for further details). The client then waits (using the *receive()* method of JMS) on receipt of the acknowledgement message before continuing execution.

An ACK message enables the caller to insure there is not even a brief (seconds or less) window during which PM cache data is stale. The sending of an acknowledgement for each request does, of course, take a bit more time and so is recommended to be used only when needed.

The JMS resources used to make an invalidation request--topic connection factory, topic destination (sometimes called just ″topic″), and so forth--must be configured by the user (using the administrative console or other method) if they want to use PM Cache Invalidation. In this way the user can chose whichever JMS provider is preferred (as long as it supports pub-sub). The names that must be used for these resources are defined as part of the API. These names are unique to the WebSphere Application Server namespace to avoid name conflict with customer JMS resources.

The following are the names that must be used when the user configures the JMS resources (shown as Java constants for clarity):

```
// The JNDI name of the topic connection factory
 private static final String topicConnectionFactoryJNDIName = "com.ibm.websphere.ejbpersistence.InvalidateTCF";
// The JNDI name of the topic destination
  private static final String topicDestinationJNDIName = "com.ibm.websphere.ejbpersistence.invalidate";
// The topic name (part of the topic destination)
  private static final String topicString = "com.ibm.websphere.ejbpersistence.invalidate";
```

Other JMS configuration, such as bus name (required if you choose the *default messaging* JMS provider), can use names you define. Also, the bus used by the invalidate JMS resources can be used by other resources; the invalidate mechanism does not require exclusive use of a bus.

Here are examples of how these constants can be used in client code:

```
// Look up the topic connection factory...
InitialContext ic = new InitialContext();
TopicConnectionFactory topicConnectionFactory = (TopicConnectionFactory) ic.lookup(topicConnectionFactoryJNDIName);
...
// Look up the topic
Topic topic = (Topic) ic.lookup(topicDestinationJNDIName);
```

Note that JMS messages can be sent not only from J2EE application code (for example, a SessionBean or BMP entity bean method) but also from non-J2EE applications if your chosen JMS provider allows for this. For example, the IBM MQ Client product installed on a database server, which typically does not have J2EE installed to create a topic connection and topic that are compatible with the topic connection factory and topic destination you configure using the WebSphere Application Server administrative console.

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 batched commands with the assembly tools

This article explains how to set the run time for batched commands using the Application Server Toolkit.

1. Start an assembly tool. Refer to *Starting WebSphere Application Server Toolkit* in the Application Server Toolkit documentation.

2. On the Project Explorer tab, click **EJB Modules** > *project* > **ejbModule** > **META-INF** > **ejb-jar.xml** The EJB Deployment Descriptor window appears.

3. Select the **Access** tab. The Add Access Intent window appears. There are two areas of the panel that deal with adding access intent:
   - Default Access Intent for Entities 2.x (Bean Level)
   - Access Intent for Entities 2.x (Method Level)

4. Select the Bean or Method level. Another access intent window appears where you can set the properties you wish to use.

5. Use the dropdown list to select the Access intent name.

6. **Optional:** Enter a description.

7. Check the **Persistence Option** box.

8. Check the **Deferred Operation** box.

9. Use the dropdown list to select **All** for deferred operation. You must select All to use the batch option.

10. Check the **Batch** box. This operation uses the JDBC batch command to insert, update, or delete rows in the database backend that this particular enterprise bean is connected to.

11. Select **Finish**.

Setting the run time for deferred create with JVM arguments

The specification for Enterprise JavaBeans (EJB) 2.x states that for Container Managed Persistence (CMP) during the ejbCreate, the container can create the representation of the entity in the database immediately, or defer it to a later time.

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 the run time for deferred commands with the assembly tools

To set the run time for deferred commands using the assembly tools, follow these steps.

1. Start the Application Server Toolkit.

2. On the Project Explorer tab, click **EJB Modules** > *project* > **ejbModule** > **META-INF** > **ejb-jar.xml** The EJB Deployment Descriptor window appears.

3. Select the **Access** tab. The Add Access Intent window appears. There are two areas of the panel that deal with adding access intent:

   - Default Access Intent for Entities 2.x (Bean Level)
   - Access Intent for Entities 2.x (Method Level)

4. Select the Bean or Method level. Another access intent window appears where you can set the properties you wish to use.

5. Use the dropdown list to select the Access intent name.

6. **Optional:** Enter a description.

7. Check the **Persistence Option** box.

8. Check the **Deferred Operation** box.

9. Use the dropdown list to select your choice for deferred operation. You have three options for deferred operation:

   NONE Nothing is deferred.

   CREATE_ONLY
   > Only ejbCreate commands are deferred until the next ejbStore occurs to create row in database.

   ALL All ejbCreate, ejbStore, and ejbRemove commands are deferred until a flush is needed, which is either before a finder method or before transaction completion.

10. Select **Finish**.

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.

1. Start the Application Server Toolkit.

2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.

3. Open the Project Explorer view. Click **Window > Show View > Project Explorer**.

4. Open the EJB Deployment Descriptor. Click **EJB Projects > *project* > ejbModule > META-INF > ejb-jar.xml**. The Deployment Descriptor editor opens.

5. In the Deployment Descriptor editor, select the **Access** tab. The access page opens.

6. In the **Default Access Intent for Entities 2.x (Bean Level)** section of the access page, select the bean for which you want to set partial operation. If an access intent has already been configured for this bean, click on the **Edit** button to edit the access intent policy. Otherwise, click on the **Add** button to add an access intent policy to the bean. This opens the **Add access intent** window.

7. Select the **Persistence Option** check box if it is not already checked.

8. Select the **Partial Operation** check box. Use the drop-down list next to the Partial Operation check box to select your preference.
   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.

9. Select **Finish**.

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*.

Unknown primary-key class

When writing an entity bean for Enterprise JavaBeans Version 2.1, 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 library.

Configuring a Timer Service

To configure a timer service, follow these steps.

1. Open the administrative console.
2. Click **Servers** >**Application Servers** > *servername* > **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 Cloudscape 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 172

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 1280 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.

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 Cloudscape, 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 172

   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: 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 normally 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().

```
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) Timer.getInfo():
     System.out.println("I was called because of"+ whyWasICalled);
  } // end of method ejbTimeout
```

In this section, a Timer is created that executes 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 cause the ejbTimeout method to invoke in 30 seconds.
public void fireInThirtySeconds() throws EJBException  {

  TimerService theTimerService = theEJBContext.getTimerService();
  String aLabel = "30SecondTimeout";
  Timer theTimer = theTinmerService.createTimer(30000, aLabel);

} // end of method fireInThirtySeconds

} // end of class MyBean
```

EJB Timer Service settings:

Use this page to configure and manage the EJB Timer Service for a specific EJB container.

To view this administrative console page, click **Servers** >**Application Servers** > *servername* > **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:

WebSphere Application Server 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 JavaBeans 2.1 for the timer service

In WebSphere Application Server, the **EJB Timer Service** implements 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 Cloudscape 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 170.

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 172 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 *findTimers()* 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 *findTimers()* 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.

- **A note about deadlock and access intent:** 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 170 for information on how to configure the data source (database) to be used for each server process timer service. Note that 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 userid 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 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 "Example: FindEJBTimers command" on page 177.

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 "Example: CancelEJBTimers command" on page
178.

Example: FindEJBTimers command:

The following examples illustrate how to use the command to find EJB timers and explain the output
statement.

To use the findEJBTimers command to find *all* Enterprise JavaBeans (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.

Example: CancelEJBTimers command:

The following examples illustrate how to use the command to cancel 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 service support

WebSphere Application Server Version 6.0 complies with the Java 2 platform, Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) 2.1 specification by enabling you to expose an EJB stateless session bean as Web service.

You can do this by simply 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 a Web service from an enterprise bean" on page 444.

Binding Web modules to virtual hosts

Web modules must be bound to specific virtual hosts. By associating a Web module to a specific host, you are specifying that all requests that match this virtual host must be handled by the Web application containing the binding.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. In the J2EE view, select the Web module to open its deployment descriptor.
4. On the Overview page, find the WebSphere bindings section.
5. Specify the virtual host name.
6. **Save** the deployment descriptor.

Binding EJB and resource references

Follow these steps to bind an enterprise bean local reference (or nickname) to a Java Naming and Directory Interface (JNDI) name.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. In the J2EE view, select the EJB module to open its deployment descriptor.
4. Switch to the References page.
5. Expand the tree under your chosen bean and select the appropriate reference.
6. In the WebSphere bindings section, specify the JNDI name.
7. Repeat these steps for all the references in the EJB module.
8. **Save** the deployment descriptor.

> **Note:** Reference bindings can be defined or overridden at deployment time in the administrative console for all modules except for application clients. For those, you must use the Application Server Toolkit.

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 artifacts (enterprise beans or resources) defined in the application server.

Create a data source or JDBC resource and give it a Java Naming and Directory Interface (JNDI) name.

For more information, see "Application bindings" on page 1396.

The following steps assume that the entity beans in your application are container-managed persistence (CMP) enterprise beans. 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.

1. Start the Application Server Toolkit.

2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.

3. In the J2EE view, select the EJB module or individual EJB to open its deployment descriptor.

4. Find the WebSphere bindings section.

5. In the JNDI name field, enter the name of the data source or JDBC resource you want to use.

6. Specify whether the authentication is handled at the container or application level.

7. **Save** the deployment descriptor.

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.

For more information about EJB containers, see "Enterprise beans: Resources for learning" on page 148.

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 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.

You can apply lightweight local mode to specific EntityBean types within your application in two ways. You can use application server tooling, or the Marker interface technique.

Using Application Server Tooling:

1. Start the Application Server Toolkit.
2. Select the EJB deployment descriptor of the entity bean that you want to work with.
3. In the **property** pane, select the **WebSphere Extension** tab.
4. Check the box labeled *Use Lightweight Local mode* .
5. Select OK.
6. Save your changes.

Marker interface technique:

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 documentation 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.

You apply access intent policies to EJB Version 2.0 (and later) entity beans and their methods by using an application assembly tool. A set of default access intent policies comes with the Application Server Toolkit (AST) .

1. Apply default access intent to CMP entity beans. For more information, see the online help available with the Application Server Toolkit.
2. Apply access intent policies to methods of CMP entity beans.

Access intent policies

An access intent policy is a named set of properties (access intents) that governs 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 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/update policy can be qualified further. The selected policy determines the appropriate isolation level and locking strategy used by the run time environment.

transition: Access intent policies are specifically designed to supplant 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 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 intent has been deprecated for Version 6.

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 such 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-middle mapping. The fields used in overqualified updates are specified in the ejb-rdb mapping. If nullable columns are selected as overqualified columns, then 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 throws 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 over which a given resource would be 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 afterwards. 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 whether 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.

best-practices: Whether or not 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 would be 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 (*RelB* and so on) 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.
  - When a read-ahead hint contains a table join

    Different access intents can result in requiring a SELECT FOR UPDATE statement. Check the matrix on the JDBC driver and SELECT FOR UPDATE support to see if readAhead is enabled.

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 can use an optimistic locking approach.

When accessing data in a database concurrently, an application must be aware of and prepared for database locking that must occur to insure the integrity of the data.

If an entity bean performs a findByPrimaryKey (which by default obtains a 'Read' lock in the database), and the entity bean is updated within the same transaction, then a lock upgrade (to 'Exclusive') occurs.

If this scenario occurs on multiple threads concurrently, then a deadlock can happen. This is because multiple 'Read' locks can be obtained concurrently, but one 'Exclusive' lock can be obtained only when all other locks have been dropped. Because all transactions are attempting the lock upgrade in this scenario, this one 'Exclusive' lock can never be obtained .

To avoid this problem, you can change the access intent policy for the entity bean from the default of wsPessimisticUpdate-WeakestLockAtLoad to wsPessimisticUpdate. This change in access intent enables the application to inform WebSphere and the database that the transaction will update the enterprise bean, and so an 'Update' lock is obtained immediately on the findByPrimaryKey. This avoids the lock upgrade when the update is performed later.

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 is deprecated in Version 6.0. (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, to handle rollbacks that could occur. Optimistic locking is really 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 ″Default Access Intent for Entities 2.x (Bean Level)″ on the Access tab of the EJB Deployment Descriptor, as described in "Applying access intent policies to beans" on page 187.

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-compliant entity beans that are packaged in EJB 2.x-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.

Access intent for both entity bean types

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.

Applying access intent policies to beans

You can apply an access intent policy to an application's entity beans through the assembly tool.

Note: This is the preferred technique to define access intent policies. Method-level access intent is deprecated in Version 6.0.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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 the assembly tools

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.

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 the Application Server Toolkit.

1. Start the Application Server Toolkit.

2. 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.

3. Select the **Access** tab. The Add Access Intent window appears. There are two areas of the panel that deal with adding access intent:

   - Default Access Intent for Entities 2.x (Bean Level)

   - Access Intent for Entities 2.x (Method Level)

4. Select the Bean or Method level. Another access intent window appears where you can set the properties you wish to use.

5. Use the dropdown list to select the Access intent name.

6. **Optional:** Enter a description.

7. Check the **Persistence Option** box.

8. Check the **Verify Read Only Data** box.

9. Use the dropdown list to select your choice for read-read consistency checking. You have three options:

   NONE No read-read checking is done.

   AT_TRAN_BEGIN
   During ejbLoad, if the data is from cache, check the database to ensure that the data of the bean (with proper locking based on access intent's concurrency control attribute) has not changed since the last load.

   AT_TRAN_END
   At the end of transaction, if the bean is not changed and did not load by the current transaction, check the database to ensure that the data of the bean has not changed from last load (with proper locking based on access intent's concurrency control attribute.) If the data has changed, fail the transaction.

10. Select **Finish**.

Examples: 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 EJB CMP application (to calculate the interest) and other applications, such as EJB BMP, 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 more 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.

Generally 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 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 WebSphere Application Server is to manage the details of persistence without having to explicitly manage any of the persistence logic from within the application.

There are situations, however, in which you might need to develop BMP entity beans. Because 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 WebSphere Application Server 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 design considerations

Use the access intent service to solve clear performance problems. Identify usage patterns that lead to poor application performance and apply appropriate access intent policies.

best-practices: 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 confusingly 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 to, 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. Finally, the collection increment and resource manager prefetch increment are only used on multi-object finders. Configuring policies on methods that will 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.

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.

Note: Method-level access intent is deprecated in Version 6.0.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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 BMP entity bean methods.

1. Look up the current access intent in the namespace. For example:

```
InitialContext ic = new InitialContext();
AccessIntent ai = ic.lookup("java:comp/websphere/AppProfile/AccessIntent");
```

2. Call the necessary get() methods. 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();
      // . . .
}
// . . .
```

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 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 BMP entity beans.

The following JNDI lookup allows BMP entity beans to access the AccessIntent interface:

```
java:comp/websphere/AppProfile/AccessIntent
```

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 best practices

When applying access intent policies to Enterprise JavaBeans (EJB) methods, consider the following issues.
- **Start by configuring the default access intent policy for an entity.** After your application is built and running, you can more finely 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 `wsPessimisticUpdate-NoCollision`.** 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 will attempt to update persistent store at any given time.

Frequently asked questions: Access intent

The following frequently asked questions involving access intent are answered.

I have not applied any access intent policies at all. My application runs just fine 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)*. Why?

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.

I am calling a finder method and I get an InconsistentAccessIntentException at run time. Why?

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.

I have two beans in a container-managed relationship. I call findByPrimaryKey() on the first bean and then call getBean2( ), a CMR accessor method, on the returned instance. At that point, I get an InconsistentAccessIntentException. Why?

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.

I have a bean with a one-to-many relationship to a second bean. The first bean has a pessimistic-update intent policy applied. When I try to add an instance of the second bean to the first bean's collection, I get an UpdateCannotProceedWithIntegrityException. Why?

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 2 Platform, Enterprise Edition (J2EE) applications.

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.

Assemble an Enterprise JavaBeans (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 such as the Application Server Toolkit (AST) or Rational Application Developer 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. This topic points you to AST documentation. The Application Server Toolkit information center accompanies this WebSphere Application Server information center.

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

An EJB module is migrated or created, reflecting the J2EE folder structure that specifies the location of enterprise bean content files, class files, class paths, the deployment descriptor, and supporting metadata. Files for the EJB module are shown in the Project Explorer view under **Enterprise Applications** and **EJB Projects**.

After you finish assembling your EJB module, you are ready to deploy your module.

You can generate EJB deployment code and deploy the module to a target server in one step. In the Project Explorer view, right-click on the EJB project and click **Deploy**.

Container transactions

Container transaction properties specify how an EJB container is to manage transaction scopes for the enterprise bean's method invocations. A transaction attribute is mapped to one or more methods.

Defining container transactions for EJB modules:

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 1.1 and EJB 2.x specifications.

To add a container transaction to an enterprise bean:

1. In the Project Explorer view of the J2EE perspective, right-click the Deployment Descriptor for your EJB project and select **Open With** → **Deployment Descriptor Editor** to open the deployment descriptor editor.
2. On the **Assembly** page of the editor, click **Add** in the Container Transactions section.
3. Select one or more enterprise beans from the list of beans found.
4. Select a container transaction type from the following choices:
   - **NotSupported -** Directs the container to invoke bean methods without a transaction context. If a client calls a bean method from within a transaction context, the container suspends the association between the transaction 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 transaction context is not passed to any enterprise bean objects or resources that are used by this bean method.
   - **Supports -** Directs the container to invoke the bean method within a transaction context if the client calls the bean method within a transaction . If the client calls the bean method without a transaction context, the container calls the bean method without a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.
   - **Required -** Directs the container to invoke the bean method within a transaction context. If a client calls a bean method from within a transaction context, the container calls the bean method within the client transaction context. If a client calls a bean method outside a transaction context, the container creates a new transaction context and calls the bean method from within that context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.
   - **RequiresNew -** Directs the container to always invoke the bean method within a new transaction context, regardless of whether the client calls the method within or outside a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.
   - **Mandatory -** Directs the container to always invoke the bean method within the transaction context associated with the client. If the client attempts to invoke the bean method without a transaction context, the container throws the javax.jts.TransactiononRequiredException exception to the client. The transaction context is passed to any EJB object or resource accessed by an enterprise bean method. EJB clients that access these entity beans must do so within an existing transaction . For other enterprise beans, the enterprise bean or bean method must implement the Bean Managed value or use the Required or Requires New value. For non-enterprise bean EJB clients, the client must invoke a transaction by using the javax.transaction.UserTransaction interface.
   - **Never -** Directs the container to invoke bean methods without a transaction context. If the client calls a bean method from within a transaction context, the container throws the java.rmi.RemoteException exception. If the client calls a bean method from outside a transaction context, the container behaves in the same way as if the Not Supported transaction attribute was set. The client must call the method without a transaction context
5. Select one or more methods elements from the list.
6. Click **Finish**.

The container transaction is added and displayed in the Container Transactions section, where the container transactions are listed by container transaction type.

After you define container transactions, you can use the deployment descriptor editor to work with them. Information about the editor can be found in the WebSphere Application Server Express documentation.
- To edit a container transaction, select it from the Container Transactions list and click **Edit**.

- To delete a container transaction, select from the list and click **Remove**.
- To take multiple container transactions that are the same container transaction type and combine them into a single container transaction definition, click **Combine**.

Method extensions

Method extensions are IBM extensions to the standard deployment descriptors for enterprise beans.

Method extension properties are used to define transaction isolation levels for methods, to control the delegation of a principal's credentials, and to define custom finder methods.

Method permissions

A method permission is a mapping between one or more security roles and one or more methods that a member of the role can call.

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 DataSource references are declared in the java:comp/env/jdbc subcontext.
  - JMS connection factories are declared in the java:comp/env/jms subcontext.
  - JavaMail 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.

To view this administrative console page, click **Applications** > **Enterprise Applications** > *application_name* > **EJB references** .

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.

Reference binding:

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.

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

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.

To view this administrative console page, click **Applications** > **Enterprise Applications** > *application_name* > **EJB JNDI Names** .

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

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.

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 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.

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

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.

1. Start an assembly tool. Refer to *Starting WebSphere Application Server Toolkit* in the Application Server Toolkit documentation.
2. Open the J2EE perspective. Click **Window > Open perspective > J2EE**.
3. In a J2EE hierarchy view (**Window > Show view > J2EE hierarchy**), right-click the EJB module containing beans that require sequence grouping, and click **Open with > EJB deployment descriptor editor**. The EJB deployment descriptor editor for the module is displayed in a view.
4. Click the **Overview** tab.
5. In the **EJB CMP sequence groups** section, click **Add**. The **EJB CMP Sequence Group** wizard panel is displayed.
6. Type a name for your sequence group.
7. Type your group type designation in all capital letters: RI_INSERT or UPDATE_LOCK
8. In the **Available Beans** list, highlight the first bean that you want to place in the group. Click the arrow pointing toward the **Selected beans** list. The bean name is removed from the Available beans list and is displayed in the Selected beans list.
9. Repeat the previous step until you complete your sequence group. You must add each bean in the order that you want the persistence manager to handle it. In the case of delete operations for an RI_INSERT group, the persistence manager reverses the order that you designate and deletes the beans and their corresponding database rows accordingly. If you need to alter the sequence of your group, select a bean and click the arrow to move the bean one position vertically.
10. Save your changes to the deployment descriptor.
    a. Close the EJB deployment descriptor editor.
    b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.

    You also can save changes to deployment descriptors at any time by pressing Ctrl+S.

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 EJB module, you install that module on a server that has been configured to support deployed modules.

Assemble one or more EJB modules, assemble one or more Web modules, and assemble them into a J2EE application.
1. Prepare the deployment environment.
2. Update the configuration for each EJB module as needed for the deployment environment. See the AST information center for more information about modifying deployment descriptors.
3. Deploy the application.

If you specify that EJB deploy be run during application installation and the installation fails with a NameNotFoundException message, ensure that the input JAR or 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.

Troubleshooting tips for EJBDEPLOY relationships
This article provides troubleshooting information for EJBDEPLOY problems.

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 > Enterprise Applications >** *applicationName* **> Manage Modules >** *moduleName*.

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

EJB deployment tool
Before you can successfully run your enterprise beans on either a test or production server, you need to generate deployment code for the enterprise beans. The EJB deployment tool provides a command-line interface that you can use to generate enterprise bean deployment code. The tool employs this command-line environment that enables you to run a build process overnight and have the deployment tool automatically invoked to generate your deployment code in batch mode.

The EJB deployment tool is invoked from the command line using the ejbdeploy command, which accepts an input EJB JAR or EAR file that contains one or more enterprise beans. It then generates an output, deployed JAR file or EAR file that contains deployment code in the form of .class files.

For a complete description of all of the options available to the ejbdeploy, see the related reference The ejbdeploy command.

The EJB deployment tool supports EJB single and multiple table inheritance. It supports the use of converters, which translate a database representation to a Java™ object type, and composers, which are used to map a single, complex bean field to multiple database columns. The EJB deployment tool supports the following levels of access intent (where AccessIntent is a WebSphere® extension):

1. wsPessimisticUpdateWeakestLockAtLoad
2. wsPessimisticUpdate
3. wsPessimisticUpdate-NoCollision
4. wsPessimisticUpdate-Exclusive
5. wsPessimisticRead
6. wsOptimisticUpdate
7. wsOptimisticRead

For more information on these access intents, see the related topic *Access intent and isolation level*.

In addition to these values, an access intent can also contain an optional *read ahead* hint.

Note: The *read ahead* hint indicates how deeply to read ahead in an EJB relationship graph.

2.0 EJB projects only: Mapping to multiple back-end databases is also supported. The schemas and the generated data definition language (DDL) file are stored in the following directory of the JAR or EAR file:

META-INF\backends*backend_id**databasename*.dbm

META-INF\backends*backend_id*\Table.ddl

If multiple backends exist and you did not set the current back-end ID in the EJB deployment descriptor, the EJB deployment tool will default to the first back-end ID that appears as a folder in the respective META-INF\backends directory as described earlier. If you map to a single backend database, then the generated DDL file will appear both in the directory as described above and also in the META-INF folder.

You can perform the following tasks with the EJB deployment tool:
- Generating deployment code for enterprise beans from the command line
- Implementing finders for CMP entity beans

Also refer to Message format for EJB validation to understand the format used for messages generated by the EJB validator.

Generating deployment code for enterprise beans from the command line:

The EJB deployment tool provides a command-line interface that you can use to generate enterprise bean deployment code. Before you can successfully run your enterprise beans on either a test or production server, you need to generate deployment code for the enterprise beans.

You generate EJB deployment code by running the `ejbdeploy` command.

Running the EJB deployment tool from the command line:
1. Open a command prompt.
2. Type the following at the prompt:

> ▶ Windows ◀ `ejbdeploy in.ear tmp out.ear`

> **UNIX** > **z/OS** > **400** `ejbdeploy.sh in.ear tmp out.ear`

This generates a EAR file called `out.ear`.

The following activities occur when you run the `ejbdeploy` command:

1. Code is imported from the input JAR or EAR file.
2. A top-down mapping is created if one does not exist.
3. Deployment code is generated.
4. The deployment code is compiled.
5. Remote Method Invocation Compiler (RMIC) is run.
6. Code is exported to the output JAR or EAR file.

Note: For CMP entity beans, a data definition language (DDL) file is generated that can be used to create corresponding database tables that are mapped to CMP fields. The DDL file is contained within the META-INF\backends*backend_id* directory and entitled Table.ddl

The ejbdeploy command:

Before you can successfully run your enterprise beans on either a test or production server, you need to generate deployment code for the enterprise beans. This reference topic describes what is the syntax, expected behavior, and descriptions of each of the parameters for running the ejbdeploy command from a command line.

Syntax

Use the following command and the optional parameters, when the schema and map are provided in the input EAR or JAR file:

```
ejbdeploy input_EAR_name|input_JAR_name working_directory output_EAR_name|output_JAR_name
[-bindear "options"] [-cp classpath] [-codegen] [-debug] [-keep] [-ignoreErrors] [-quiet]
[-nowarn] [-noinform] [-rmic "options"][-trace] [-sqlj] [-outer] [-complianceLevel "1.4"|"5.0"]
```

Use the following command and the optional parameters, when the schema and map are not available in the input EAR or JAR file, and a top-down mapping approach is needed:

```
ejbdeploy input_EAR_name|input_JAR_name working_directory output_EAR_name|output_JAR_name
[-bindear "options"] [-cp classpath] [-codegen] [-dbname "name"] [-dbschema "name"] [-dbvendor
name] [-debug] [-keep] [-ignoreErrors] [-quiet] [-nowarn] [-noinform] [-rmic "options"][-trace]
[-sqlj][-OCCColumn] [-outer] [-complianceLevel "1.4"|"5.0"]
```

The -dbschema, -dbname, -dbvendor, and -OCCColumn options are only used when creating a database definition in the top-down mode of operation. The database information is then saved in the schema document in the JAR or EAR file, which means that the options do not need to be specified again. It also means that when a JAR or EAR is generated, the correct database must be defined at that point because it cannot be changed later.

Behaviour

If your input JAR or EAR file contains CMP beans, the EJB deployment tool looks for an existing schema and map to use when generating deployment code. If no existing schema and map are found, a schema and map are created using top-down mapping rules.

In the top-down mapping approach, you already have existing enterprise beans and their design determines the database design. The generated schema contains one table for each CMP entity bean. In these tables, each column corresponds to a CMP field of the enterprise bean, and the generated mapping maps the field to the column.

If the -dbvendor option is not set, the default database backend is **DB2UDB_V82**. If you want to set a different database backend, use the -dbname, -dbschema, and -dbvendor options to specify your choice. A data definition language (DDL) file, Table.ddl, is created for the database backend set in the -dbvendor option, when you run the ejbdeploy command. However, you can specify only one backend at a time using the -dbvendor option.

If the -dbvendor option is specified for mapped jars, for example the JAR file already contains a DB2® backend and you specify -dbvendor ORACLE on the command line; in previous releases of the product, rather than getting a second backend, the database vendor specification was ignored. Starting in WebSphere Application Server v6.0.2, the following changes were made for the scenario where the -dbvendor option is specified for a mapped jar:

For 2.x CMP beans where multiple mappings to different database vendors are supported:
- If the value for the -dbvendor option is different from the existing maps, then a new top-down map is generated, and that becomes the current backend.
- If the value for the -dbvendor option is the same as one of the existing maps, then that map becomes the current backend, and the following message is issued:

  ```
  A mapping to the database vendor, database_vendor, already exists. Setting the current
  backend id to backend_id.
  ```

For 1.1 CMP beans that can only be mapped once:
- If the value for the -dbvendor option is the same as the existing map, then the following message is issued and deployment continues:

  ```
  A mapping to the database vendor, database_vendor, already exists.  Using the existing map to continue
  ```

- If the value for the -dbvendor option is different as the existing map, the following exception is thrown and deployment stops:

  ```
  A mapping already exists for a different database vendor.
  Action: If you want to generate deployment code against this existing map, for the -dbvendor argument
  ```

Another general behavior of the ejbdeploy command is if the abstract fields or bean name for CMP entity beans use any SQL reserved keywords, the top-down mapping adds a numeric suffix to the column name when generating the data definition language file (Table.ddl). This is to avoid SQL command conflicts when SQL reserved words are used as column names. For a list of SQL reserved words, see the topic SQL reserved keywords.

Parameters

ejbdeploy
    The command to generate deployment code. If run without any arguments, the `ejbdeploy` command displays a list of arguments that can be run with the command.

input_JAR_name **or** *input_EAR_name*
    The fully qualified name of the input JAR or EAR file that contains the enterprise beans for which you want to generate deployment code; for example, c:\ejb\inputJARs\myEJBs.jar. (This argument is required.)

    The ejbdeploy command no longer uses what is specified on the system class path. Instead, the dependent classes need to be contained in a JAR file or included in the command processing using the -cp option. You must ensure that the .class files of each enterprise bean's home and remote classes are packaged in the input JAR or EAR file.

You should not include source files in the input JAR or EAR file. If there are source files in the input JAR or EAR file, the EJBDeploy tools runs a rebuild before generating the deployment code. **Recommendation**: Either remove the source files, or include all dependent classes and resource files on the class path. Otherwise, this might cause problems during rebuild of your application on the server.

working_directory
> The name of the directory where temporary files that are required for code generation are stored. (This argument is required.) If the working directory that you specify already exists prior to running the `ejbdeploy` command, the temporary files are generated into the working directory (as an Eclipse workspace). However, if the working directory does not already exist prior to running the command, the directory is created and the Eclipse workspace is generated into it. In both cases, the workspace and all of its files are automatically removed when the deployment code generation is complete unless you specify the `-keep` option. (Retaining the workspace is useful for problem determination.)

output_JAR_name **or** *output_EAR_name*
> The fully qualified name of the output JAR or EAR file that is created by the ejbdeploy command and that contains the generated classes required for deployment; for example: c:\ejb\outputJARs\ myEJBs.jar. (This argument is required.) The directories specified in the fully-qualified name must already exist before you run the ejbdeploy command. (Note that when you specify a name for the output JAR or EAR file and then run the ejbdeploy command, any existing output JAR or EAR file of the same name will be overwritten without warning.)

-cp *classpath*
> If you intend to run the ejbdeploy command against JAR or EAR files that have dependencies on other zipped or JAR files, you can use the -cp option to specify the class path of the other JAR or zipped files. Using the -cp option, you can specify multiple zipped and JAR files as arguments. However, the zipped and JAR file names must be fully qualified, separated by semicolons, and enclosed in double quotation marks. For example: `-cp "path\myJar1.jar;path\myJar2.jar; path\myJar3.jar"`

> **Tip:** If you specfied the -sqlj option, you need to specify the location of the SQLJ translator classes, sqlj.zip. The default path for this file is *x*:\java, where *x* is the installation directory of DB2, for example, d:\sqllib\java\sqlj.zip on Windows®.

-codegen
> Restricts the ejbdeploy command to just (a) importing code from the input JAR or EAR file (b) generating the deployment code, and (c) exporting code to the output JAR or EAR file. It will not compile the generated deployment code or run remote method invocation compiler (RMIC). Since Java source code is not usually exported in the output EAR or JAR, this is the only way to save the generated code.

-bindear *"options"*
> Enables you to populate an EAR file with bindings. This argument applies only to EAR files. You can also use this command without specifying any options. The options must be separated by a space and enclosed in double quotation marks. For example: `-bindear "xx yy zz"` For more information on these options, see the WebSphere Application Server documentation.

-dbname *"name"*
> The name of the database you want to define in the data definition language (DDL) file that gets generated. If the name of the database contains any spaces, the entire name must be enclosed in double quotes. For example: `-dbname "my database"`

-dbschema *"name"*
> The name of the schema you want to create. If the name of the schema contains any spaces, the entire name must be enclosed in double quotes. For example: `-dbschema "my schema"`

-dbvendor *name*
> The name of the database vendor, which is used to determine database column types, mapping information, Table.ddl, and other information. The valid database vendor names are:

DB2UDB_V81
DB2 Universal database V8.1 for Linux®, UNIX®, and Windows

DB2UDB_V82
DB2 Universal database V8.2 for Linux, UNIX, and Windows

DB2UDBOS390_V7
DB2 Universal Database™ for z/OS®, V7

DB2UDBOS390_V8
DB2 Universal Database for z/OS, V8

DB2UDBOS390_NEWFN_V8
DB2 Universal Database for z/OS, V8

Additional to the DB2UDBOS390_V8 option, this option includes the generated data model that has all the new catalog features of DB2 Universal Database for z/OS v8 specified in the new function mode. Use this option if you plan to work with the generated data model available in the WebSphere Application Server Toolkit or IBM® Rational® Software Development Platform products.

DB2UDBISERIES_V53
DB2 Universal Database for iSeries™, V5R3

DB2UDBISERIES_V54
DB2 Universal Database for iSeries, V5R4

DERBY_V10
IBM Cloudscape™, V10.1

ORACLE_V9I
Oracle, V9i

ORACLE_V10G
Oracle, V10g

INFORMIX_V93
Informix® Dynamic Server, V9.3

INFORMIX_V94
Informix Dynamic Server, V9.4

INFORMIX_V100
Informix Dynamic Server, V10.0

SYBASE_V1250
Sybase Adaptive Server Enterprise, V12.5

SYBASE_V15
Sybase Adaptive Server Enterprise, V15.0

MSSQLSERVER_2000
Microsoft® SQL Server 2000

MSSQLSERVER_2005
Microsoft SQL Server 2005

The following backend ids are deprecated:

SQL92 (1992 SQL Standard)

SQL99 (1999 SQL Standard)

Although SQL92 and SQL99 are deprecated, the SQL92 and SQL99 options remain available. If you choose to use the deprecated SQL92 or SQL99 backend id, see the topic EJB query to SQL syntax to help determine what backend you should use, in the near future, when the deprecated SQL92 and SQL99 backends are no longer available.

If you want to use an unsupported database, see the topic EJB query to SQL syntax to help choose a valid database vendor backend id that matches closely to your unsupported deployment environment.

Note:
- The default is **DB2UDB_V82** (DB2 for Windows, V8.2 and UNIX)
- If -sqlj is specified, it supports **DB2UDB_V82** (DB2 for Windows, V8.2 and UNIX), **DB2UDB_V81** (DB2 for Windows, V8.1 and UNIX), **DB2UDBOS390_V8** (DB2 for z/OS, V8) and **DB2UDBOS390_V7**(DB2 for z/OS, V7).

-debug
Specifies that deployment code will be compiled with debug information.

-keep
Controls the disposition of the temporary files that are created (that is, the Eclipse workspace) when the `ejbdeploy` command has run. Without this option, the Eclipse workspace is deleted when the command has completed.

-ignoreErrors
Specifies that processing should continue even if validation errors are detected.

-quiet
During validation, suppresses status messages (but does not suppress error messages).

-nowarn
During validation, suppresses warning and informational messages.

-noinform
During validation, suppresses informational messages.

-rmic *"options"*
Enables you to pass RMIC options to RMIC. The options, which are described in Sun's RMI Tools documentation, must be separated by a space and enclosed in double quotation marks. For example: `-rmic "-nowarn -verbose"`

-trace
Generates additional progress messages to the console.

-sqlj

Note: This option is valid only on enterprise beans compliant with the 2.0 specification.

Enables you to use SQLJ instead of JDBC to make calls to a DB2 database. With the **-sqlj** option specified, the EJB deployment tool generates SQLJ code for your CMP beans to use SQLJ to access the database. It also automatically invokes the SQLJ translator to translate the SQLJ source files. Finally, an Ant script will be created by the EJB deployment tool to help you to customize the SQLJ profiles easily. You can run the Ant script against the profile to produce a DB2 package. These DB2 packages can be used at runtime to avoid extensive runtime checking. Once you have generated the deployment code for SQLJ using the EJB deployment tool, you will need to run the DB2 SQLJ profile customizer, `db2sqljcustomize`, against the generated .ser file, which is found in the subfolder of the *websphere_deploy* folder associated with the DB2 backend. Consult the DB2 documentation for more information on running the DB2 SQLJ profile customizer, or visit www7b.boulder.ibm.com/dmdd/zones/java/bigpicture.html (section *SQLJ support*).

-OCCColumn

Note: This option is valid only on EJB 2.x CMP entity beans when generating top-down mapping. Enables you to add a column to your relational database table for collision detection. The *collision detection column* is the additional database column reserved to determine if a record has been updated. Adding a column for collision detection is an alternative optimistic concurrency control scheme of including attributes in a predicate for optimistic access intents. To manage the collision

detection column, you will need to provide your own database trigger implementation. The following are the result of adding a column for collision detection:

- The data type of the collision detection column is a 64 bit integer.
- The naming convention of the collision detection column has the following format: **OCC**_*bean_name*
- The top-down mapping generates an extra relational column. This column can not be mapped to the enterprise bean.

-outer
> This is an optional parameter and is only supported for deploying J2EE 1.3 applications. It specifies to use `OUTER` semantics for path expressions in EJB query language queries. If this parameter is not specified, the default setting is `INNER` semantics.

> **Note:** If you specify this parameter for deploying a J2EE 1.4 application, this option is ignored because the specification for J2EE 1.4 defines the `INNER` semantics be used for J2EE 1.4 applications.

-complianceLevel ″**1.4** ″ | ″**5.0**″
> Specify the Java Development Kit (JDK) compiler compliance level to either 1.4 or 5.0, if you have included application source files for compilation. If this parameter is not specified, the default setting is JDK v1.4. If your application is using new functionality defined in JDK v5.0 or you have included source files (which is not recommended) then you must specify the parameter value as ″5.0″.

Example

ejbdeploy AceEmp.ear d:\deploydir AceEmp_sqlj.ear -dbvendor DB2UDB_V82 -keep -sqlj -cp "e:\sqllib\java\ssqlj.zip"

Explanation:

We have DB2 Universal database (version 8.2 for Windows and UNIX) installed in e:\sqllib.

The ejbdeploy command takes the AceEmp.ear file (which has enterprise beans that are compliant with the EJB 2.0 specification) as input and produces the AceEmp_sqlj.ear as output. Since the -sqlj option is used, SQLJ is used instead of JDBC in the generated code to make calls to DB2.

When ejbdeploy runs, it creates an Eclipse workspace in the directory that you specify as the working directory: d:*deploydir*. When it has completed running, it deletes this workspace. However, the -keep option causes ejbdeploy to end without deleting the workspace.

Implementing query methods in home interfaces for CMP entity beans:

EJB 2.x provides a query syntax called EJB QL for both finder and select methods of CMP entity beans.

Finder methods obtain one or more entity bean instances from a database, and are defined in the home interface. Select methods are defined on the abstract bean class and can return entity beans (any entity bean type defined in the EJB JAR file) or CMP field values.

The <query> element is used to define the query for the finder method in the deployment descriptor, for every finder method except findByPrimaryKey(key). Queries specified in the deployment descriptor are translated into SQL during deployment. The query statement is contained in the <ejb-ql> element of the <query> element:

```
<query>
<query-method>
<method-name>findAll</method-name>
<method-params>
<method-param></method-param>
</method-params>
```

```
</query-method>
<result-type-mapping></result-type-mapping>
<ejb-ql>select object(o) from Employee o</ejb-ql>
</query>
```

Where to find additional information about the EJB query language

Detailed information on how to structure EJB queries is found in Chapter 11 of the EJB 2.x specification available at java.sun.com/products/ejb/docs.html. However, the WebSphere documentation contains Sun's information along with the WebSphere extensions.

Client applications

Using application clients

An application client module is a Java Archive (JAR) file that contains a client for accessing a Java application.

Complete the following steps for developing different types of application clients.

1. Decide on a type of application client.
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.

   Deploy the application client on Windows systems.
5. Run the application client.

View the Application Clients Samples Gallery for more information. To access these samples, install Application Clients, and retrieve the samples from your local file system as the following command indicates:

`<app_server_root>/samples/index.html`

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 Version 6.1 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 6 (Application Client) consists of the following client applications:
- J2EE application client application (Uses services provided by the J2EE Client Container)
- Thin application client application (Does not use services provided by the J2EE Client Container)
- Applet application client application
- ActiveX to EJB Bridge application client application (Windows only)

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 J2EE application client applications or Thin application client applications
- ▶ Windows ◀ An ActiveX to EJB Bridge run time for ActiveX to EJB Bridge application client applications (Windows only)
- ▶ Windows ◀ 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 *ActiveX application client* model, uses the Java Native Interface (JNI) architecture to programmatically access the Java virtual machine (JVM) API. Therefore the JVM code exists in the same process space as the ActiveX application (Visual Basic, VBScript, or Active Server Pages (ASP) files) and remains attached to the process until that process terminates.

In the *Applet client* model, a Java applet embeds 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 *J2EE application client* is a Java application program that accesses enterprise beans, Java DataBase Connectivity (JDBC) APIs, and Java Message Service message queues. The J2EE application client program runs on client machines. This program follows the same Java programming model as other Java programs; however, the J2EE 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 a Java client application exists but the application needs enhancements to use enterprise beans, or where the client application requires a thinner, more lightweight environment than the one offered by the J2EE 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 uses the Sun Java Development Kit, and the Thin application client uses the IBM Developer Kit for the Java platform.

The J2EE application client programming model provides the benefits of the J2EE platform for the Java client application. Use the J2EE application client to 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 J2EE platform, you can put the client application code from one J2EE platform implementation to another. The client application package can require redeployment using each J2EE 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 J2EE 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 J2EE 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 J2EE 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.

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	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 such as the Application Server Toolkit (AST) or Rational Application Developer. 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.

J2EE application clients:

The J2EE application client programming model provides the benefits of the Java 2 Platform for WebSphere Application Server Enterprise product.

The J2EE 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 J2EE platform, you can put the client application code from one J2EE platform implementation to another. The client application package can require redeployment using each J2EE platform deployment tool, but the code that comprises the client application does not change.

The J2EE application client run time supplies a container that provides access to system services for the application client code. The J2EE application client code must contain a main method. The J2EE 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 J2EE platform. These deployment descriptors identify enterprise beans or local resources (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.

The J2EE 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 J2EE application client run time also provides support for security authentication to the enterprise beans and local resources.

The J2EE 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 J2EE 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 J2EE 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.

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.

Note: The Pluggable application client is only available on the Windows platform.

This client is designed to support those users who want a lightweight Java client application programming environment, without the overhead of the J2EE 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 J2EE 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 J2EE application client. However, you cannot access the services in the Pluggable application client as easily as you can in the J2EE application client. The J2EE 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 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 J2EE 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 J2EE application client accesses a logical name from the `java:/comp` name space. The J2EE 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 J2EE 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 (security, Workload Management (WLM), and others). 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 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 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 Version 6.1 supports the pluggable client.

The thin client is designed to support those users who want a lightweight Java client application programming environment, without the overhead of the J2EE 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 J2EE 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 J2EE application client. However, you cannot access the services in the thin client as easily as you can in the J2EE application client. The J2EE 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 J2EE 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 Java 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 J2EE application client accesses a logical name from the `java:/comp` name space. The J2EE client run time resolves that name to the physical location and returns the reference to the client application. The thin 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 J2EE 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 (security, Workload Management (WLM), and others). 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.

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 the Application Server Toolkit or the Rational Web Developer 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. 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 Application Client Resource Configuration 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, usually a colon (on operating systems such as AIX or Linux) or a semi-colon (on Windows systems).
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.

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 cannot be found in the path specified on the launchClient command.
Recommended response	Verify that the path and file name specified on the launchclient command are correct. If you are running on the Windows operating system and the path and file name are correct, use a short version of the path and file name (8 character file name and 3 character extension).

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.

The applet client application client fails to launch an HTML browser in Internet Explorer

Explanation

Applet client applications run only on Windows systems. When the applet client application runs, the application output data is displayed in a browser window. If you are using Internet Explorer with the Windows XP operating system for Service Pack 2 , then you might get errors when trying to display output data.

Possible causes

The Windows XP operating system for Service Pack 2 has a security feature that blocks pop-up browser windows from appearing.

Recommended response

- Locate the information bar found under the URL Address bar in the Internet Explorer pop-up browser that has been blocked.
- Click the Information Bar to display options that disable the operating system security feature.
- Select **Allow blocked content**. You are prompted with a security window asking you to confirm your selection to allow blocked content.
- Click **Yes**.
- The applet client application runs successfully, and the browser information is displayed appropriately.

Installing the Developer Kit feature downgrades the JRE files from Version 6.0.1 or Version 6.0.2 to Version 6.0

Explanation

If you select the Developer Kit feature on the Application Client Version 6.0.0 installer, all the files are installed under the *<client_install_root>*/java directory, instead of leaving the Java Runtime Environment (JRE) files intact. The JRE files are unexpectedly downgraded to the Version 6.0.0 level.

Possible causes

Selecting the **Developer Kit** feature on the Application Client 6.0.0 installer will actually install all files under the *<client_install_root>*/java directory rather than leave the JRE files intact. Therefore, the JRE files are unexpected downgraded to the 6.0.0 level in the above installation scenario.

Recommended response

Complete the following installation steps to prevent unexpected downgrading of the JRE files.

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/was/support/

Developing application clients

This topic provides the steps for programming application clients to access resource objects defined on the server.

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, such as the Application Server Toolkit (AST) or Rational Application Developer. 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.

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.

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.

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.

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.

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.

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.

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.

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.

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 7. 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	

Table 7. ActiveX to Java primitive data type conversion (continued)

Visual Basic Type	Variant Type	Java Type	Notes
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.

best-practices: 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.

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.

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:

```
-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>\lib\WMQ\java\lib"
-Dcom.ibm.CORBA.ConfigURL=file:<app_client_root>\properties\sas.client.props
-Dcom.ibm.SSL.ConfigURL=file:<app_client_root>\properties\ssl.client.props
-classpath <app_client_root>\properties
```

   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.

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";
```

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.

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 highlighted third line to your code. That highlighted line 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 J2EE application client code

This topic provides the steps required to develop J2EE application client code.

A *J2EE application client* program operates similarly to a standard J2EE program in that it runs its own Java virtual machine (JVM) code and is invoked at its main method.

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 J2EE application client program on any development machine. At this stage, you do not require access to the WebSphere Application Server.

   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` 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 such as the Application Server Toolkit (AST) or Rational Application Developer.

   The JNDI namespace knows what to return on a lookup because of the information assembled by the assembly tool.

   Assemble the J2EE 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 the Application Server Toolkit (AST) or Rational Application Developer 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 J2EE resource. The J2EE 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.

After developing the J2EE application client code, launch the application client.

J2EE application client class loading

When you run your J2EE 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 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 J2EE 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 J2EE 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 J2EE 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 J2EE 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 such as the Application Server Toolkit (AST) or Rational Application Developer. 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 the Application Server Toolkit (AST) or Rational Web Developer, 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, 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 JAR file.

Assemble a client module to contain application client code. Group enterprise beans, Web components, and resource adapter code in separate modules.

Use an Application Server Toolkit (AST) or Rational Application Developer 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 J2EE modules. Ensure that **J2EE** 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.

After you finish assembling all of your application's modules, you are ready to deploy your application.

▶ Windows   To deploy your application on Windows, refer to "Deploying J2EE application clients on workstation platforms" on page 249.

For more information, see the online help for the assembly tool. Similar information is in the **Application Server Toolkit** information center available with this information center. Click **Application Server Toolkit > J2EE applications >** *application_clients_topic*.

Developing Pluggable application client code

This topic provides steps to install and use the Pluggable application client.

WebSphere Application Server Version 6.1 supports the pluggable client.

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** shell, 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.

```
"%JAVA_HOME%\bin\java" %WAS_LOGGING% -Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"
-classpath "%WAS_CLASSPATH%;<list of your application jars and classes>
-Djava.ext.dirs="%JAVA_JRE%\lib\ext;%WAS_EXT_DIRS%;%WAS_HOME%\plugins;%WAS_HOME%\lib\WMQ\java\lib"
-Djava.naming.provider.url=iiop://<your WebSphere server machine name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
-Djava.endorsed.dirs="%WAS_ENDORSED_DIRS%"
%SERVER_ROOT% %CLIENTSAS% %CLIENTSSL% <fully qualified class name to run>
```

View the Samples Gallery for more information about the Application Client.

Developing Thin application client code

You can develop and run Java thin client applications on machines installed with either a client or a server. The client provides a setup command shell which sets up your environment for either a thin client application or a J2EE client application. The server provides a command shell which sets up your environment for J2EE application clients only.

Both J2EE application clients and thin application clients can access JMS resources provided by the default messaging provider.

WebSphere Application Server Version 6.1 supports the pluggable client.

Note: Thin clients are not packaged with JDBC provider classes. For example, the WebSphere Application Server Version 6.1 thin client is not packaged with Cloudscape version 10.1 classes. Likewise, the version 6.02 thin client is not packaged with Cloudscape Version 5.1 or Cloudscape Version 10.0 classes. Therefore, to utilize the JDBC provider classes (such as Cloudscape, Oracle, DB2, Informix, or Sybase) on a thin client, you must:

1. Add the classes to your thin client environment.
2. Make the classes visible to the thin client application. 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 client application 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 Java application thin client by selecting option **J2EE 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.

View the Samples gallery for more information about the Application Client.

Developing Thin application client code on a client machine

This topic provides the steps necessary to develop Thin application client code on a client machine.

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 the Java application thin client environment.

   Use the setupClient shell.

   ► Windows

   ```
   app_client_root\AppClient\bin\setupClient.bat
   ```

2. Compile your client application.

   ► Windows   Run the following 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
   ```

3. Invoke your client application.

   Run the following Java command: ► Windows

   ```
   "%JAVA_HOME%\bin\java" %WAS_LOGGING% -Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"
   -classpath "%WAS_CLASSPATH%;<list of your application jars and classes>"
   -Djava.ext.dirs="%JAVA_JRE%\lib\ext;%WAS_EXT_DIRS%;%WAS_HOME%\plugins;%WAS_HOME%\lib\WMQ\java\lib"
   -Djava.naming.provider.url=iiop://<your WebSphere server machine name>
   -Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
   "%SERVER_ROOT%" "%CLIENTSAS%" "%CLISENTSSL%" <fully qualified class name to run>
   ```

   For more information on IIOP and `corbaloc` URLs, see Developing applications that use JNDI.

View the Samples gallery for more information about the Application Client.

Developing Thin application client code on a server machine

This topic provides the steps necessary to develop Thin application client code on a server machine.

You must install WebSphere Application Server before performing this task.

1. Set up the Thin application client environment.

   Use the **setupCmdLine** shell, located in: ► Windows

   ```
   profile_root\bin\setupCmdLine.bat
   ```

2. Compile your client application.

   ► Windows

   ```
   "%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.
   - Use Java code to call your main class directly:

     ► Windows

```
"%JAVA_HOME%\bin\java" %WAS_LOGGING%"
-Djava.security.auth.login.config="%WAS_HOME%\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
WebSphere server machine name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
-Dserver.root="%WAS_HOME%" "%CLIENTSAS%" "%CLIENTSSL%" %USER_INSTALL_PROP%
-classpath "%WAS_CLASSPATH%;<list of your application jars and classes>"
<fully qualified class name to run><your application parameters>
```
  - Use the WebSphere Application Server launcher.

    > Windows ◄ Enter:

```
"%JAVA_HOME%\bin\java" %WAS_LOGGING%
-Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"
-classpath "%WAS_CLASSPATH%;<list of your application jars and classes>
-Djava.ext.dirs="%WAS_EXT_DIRS%;%WAS_HOME%\plugins"
-Djava.endorsed.dirs="%WAS_ENDORSED_DIRS%
-Djava.naming.provider.url=iiop://<your WebSphere server machine name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
%SERVER_ROOT% %CLIENTSAS% %CLISENTSSL% <fully qualified class name to run>
```

For more information on IIOP and `corbaloc` URLs, see Developing applications that use JNDI.

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.

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.

Note: This task only applies to J2EE application clients. Only perform this task if you configured your J2EE application client to use resource references.

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.

On the Windows operating systems:

- clientRAR add c:\rars\myrar.rar
- clientRAR delete myrar.rar

On the UNIX operating systems:

- ./clientRAR add /usr/rars/myrar.rar
- ./clientRAR delete myrar.rar

Configuring new connection factories for resource adapters:

Use the Application Client Resource Configuration Tool (ACRCT) to configure new connection factories for resource adapters.

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:

252   Developing and deploying applications

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.

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.

Note: This task only applies to J2EE application clients.

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 for a Windows system or the `clientConfig.sh` file for a UNIX system.
3. Open an EAR file within the Application Client Resource Configuration Tool (ACRCT):
   - Click **File** > **Open**.
   - Select the file and click **Open**.
4. Save your changes to the file and close the tool:
   - Click **File** > **Save**.
   - 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, since 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 Supported hardware, software, and APIs Web site:

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) .

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.

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

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 JavaMail sessions and providers for your application clients using the Application Client Resource Configuration Tool (ACRCT).

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of JavaMail sessions and providers for your application clients to use.

1. Start the ACRCT.
2. Open an EAR file.
3. Locate the JavaMail objects in the tree that displays. For example, if your file contains JavaMail sessions, expand **Resources** > **application**.jar > **Mail Providers** > **java_mail_provider_instance** > **Mail Sessions**.

   In this example, **java_mail_provider_instance** is a particular JavaMail provider.

The JavaMail 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 JavaMail provider and JavaMail session settings for application clients:

You can configure JavaMail provider and JavaMail session settings. This topic provides the required fields, special cases, and an example.

The purpose of this article is to help you configure JavaMail provider and JavaMail session settings.
- Required fields:
  - JavaMail Provider Properties page: name, and at least one protocol provider
  - JavaMail Session Properties page: name, jndiName, mail transport protocol, mail store protocol
- Special cases:
  - The password is encrypted when using the ACRCT tool. Without the tool, you cannot encrypt this field.
- Example:

```
<resources.mail:MailProvider xmi:id="MailProvider_1" name="Default Mail Provider"
description="IBM JavaMail Implementation">
<classpath>mailProvider:classpath</classpath>
<factories xmi:type="resources.mail:MailSession" xmi:id="MailSession_1"
name="mailSession:name" jndiName="mailSession:jndiName"
description="mailSession:description" mailTransportHost="mailSession:mailTransportHost"
```

```
mailTransportUser="mailSession:mailTransportUser"
mailTransportPassword="{xor}Mj42Mww6LCw2MDFlMT4yOg=="
mailFrom="mailSession:mailFrom" mailStoreHost="mailSession:mailStoreHost"
mailStoreUser="mailSession:mailStoreUser"
mailStorePassword="{xor}Mj42Mww6LCw2MDFlMT4yOg==" debug="true"
mailTransportProtocol="ProtocolProvider_1" mailStoreProvider="ProtocolProvider_1">
<propertySet xmi:id="J2EEResourcePropertySet_1">
<resourceProperties xmi:id="J2EEResourceProperty_1"
name="mailSession:customName" value="mailSession:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_2">
<resourceProperties xmi:id="J2EEResourceProperty_2" name="mailProvider:customName"
value="mailProvider:customValue"/>
</propertySet>
<protocolProviders xmi:id="ProtocolProvider_1" protocol="smtp"
className="smtp:className"/>
<protocolProviders xmi:id="ProtocolProvider_2" protocol="pop3"
className="pop3:className"/>
<protocolProviders xmi:id="ProtocolProvider_3" protocol="imap"
className="imap:className"/>
</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.

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-4.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 Hyper Text 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).

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.

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).

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of URL providers and URLs to be used by your application clients.

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).

During this task, you create URLs for your client application.
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 J2EE 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 J2EE applications, as JMS clients, to exchange messages asynchronously with other JMS clients, by using JMS destinations (queues or topics). A J2EE application can use JMS queue destinations for point-to-point messaging and JMS topic destinations for Publisher and Subscriber messaging. A J2EE 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 J2EE 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 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 J2EE application, without the application having to explicitly poll JMS destinations.

Java Message Service (JMS) providers for clients

This topic describes the different ways that client applications can use JMS providers with WebSphere Application Server. A JMS provider enables use of the Java Message Service (JMS) and other message resources in 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*.

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.

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.

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 the WebSphere MQ *Using Java* book, located in the WebSphere MQ Family 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.

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-3.ibm.com/ software/ts/mqseries/library/manualsa/manuals/platspecific.html, 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 Java Message Service 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).

During this task, you create new resource environment provider configurations for your application client.

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).

During this task, you create new resource environment entries for your client application.

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 J2EE application clients using the Application Client Resource Configuration Tool (ACRCT).

Perform the following tasks after deploying application clients. This task only applies to J2EE application clients.

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).

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).

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).

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.

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).

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 J2EE application client resources using the Application Client Resource Configuration Tool (ACRCT).

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 J2EE application clients.

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.

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 article and read the related topics, before you start to use the installation tools. Specifically, read these topics before installing the product:

*   Installing silently
*   Best practices for installing

As a general rule, if you launch an installation and there is a problem such as not having enough temporary space or not having the right packages on your Linux or UNIX systems, then cancel the installation, and make the required changes. Restart the installation to initiate your changes.

You can install this product by a non-root user on a UNIX operating system and non-administrator user on a Windows operating system. However, the following functions are not enabled on Windows if the product is installed by a non-administrator user:

*   ActiveX to EJB bridge
*   Applet Client
*   Launching Java Web Start from browser

In Version 6.x, the Application Client for WebSphere Application Server is installable on a machine with a previous version of Application Clients. However, you cannot install a Version 6.x Application Client on top of a previous version of the Application Client. For example, if a Version 5 Application Clients install under the `C:\WebSphere\AppClient` directory, you can not choose the same install location for your V6.x

Note: For Application Clients to coexist, there is a limitation on Applet client and ActiveX client on Windows that can not be coexisted with V5.0.x and V4.x of the clients. For example, the Applet client feature in Version 6.x cannot coexist with the Applet client feature in any previous release. This coexistence is not available because the installation of Applet client feature in Version 6.x sets the browser default Java Virtual Machine (JVM) using the Java Runtime Environment (JRE) from the Version 6.x installation, which is Java Runtime Environment Version 1.4.2. Similarly, the ActiveX to EJB Bridge feature in Version 6.x sets the Windows system path to use the JRE from the Version 6.x installation.

1. Install Application Client for WebSphere Application Server using the launchpad.

   See Install Application Client for WebSphere Application Server using the launchpad.

   ▶ **Linux**   The launchpad program is available on the root directory of the product CD in the program, `launchpad.sh`.

   ▶ **Windows**   The launchpad program is available on the root directory of the product CD in the program `launchpad.bat`.

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 /install_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. Application Client V6.1 can be installed together with previously installed Application Clients if their versions are lower than 6.1. However, only one instance of Application Client V6.1 can be installed on the same system. When the installer is launched, it will detect any existing installation of Application Client V6.1 and direct the install flow to the feature panels so that additional features can be added on top of the existing installation.

b. As indicated in the previous example, you can start the installation wizard from the product CD-ROM, using the command line.

On other Linux platforms and UNIX-based platforms, run the `./install` command.

On Windows platforms, run the `Install.exe` command.

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 is displayed.

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 is displayed 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.

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 J2EE 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 J2EE/Thin Client and Custom Pluggable Client. For other platforms, there is only one custom installation type.

▶ Windows Custom J2EE/Thin Client

- If you select the **ActiveX to EJB Bridge** feature, then the following is displayed 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 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 J2EE/Thin Client installation type. The SDK component is optional; however, you must install the SDK component to compile the sample.

- Install the Administration Thin Client, if you need a runtime jar that is customed to enable client applications to perform WebSphere administration tasks. The Administration Thin Client is installed into *<install_root>/runtimes*.

- Install the Web Services Thin Client, if you need a runtime jar that is customed to enable client applications to communicate with the Application Server through Web Services. The Web Services Thin Client is installed into *<install_root>/runtimes*.

▶ Windows Custom Pluggable Client

- Install the Pluggable Client samples if you want to make use of the Pluggable Client applications.

▶ Linux Custom

- 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 J2EE/Thin Client installation type. The SDK component is optional; however, you must install the SDK component to compile the sample.

- Install the Administration Thin Client, if you need a runtime jar that is customized to enable client applications to perform WebSphere administration tasks. The Administration Thin Client is installed into *<install_root>/runtimes*.

- Install the Web Services Thin Client, if you need a runtime jar that is customed to enable client applications to communicate with the Application Server through Web Services. The Web Services Thin Client is installed into *<install_root>/runtimes*.

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 is displayed 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 `<install_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.

You successfully installed the Application Client for WebSphere Application Server and the features you selected.

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.

The following table offers tips for installing Application Client on multiple platforms.

Operating environment	Tip
Linux and UNIX systems	Spaces are not supported in the name of the installation directory on Linux and UNIX platforms.
UNIX systems	When the application client installations are successful, the return code **0** is issued from the UNIX shell where you issued the /install command. Other return codes include: • **1** -- Failure • **2** -- Partial success Any other return code indicates an unsuccessful installation.
Solaris systems	Double-byte character set (DBCS) characters are not supported in the name of the installation directory on Solaris systems.
All platforms	Reserve at least 4 to 5MB free space in the target platform temporary directory.

All platforms	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
``` |
| All platforms | After the installation, when changing the installation settings for the WebSphere Application Server host name and the port number, edit the `setupClient.bat` for Windows or `setupClient.sh` for UNIX. 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.bat` file or `setupClient.sh` file is located in the `bin` sub-directory under the Application Client installation destination. |

Installing Application Client for WebSphere Application Server silently

This topic provides the steps necessary to use the installation wizard and perform a silent installation.

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.

      Note: To prepare the file for a silent installation on AIX, use UNIX line-end characters (0x0D0A) to terminate each line of the options response file.
   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.

You installed application clients silently by using the response file.

To verify the silent install, look for the string `INSTCONFSUCCESS` in the log.txt file for successful installation and `INSTCONFFAILED` for a failed installation. For UNIX platforms, the install command returns a return code of **0** to indicate a successful installation, **1** to indicate failure and **2** to indicate partial success. Any other return code means that the installation failed.

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:

- *<system_temp_dir>*/niflogs
- *<user_home>*/cltlogs

Uninstalling Application Client for WebSphere Application Server

This task describes using the uninstall program to uninstall the Application Client for WebSphere Application Server.

If you want to uninstall IBM Application Client for WebSphere Application Server manually, see the topic, Manually uninstalling on a Windows system.

1. Stop any browsers and any Java processes related to Application Client products.

   See Uninstalling the product.

2. Change directories to the `uninstall` directory before issuing the command to uninstall the application client. The command file is located in the *install_root*/`uninstall` directory on a Linux or z/OS platform, and in the *install_root*\\`product`\\`uninstall` directory on a Windows system.

   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.

   Use the **uninstall** command.

   ▶ Linux   The command file is named `uninstall`.

   ▶ Windows   The command file is named `uninstall.exe`.

   ▶ Linux   On Linux and z/OS platforms, issue the **uninstall** command from the *install_root*/`uninstall` directory:

   `./uninstall`

   ▶ Windows   On Windows platforms, call the **uninstall.exe** command:

   install_root\\`uninstall`\\`uninstall.exe`

   ▶ Windows   Call the program directly from the *install_root*\\`uninstall` directory. For example, if the installation root is `C:\IBM\WebSphere\AppClient`, issue the following command:

   `C:\IBM\WebSphere\AppClient\uninstall> uninstall.exe`

   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.

Application Client for WebSphere Application Server is uninstalled.

Verify the uninstall procedure by viewing the *install_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
```

Running application clients

The J2EE specification requires support for a client container that runs stand-alone Java applications (known as J2EE application clients) and provides J2EE services to the applications. J2EE services include naming, security, and resource connections.

You are ready to run your application client using this tool after you have:
1. Written the application client program.
2. Assembled and installed an application module (`.ear` file) in the application server run time.
3. Deployed the application using the Application Client Resource Configuration Tool (ACRCT) on Windows .

This task only applies to J2EE application clients. To launch J2EE application clients using the launchClient script, perform the following steps:

1.

   On the CL command line, enter the following command to start the Qshell environment:

   `STRQSH`

   a. Enter the following command to launch J2EE 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.ear* 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.

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=c:\mydir\util.jar;c:\mydir\harness.jar;c:\production\G19
\global.jar BootstrapHost=abc.westcoast.mycompany.com tracefile=c:\WebSphere\mylog.txt
```

launchClient tool

This topic describes the Java 2 Platform Enterprise Edition (J2EE) 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.**

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.ear* 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. 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 J2EE 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. The SOAP connector does not prompt for authentication.

-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 J2EE application client
> * Resources defined in the J2EE application
> * Classes specified on the manifest of the J2EE 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
Prints out the Java portion of the Java Naming and Directory Interface (JNDI) name space for WebSphere Application Server. The true value uses the short format that prints out the binding name and the type of the object bound at that location. The long value uses the long format that prints out the binding name, bound object type, local object, type and string representation of the local object, for example, IORs and string values. The default value is false.

-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.

For more information on basic and advanced trace formatting, see Interpreting trace output.

-CCverbose

This option displays additional information messages. The default is `false`.

The following examples demonstrate correct syntax.

▶ Windows

```
launchClient c:\earfiles\myapp.ear -CCBootstrapHost=myWASServer -CCverbose=true app_parm1 app_parm2
```

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.

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.

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 2 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 J2EE infrastructure, the technology inherits the complete security architecture of the J2EE 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). Base 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, Version 6 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 three JNLP descriptor files:
• Client application JNLP descriptor (application-desc in the diagram)
• Application Clients run-time installer JNLP descriptor (installer-desc in the diagram)
• Application Clients run-time library component JNLP descriptor (component-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 J2SE Java Runtime Environment 5.0 or later is highly recommended. All the platforms supported by the application client for WebSphere Application Server are supported with the exception Linux on Power and OS400 platforms.

You can use any of the following:

- Java Web Start on the Java 2 Standard Edition Developer Kits that IBM provides, packaged in Application Client for WebSphere Application Server, Version 6.1
- Java Web Start on Sun Microsystems J2SE Software Development Kit or J2SE Java Runtime Environment 5.0, 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 2 Platform Standard Edition, version 5, which you can download from the HP Web site

Using Java Web Start

This topic provides the steps and prerequisites necessary to use Java Web start.

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 336
- "Client application Java Network Launcher Protocol deployment descriptor file" on page 339
- "ClientLauncher class" on page 343

Note: You can use any of the following:
- Java Web Start on Java 2 Standard Edition Developer Kits that IBM provides, packaged in the Application Client for WebSphere Application Server, Version 6.1
- Java Web Start on Sun Microsystems J2SE Software Development Kit or J2SE Java Runtime Environment 5.0, 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 2 Platform Standard Edition, version 5.0, which you can download from the HP Web site.

1. Prepare the Application Clients run-time dependency component for JWS.
2. Prepare the Application Clients run-time library component for JWS.
3. Installing JWS.
4. **Optional:** Run the Java Web Start sample.

   Problem: When you run Web services clients from Java Web Start using a Mozilla browser, you might get errors if the client argument contains quotations in the jnlp.jsp file. For example, the following argument results in an error:

   ```
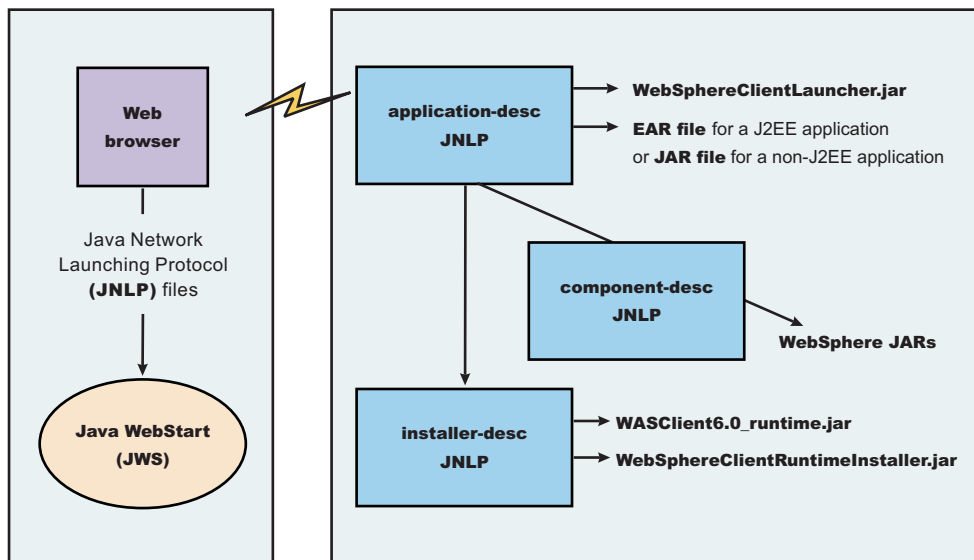   <argument>-url="wsejb:/com.ibm.wssvt.tc.pli.ejb.WSMultiProtocolHome?jndiName=com/ibm/wssvt/tc
   /pli/ejb/WSMultiProtocolHome&"</argument>
   ```

   Error: The following errors display in the Java Web Start console:

   If using the EJB protocol, the following error is displayed:

   ```
   Client caught exception getting the InsuranceWebServicesPort
   using the URL
   "wsejb:/com.ibm.wssvt.tc.pli.ejb.WSMultiProtocolHome?jndiName=com/ibm/wssvt/tc
   /pli/ejb/WSMultiProtocolHome&"
   java.net.MalformedURLException: no protocol:
   "wsejb:/com.ibm.wssvt.tc.pli.ejb.WSMultiProtocolHome?jndiName=com/ibm/wssvt/tc
   /pli/ejb/WSMultiProtocolHome&"
    at java.net.URL.<init>(URL.java(Compiled Code))
    at java.net.URL.<init>(URL.java(Compiled Code))
    at java.net.URL.<init>(URL.java:411)
    at com.ibm.wssvt.tc.pli.webservice.InsuranceWebServicesClient
   .getInsuranceServicesClientURL(InsuranceWebServicesClient.java:231)
    at com.ibm.wssvt.tc.pli.webservice.InsuranceWebServicesClient
   .main(InsuranceWebServicesClient.java:748)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:85)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:58)
   ```

```
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:60)
 at java.lang.reflect.Method.invoke(Method.java:391)
 at com.ibm.websphere.client.applicationclient.launchClient.
createContainerAndLaunchApp(launchClient.java:649)
```

If using the HTTP protocol, the following error is displayed:

```
Client caught exception getting the InsruanceWebServicesPort
using the URL
"http://svtlnx1:9081/WebSvcsInsSession20EJB/services/WSMultiProtocol"
java.net.MalformedURLException: no protocol:
"http://svtlnx1:9081/WebSvcsInsSession20EJB/services/WSMultiProtocol"
```

If using the JMS protocol, the following error is displayed:

```
Client caught exception getting the InsruanceWebServicesPort
using the URL
"jms:/queue?destination=jms/MultiProtocol_Q&connectionFactory=jms/InsuranceServices_Q

CF&targetService=WSMultiProtocolJMS&jndiProviderURL=IIOP://svtlnx1.austin.ibm.com:981
1"
java.net.MalformedURLException: no protocol:
"jms:/queue?destination=jms/MultiProtocol_Q&connectionFactory=jms/InsuranceServices_Q

CF&targetService=WSMultiProtocolJMS&jndiProviderURL=IIOP://svtlnx1.austin.ibm.com:981
1"
        at java.net.URL.<init> (URL.java(Compiled Code))
Making calls to methods in WSMultiprotocolWebServicesBean ...
```

Solution: To resolve the problem, update the jnlp.jsp file to remove the quotations (″ ″) from the argument. When a jnlp.jsp file has statements or expressions that begin with ″${″ and the statement is not to be interpreted as an expression, then add a backward slash ″\″, as shown in the following example:

```
<argument>-CCDcom.ibm.ssl.keyStore=${WAS_ROOT}/etc/DummyClientKeyFile.jks</argument>
<argument>-CCDcom.ibm.ssl.trustStore=${WAS_ROOT}/etc/DummyClientTrustFile.jks</argument>
```

For the EJB protocol, use the following example argument to correct the errors:

```
<argument>-url=wsejb:/com.ibm.wssvt.tc.pli.ejb.WSMultiProtocolHome?jndiName=com/ibm/wssvt/tc
/pli/ejb/WSMultiProtocolHome&</argument>
```

For the HTTP protocol, use the following argument to correct the errors:

```
<argument>-url=http://svtaix23:9081/WebSvcsInsSession20EJB/services/WSMultiProtocol</argument>
```

For the JMS protocol, use the following argument to correct the errors:

```
<argument>-url=jms:/queue?destination=jms/MultiProtocol_Q&connectionFactory=
jms/InsuranceServices_QCF&targetService=
WSMultiProtocolJMS&jndiProviderURL=IIOP://svtaix23.austin.ibm.com:9811</argument>
```

Now, rerun the client from Java Web Start.

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 2 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.

```
<j2se version="WASclient6.1.0"
href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJre/AppClientRT.jsp"/>
```

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 J2EE Application client application (that is the resources for the application contain an EAR file with a client application), then the launcher class starts a second Java Virtual Machine (JVM) using the JRE provided by the Application Clients run-time dependency and launches the J2EE Application client application that is packaged in the EAR file.

  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 J2EE Application client application" on page 341 for an example.

- 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.

  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 342 for an example.

  Unlike the J2EE Application client application, the Thin Application client application is not loading the Application Clients run-time library JAR files from the Application Clients run-time dependency. It is downloaded from the server directly as it is for the Thin Application client application JAR file. An Application Clients run-time library component JNLP descriptor is used for specifying the Application Clients run-time library JAR files resources, as shown in the following example:

```
<extension name="WAS Thin EJB Client Library"
              href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJars/AppClientLib.jnlp"/>
```

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 J2EE 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 J2EE Application client application (that is the resources for the application contain an EAR file with a client application), then the launcher class starts a second Java Virtual Machine (JVM) using the JRE provided by the Application Clients run-time dependency and launches the J2EE Application client application that is packaged in the EAR file.

Here is an example of the client application JNLP descriptor file for a J2EE 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."

Product 5630-A36,  (C) COPYRIGHT International Business Machines Corp., 2004
All Rights Reserved * Licensed Materials - Property of IBM
-->
<!--
 This is a generic jnlp for a client app.  It will specify the WAS JRE
  as a dependency as well as the client launcher
-->
<!-- ==================================================================== -->
<!-- TODO: change "codebase" to the actual url location of this jnlp file -->
<!-- ==================================================================== -->
<jnlp spec="1.0+"
 codebase="http://your_server:port_number/J2EEWebStartWeb/JavaClients/WebStartExample">

  <information>
    <title>J2EE Client Exampletitle>J2EE Client Example</title>
    <vendor>IBM Client Teamvendor>IBM Client Team</vendor>
    <description>JVE WebStart example of: J2EE Client Example</description>
    <description kind="short">J2EE Client Example</description>
    <description kind="tooltip">J2EE Client Example</description>
  </information>

  <security>
    <all-permissions/>
  </security>

  <resources>

    <!-- ========================================================================= -->
    <!-- TODO: Update the "version" value to match your Application Client runtime -->
    <!--       Update the "href" value to the url for downloading the Application  -->
    <!--        Client runtime.                                                    -->
    <!-- ========================================================================= -->
    <j2se version="WASClient6.1.0"
     href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJre/AppClientRT.jsp"/
```

```
    <!-- The client app will require a client launcher -->
    <jar href="../Launcher/WebSphereClientLauncher.jar" main="true"/>

    <!-- Ear we want to download to the client -->
    <jar href="J2eeJWS.ear"/>

    <!-- The launcher depends on this property to be set -->
    <property name="com.ibm.websphere.client.launcher.ear" value="J2eeJWS.ear"/>

  </resources>

  <!-- WebStart will consider the Launcher as the app. to run  -->
  <application-desc main-class="com.ibm.websphere.client.launcher.ClientLauncher">
    <argument>-CCproviderURL=corbaloc:iiop:your_server</argument>
    <argument>-CCDcom.ibm.ssl.keyStore=${WAS_ROOT}/etc/key.p12</argument>
    <argument>-CCDcom.ibm.ssl.trustStore=${WAS_ROOT}/etc/trust.p12</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."

Product 5630-A36,  (C) COPYRIGHT International Business Machines Corp., 2004
All Rights Reserved * Licensed Materials - Property of IBM
-->

<!--
   This is a generic jnlp for a client app.  It will specify the WAS JRE
   as a dependency as well as the client launcher
 -->

<!-- ===================================================================== -->
<!-- TODO: change "codebase" to the actual url location of this jnlp file -->
<!-- ===================================================================== -->
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+"
codebase="http://your_server:port_number/J2EEWebStartWeb/JavaClients/WebStartExample">
  <information>
    <title>Thin Basic Calculator Client Samples</title>
    <vendor>IBM</vendor>
    <description>Thin Basic Calculator Client Samples</description>
    <offline-allowed/>
  </information>

  <security>
    <all-permissions/>
  <security>

  <resources>
    <j2se version="WASClient6.1.0"
     href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJre/AppClientRT.jspz"/>
```

```
   <extension name="WAS Thin EJB Client Library"
              href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJars/AppClientLib.jnlp"/>

   <!-- you must use the jar resource for JWS LaunchClient class here if using JWS LaunchClient
wrapper launcher -->
   <jar href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJars/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:your_server:port_number"/>
  <property name="com.ibm.CORBA.ConfigURL"

value="http://your_server:port_number/J2EEWebStartWeb/JavaClients/sas.client.props"/>
  <property name="com.ibm.SSL.ConfigURL"

value="http://your_server:port_number/J2EEWebStartWeb/JavaClients/ssl.client.props"/>

  <!-- *** Logging Properties ***
      <property name="com.ibm.websphere.client.launcher.jws.trace" />
      <property name="java.util.logging.configureByServer" value="true" />
      <property name="traceSettingsFile" value="TraceSettings.properties" />
      <property name="com.ibm.CORBA.Debug" value="true" />
      <property name="com.ibm.CORBA.CommTrace" value="true" />
      <property name="java.util.logging.manager" value="com.ibm.ws.bootstrap.WsLogManager" />
      <property name="com.ibm.CORBA.RasManager" value="com.ibm.websphere.ras.WsOrbRasManager" />
  -->
  </resources>
   <application-desc>
     <argument>-CCDcom.ibm.ssl.keyStore=${WAS_ROOT}/etc/key.p12</argument>
     <argument>-CCDcom.ibm.ssl.trustStore=${WAS_ROOT}/etc/trust.p12</argument>
     <argument>add</argument>
     <argument>1</argument>
     <argument>2</argument>
   </application-desc>
  </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 a WebSphere Application Server clients installation under the *<app_server_root>*/ JWS directory.

This launcher class configures the run-time environment for J2EE application clients and thin client applications (not J2EE application clients).

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 J2EE 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.

Launcher tool:

The launcher configures the run-time environment for J2EE application clients and thin client applications (not J2EE application clients). The launcher utility is located in the main entry point of the Java Network Launching Protocol (JNLP) application client. The main class launcher name is com.ibm.websphere.client.launcher.ClientLauncher and is located in the `WebSphereClientLauncher.jar` file.

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.

com.ibm.websphere.client.launcher.ear

> If the client that is to run is the J2EE 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.

com.ibm.websphere.client.launcher.classpath.* (required for J2EE client applications only)

> There can be a set of properties that are prefixed with com.ibm.websphere.client.launcher.classpath. Each property specifies a JAR file that is to be added to the class path of the application. This JAR file is a JAR file that is already defined as a resource for the application. This file is needed so that the correct elements of the class path of the Java Virtual Machine (JVM) starting the client launcher can be retrieved and added to the class path of the (JVM) that is to be spawned for the application client.

These properties are not defined in a separate properties file. Instead, they are defined as part of the Java Network Launching Protocol files.

Preparing the Application Client run-time dependency component for Java Web Start:

For a J2EE application client application and or Thin application client application to be launched using Java Web Start (JWS), an Java Runtime Environment that IBM provides, the library JAR files and properties files bundled in Application Client for WebSphere Application Server must be installed in the JWS. This article provides 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.

Install the Application Client for WebSphere Application Server for the platform to which the client application deploys. If there is a requirement to deploy the client application to multiple platforms, the Application Client run-time dependency component must be built separately for each platform that client application supports.

For example, if the client application deploys to both the Windows platform and Linux platform, follows the steps for this task to build the Application Client run-time dependency component for Windows on a Windows platform machine with the Application Client for WebSphere Application Server for Windows installed. Now, repeat the steps for this task to build the Application Client run-time dependency component for Linux on a Linux platform machine with the Application Client for WebSphere Application Server for Linux installed.

1. Install the Application Client for WebSphere Application Server for the client application supported operating systems. Install Application Client in the `C:\Program Files\IBM\WebSphere\AppClient` directory.

2. Change the directory to the installation bin directory. See the following example for help:

   ```
   CD C:\Program files\IBM\WebSphere\AppClient\bin
   ```

3. Run the buildClientRuntime tool to generate the Application Client run-time JAR file in a temporary directory which contains the Java 2 Runtime Environment, Application Client run-time properties, the SSL KeyStore and TrustStore file, and the Application Client run-time library JAR files. See the following example for help:

   ```
   buildClientRuntime C:\WebApp1\runtime\WASClient6.1_windows.jar
   ```

   If you are building an Application Client run-time JAR file only for serving Thin application client applications and not for J2EE application client applications, you can reduce the size of the generated JAR file by not including the Application Client run-time library JAR files. An extra parameter is passed to the buildClientRuntime tool, as the following example shows:

   ```
   buildClientRuntime C:\WebApp1\runtime\WASClient6.1_windows.jar
                      buildThin
   ```

4. Copy the `WebSphereClientRuntimeInstaller.jar` file to the same location of the JAR file generated in the previous step. This JAR file is located in the JWS directory of the WebSphere Application Server clients installation. See the following example for help:

   ```
   copy ..\JWS\WebSphereClientRuntimeInstaller.jar   C:\WebApp1\runtime
   ```

5. Sign the JAR files created from the previous steps, using the Java 2 SDK jarsigner utility, as the following example shows:

   ```
   cd C:\WebApp1\runtime

   jarsigner -keystore myKeystore -storepass myPassword
               WASClient6.1_windows.jar myKeyAliasName

   jarsigner -keystore myKeystore -storepass myPassword
               WebSphereClientRuntimeInstaller.jar myKeyAliasName
   ```

   a. This step also requires you to create a keystore file, such as `myKeystore`.

   b. You must also create a self-signed certificate for the `myKeystore` file.

   Note: When running the JAR signer tool on HP platforms, add the `-J″-Xmx256m″` flag to the jarsigner command to increase the available heap size and prevent the error, `OutOfMemoryError`. See the following example for help:

   ```
   jarsigner -J"-Xmx256m" -keystore myKeystore -storepass myPassword
               WebSphereClientRuntimeInstaller.jar myKeyAliasName
   ```

6. Create an Application Client run-time installer JNLP descriptor file or a JavaServer Pages (JSP) file if it is generated dynamically in the same temporary directory as previous step. See the sample JNLP file shown in the Example section of this topic.

7. Package the two signed JAR files and the Application Client run-time installer JNLP descriptor file into a WAR file. This WAR file is packaged into an EAR file that can be deployed to an Application Server.

Your Web application is ready to serve the Application Client run time and the JRE environment.

```
<!-- 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.

Product 5630-A36,  (C) COPYRIGHT International Business Machines Corp., 2005
All Rights Reserved * Licensed Materials - Property of IBM
-->

<%-- // to set the Last_Modified header so that the JNLP client will know whether to download
     // the JNLP file again and update the cached copy.
```

```
        String jspPath = application.getRealPath(request.getServletPath());
        java.io.File jspFile = new java.io.File(jspPath);
        long lastModified = jspFile.lastModified();
%><%
        // locally declared variables
        String url=request.getRequestURL().toString();
        String jnlpCodeBase=url.substring(0,url.lastIndexOf('/'));
        String jnlpRefURL=url.substring(url.lastIndexOf('/')+1,url.length());

        // Need to set a JNLP mime type - if WebStart is installed on the client,
        // this header will induce the browser to drive the WebStart Client
        response.setContentType("application/x-java-jnlp-file");                        1
        response.setHeader("Cache-Control", null);
        response.setHeader("Set-Cookie", null);
        response.setHeader("Vary", null);
        response.setDateHeader("Last-Modified", lastModified);

        // An installer must reply with the version number for a given install
        if (response.containsHeader("x-java-jnlp-version-id"))
          response.setHeader("x-java-jnlp-version-id", "WASClient6.1.0");              2
        else
          response.addHeader("x-java-jnlp-version-id", "WASClient6.1.0");
%>

<?xml version="1.0" encoding="utf-8"?>


<!-- ============================================================ -->
<!-- TODO: change "codebase" to the actual url location of this jsp -->
<!-- ============================================================ -->

<jnlp spec="1.0+"
codebase="http://YOUR_APP_SERVER:PORTNUMBER/WEBAPP_CONTEXT_ROOT/Runtime/WebSphereJre">

<information>
  <title>Application Client Java Runtime Environment</title>
  <vendor>IBM</vendor>
  <icon href="icon.gif"/>
  <description>Application Client Java Runtime Environment</description>
  <description kind="short">Applicaiton Client JRE</description>
  <description kind="tooltip">Applicaiton Client JRE</description>
  <offline-allowed/>
</information>

<security>
 <all-permissions/>
</security>

 <resources>
  <j2se version="1.4+"/><%-- The installer can use any 1.4 JRE --%> 3
   <jar href="WebSphereClientRuntimeInstaller.jar" main="true"/> 4

  <!-- JRE version registration with Web Start -->
  <property name="com.ibm.websphere.client.jre.version" value="WASClient6.1.0"/> 5
 </resources>

 <resources os="Windows"> 6
<!-- ============================================================ -->
<!-- TODO: the property value for unix platform is "java/jre/bin/javaw" -->
<!--    and the "os" value match to your target client machine platform          -->
<!-- ============================================================ -->

   <jar href="WASClient6.1.0_Windows.jar"/> 7


<!-- ============================================================ -->
<!-- TODO:  property value for unix platform is "java/jre/bin/javaw" -->
<!-- ============================================================ -->
<!-- relative path of the jre executable -->
```

346   Developing and deploying applications

```
   <property name="com.ibm.websphere.client.jre.launch.java"
value="java\jre\bin\javaw.exe"/> 8

 </resources>
 <installer-desc main-class="com.ibm.websphere.client.installer.ClientRuntimeInstaller"/>
</jnlp>
```

1. Specifies that the file is a JNLP mime type so that the browser can process the JNLP file.
2. Specifies the exact version of this Application Client run-time dependency component in the response by setting the HTTP header field: `x-java-jnlp-version-id`.
3. Specifies the required JRE version to run the installer program.
4. Specifies the installer `WebSphereClientRuntimeInstaller.jar` file, which contains the ClientRuntimeInstaller class.
5. Specifies a system property that defines the version of Application Client run-time dependency component. This version is registered to the JNLP client.
6. Specifies resources for a particular platform. Each supported client application platform needs its own separate JAR file.
7. Specifies the Application Client run-time dependency component JAR file.
8. Specifies the program to call that starts a JVM for the client application.

Preparing Application Client run-time library component for Java Web Start.

buildClientRuntime tool:

For a J2EE 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 2 Runtime Environment (JRE) that IBM provides
- Application Clients run-time properties and configuration
- SSL KeyStore and TrustStore files
- Run-time library JAR files

In the case of building an Application Clients run-time JAR file only for serving Thin Application client applications and not for J2EE Application client applications, the run-time library JAR files and the Application Clients run-time 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  jar_file  [type]
Unix Usage:       buildClientRuntime.sh   jar_file  [type]
Where:
      jar_file
Specifies the target jar file name.

Range:
                  buildJ2EE - Default value that builds a Application Clients
                          run-time library for J2EE application.
                  buildThin - Builds a Application Clients run-time library
                          for Thin application.
```

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="java\jre\bin\javaw.exe"/>
<property name="com.ibm.websphere.client.jre.launch.java" value="WASclient6.1"/>
```

Preparing Application Clients run-time library component for Java Web Start:

The Java Web Start client is used with platforms that support a Web browser. For a Thin Application client application to be launched using Java Web Start (JWS), you also need to create a Java Network Launching Protocol (JNLP) component to serve the Application Clients run-time library JAR files from the Application server. This JNLP component is referenced in the client application JNLP file with the <extension> tag. This article provides the steps to build the Application Clients run-time library component from an Application Clients installation. It is packaged as its own Web Archive Resource (WAR) file or to the same WAR file that contains the Application Clients run-time dependency component, and can be installed in an Application server.

Install the Application Client for WebSphere Application Server for the platform to which client applications deploy.
1. Install the Application Clients on the client application supported operating system. For example, install Application Clients in the `C:\Program Files\IBM\WebSphere\AppClient` directory.
2. Change the directory to the installation bin directory. For example:

   `CD C:\Program files\IBM\WebSphere\AppClient\bin`
3. Run `buildClientLibJars` to copy the Application Clients run-time library JAR files from the Application Clients installation to a temporary directory. All the JAR files in the temporary directory are signed, as shown in the following example:

   ```
   buildClientLibJars C:\WebApp1\runtime\WebSphereJars
                    myKeystore myPassword myKeyAliasName
   ```
   a. This step also requires you to create a keystroke file, such as `myKeystore`.
   b. You must also create a self-signed certificate for the `myKeystore` file.
4. Create an Application Clients run-time installer JNLP descriptor file in the same temporary directory as the previous step. See the sample JNLP file shown in the Example section of this topic.
5. Package these JAR files and the Application Clients run-time library component JNLP descriptor file into a WAR file. You can also package both Application Clients run-time library component and

Application Clients run-time dependency component in the same WAR file. This WAR file is packaged into an EAR file that can deployed to an Application server.

```
<!--
  "This sample program is provided AS IS and can 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."
Product 5630-A36,  (C) COPYRIGHT International Business Machines Corp., 2005
All Rights Reserved * Licensed Materials - Property of IBM
-->
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+"
codebase="http://YOUR_APP_SERVER:PORTNUMBER/WEBAPP_CONTEXT_ROOT/Runtime/WebSphereJars">
 <information>
  <title>Application Client Library</title>
  <vendor>IBM</vendor>
  <icon href="icon.gif"/>
  <description>Application Client Library</description>
  <description kind="short">Application Client Library</description>
  <description kind="tooltip">Application Client Library</description>
  <offline-allowed/>
 </information>

<security>
 <all-permissions/>
</security>

<component-desc/>
 <resources><jar href="activation-impl.jar"/>
<jar href="bootstrap.jar"/>
<jar href="com.ibm.events.client_6.1.0.jar"/>
<jar href="com.ibm.mq.jar"/>
<jar href="com.ibm.mqjms.jar"/>
<jar href="com.ibm.uddi.client_1.0.0.jar"/>
<jar href="com.ibm.ws.bootstrap_6.1.0.jar"/>
<jar href="com.ibm.ws.debug.osgi_6.1.0.jar"/>
<jar href="com.ibm.ws.emf_2.0.0.jar"/>
<jar href="com.ibm.ws.j2ee.client_6.1.0.jar"/>
<jar href="com.ibm.ws.runtime.dist_6.1.0.jar"/>
<jar href="com.ibm.ws.runtime.gateway_6.1.0.jar"/>
<jar href="com.ibm.ws.runtime_6.1.0.jar"/>
<jar href="com.ibm.ws.security.crypto_6.1.0.jar"/>
<jar href="com.ibm.ws.sib.client_2.0.0.jar"/>
<jar href="com.ibm.ws.sib.utils_2.0.0.jar"/>
<jar href="com.ibm.ws.wccm_6.1.0.jar"/>
<jar href="com.ibm.wsspi.extension_6.1.0.jar"/>
<jar href="dhbcore.jar"/>
<jar href="j2ee.jar"/>
<jar href="launchclient.jar"/>
<jar href="lmproxy.jar"/>
<jar href="mail-impl.jar"/>
<jar href="org.eclipse.core.runtime_3.1.1.jar"/>
<jar href="org.eclipse.osgi_3.1.1.jar"/>
<jar href="org.eclipse.update.configurator_3.1.0.jar"/>
<jar href="properties.jar"/>
<jar href="serviceadapter.jar"/>
<jar href="startup.jar"/>
<jar href="urlprotocols.jar"/>
<jar href="WebSphereClientLauncher.jar"/>
 <resources/><jnlp/>
```

buildClientLibJars tool:

For a J2EE 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` and `alias` to sign all the JAR files in the specified location.

```
Windows Usage: buildClientLibJars.bat  target_dir keystore  storepass  alias
Unix Usage:       buildClientLibJars.sh  target_dir keystore  storepass  alias
Where:
    target_dir        Specifies the target directory where the Application
                        Clients library JAR files copied to.
    keystore          Specifies a keystore file.
    storepass         Specifies the keystore password.
    alias               Specifies an alias for the key object in the key file.
```

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.

Install the Application Client for WebSphere Application Server for the platform to which the client application deploys. If there is a requirement to deploy the client application to multiple platforms, the Application Clients run-time dependency component must be built separately for each platform that the client application supports.

1. Install the Application Clients on the client application supported operating system. For example, install Application Clients in the `C:\Program Files\IBM\WebSphere\AppClient` directory.
2. Create the following temporary working directories:
   ```
   MKDIR C:\WebApp1
   MKDIR C:\WebApp1\runtime
   MKDIR C:\WebApp1\runtime\Widnows
   MKDIR C:\WebApp1\runtime\WebSphereJars
   ```
3. Change directory to the installation bin directory. See the following example for help:
   ```
   CD C:\Program files\IBM\WebSphere\AppClient\bin
   ```
4. Run the buildClientRuntime tool to generate the Application Clients run-time JAR file in a temporary directory that contains the Java 2 Runtime Environment that IBM provides, Application Clients run-time properties, the SSL KeyStore and TrustStore files, and the Application Clients run-time library JAR files. See the following example for details:
   ```
   buildClientRuntime C:\WebApp1\runtime\windows\WASClient6.1.0_Windows.jar
   ```
5. Copy the `WebSphereClientRuntimeInstaller.jar` file to the same location of the JAR file generated in the previous step. This JAR file is located in the `JWS` directory of the Application Client for WebSphere Application Server installation. For example, copy the `..\JWS\WebSphereClientRuntimeInstaller.jar` file to the `C:\WebApp1\runtime` directory.
6. Sign the JAR files created from the previous steps, using the Java 2 SDK jarsigner utility. See the following example for details:
   ```
   cd C:\WebApp1\runtime

   jarsigner -keystore myKeystore -storepass myPassword
               WASClient6.1_windows.jar myKeyAliasName

   jarsigner -keystore myKeystore -storepass myPassword
               WebSphereClientRuntimeInstaller.jar myKeyAliasName
   ```

a. This step also requires you to create a keystore file, such as `myKeystore`.

b. You must also create a self-signed certificate for the `myKeystore` file.

7. Run buildClientLibJars to copy the Application Clients run-time library JAR files from the Application Client for WebSphere Application Server installation to a temporary directory. All the JAR files in the temporary directory are signed. See the following example for details:

```
buildClientLibJars C:\WebApp1\runtime\WebSphereJars
                    myKeystore myPassword myKeyAliasName
```

a. This step also requires you to create a keystroke file, such as `myKeystore`.

b. You must also create a self-signed certificate for the `myKeystore` file.

8. Add all the JAR files created in the previous steps in the `C:\WebApp1` directory to the WAR file within the `WebSphereClientRuntime.ear` file. The contents of the WAR file are shown in the following example:

```
The root of the WAR
├───META-INF
                MANIFEST.MF

├───Runtime
│   ├───WebSpherejars
│   │       AppClientLib.jnlp
│   │       com.ibm.ws.runtime_6.1.0.jar
│   │       com.ibm.ws.j2ee.client_6.1.0.jar
│   │       com.ibm.ws.wccm_6.1.0.jar
│   │                :
│   │       (all the jars created in step 7 under
│   │       c:\WebApp1\Runtime\WebSphereJars)
│   └───WebSphereJre
│           AppClientRT.jsp
│           WASClient6.1.0_Windows.jar
│           WebSphereClientLauncher.jar
│           WebSphereClientRuntimeInstaller.jar |

└───WEB-INF
        ibm-web-bnd.xmi
        ibm-web-ext.xmi
        web.xml
```

9. Install the `WebSphereClientRuntime.ear` file to an Application Server. You have just created an Application Clients run-time dependency component and Application Clients run-time libraries for serving J2EE Application client applications and Thin Application client applications using Java Network Launching Protocol (JNLP) or Java Web Start (JWS).

Installing Java Web Start:

This topic provides the steps necessary to install JWS on the AIX platform.

Before you begin this task, see the following topics to understand Java Web Start (JWS) technology and its components:

- Prepare the Application Clients run-time dependency component for JWS
- Prepare the Application Clients run-time library component for JWS

Note: You can use Java Web Start on Java 2 Standard Edition Developer Kits that IBM provides, packaged in the Application Client for WebSphere Application Server, Version 6; Java Web Start on Sun Microsystems J2SE Software Development Kit or J2SE Java Runtime Environment 1.4.2, which you can download from the Sun Microsystems Web site for Windows, Linux and Solaris operating systems, or the Java Web Start on HP SDK or RTE for Java 2 version 1.4.2, which you can download from the HP Web site.

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` path.

3. Run `sh bin/webstart_install_sdk.sh`.

4. When prompted for the Java path, enter your JRE path. Use the following example:
   `client_install_root/java/jre`.

5. You should see the following messages, which indicate that JWS installs successfully:
   - Obtaining version...
   - You appear to be running 1.4.2
   - Extracting...
   - Updating ~/.mailcap...
   - Updating ~/.mime.types...

6. Change your path to the JWS installed path. For example, enter `client_install_root/java/jre/javaws/`.

7. Run `./javaws` from the path mentioned in the previous step.

Java Web Start for Application client best practices:

Do not use JSP to dynamically generate a JNLP file, otherwise the JNLP jsp page cannot be opened in some IE browsers. 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 base interface for all commands is the Command interface.

To write a command interface, you extend one or more of the three interfaces 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.

The implementation class for your interface must contain implementations for the isReadyToCallExecute and reset methods. The execute method is implemented for you elsewhere; for more information, see "Implementing command interfaces" on page 356. 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 354.

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 "Writing a command target (client-side adapter)" on page 365

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.

Overview of the command package

The command package can be used by distributed applications to reduce the number of remote invocations that a client makes.

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

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 ... }
```

Facilities for 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.

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.

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.

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.

* 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;
...
}
```

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.

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.");
}
}
...
}
```

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();
}
...
}
```

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() {}
}
```

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.

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());
}
...
}
```

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());
      }
      ...
      }
```

Using the WebSphere Application Server EJBCommandTarget bean as a command target:

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.

Writing a command target (server):

A server must implement the CommandTarget interface and its single method, executeCommand, to accept commands.

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. "Writing a command target (server)" shows 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. "Writing a command target (server)" shows 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;
}
}
```

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
...
}
```

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
....
}
```

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.

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 362 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 362 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());
}
...
}
```

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 362, 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");
...
}
```

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 example

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.

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());
}
}
```

Web services

Implementing Web services applications

This topic introduces you to using Web services that are based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification. WebSphere Application Server supports Web services that are developed and implemented based on Web Services for J2EE. Use Web services when operating across a variety of platforms, including the J2EE 1.4 and non-J2EE platforms.

Decide if a Web service implementation benefits your business process.

Implementing Web services applications is an easy way to integrate application systems together within or outside your company's infrastructure that otherwise function as a standalone systems. For example, your customer information database is a standalone application, but you want your accounting application to be able 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 manner.

Because Web services are easily applied to existing applications and information technology assets, new solutions can be deployed quickly and recomposed 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 WebSphere Application Server by following the steps provided:

1. Plan to use Web services. Review the Universal Description, Discovery, and Integration (UDDI) and Web services enablement of the service integration bus (SIBWS) concepts to learn how these components can make your Web services plan more robust.

2. (Optional) Migrate existing Web services.

   If you have used Web services based on Apache SOAP and now want to develop and implement Web services based on the Web Services for J2EE specification, you need to migrate client applications developed with all versions of 4.0, and versions of 5.0 prior to 5.0.2.

3. Develop Web services.

   This topic is a good starting point in learning about how to develop a J2EE Web service.

4. Configure Web services deployment descriptors.

   You need to configure the deployment descriptors so that WebSphere Application Server can process the incoming Web services requests.

5. Assemble Web services.

   This topic presents what you need to assemble a Web service and in what order you should assemble the parts, for example an enterprise archive (EAR) file.

6. Deploy Web services.

   This topic presents the steps necessary to deploy the EAR file that has been configured and enabled for Web services.

7. Configure Web service client bindings. This topic explains how to edit bindings for a Web service after these bindings are deployed on a server. When one Web service communicates with another Web service, you must configure the client bindings to access the downstream Web service.

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. This topic presents the steps necessary to publish the WSDL files so that this information is available.

9. Develop Web services clients.

   This topic explains how to develop a Web services client based on the Web Services for J2EE specification.

10. Secure Web services.

    This topic presents the methods used to integrate message-level security into a WebSphere Application Server environment. If you are using V5.x, refer to Securing Web services for version 5.x applications based on WS-Security. If you are using V6.x, refer to Securing Web services for version 6 applications based on WS-Security

11. Monitoring the performance of Web services applications.

    This topic includes information to help you use the Performance Monitoring Infrastructure (PMI) to measure the time required to process Web services requests.

12. Troubleshoot Web services.

    You can use this topic to learn more 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.

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 flower shop owner's criteria.

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 supplier's Web page, received through e-mail, or retrieved from the supplier's UDDI registry entry.

The WSDL describes the procedure call. When using WebSphere Application Server, the procedure call is a Java API for XML-based remote procedure call (JAX-RPC), which retrieves price lists. 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 back 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 JAX-RPC. 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 all the suppliers. If the supplier ships directly to the customer, the order tracking inquiries can pass directly to the supplier's order tracking system. The supplier can also use Web services to send invoices for orders and by the flower shop to pay the supplier's invoices. 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, but 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 money invested into additional product.

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 they incorporated the Web services concept.

Web Services for J2EE specification

The *Web services for Java 2 Platform, Enterprise Edition (J2EE)* specification defines the programming model and run-time architecture for implementing Web services based on the Java language. Another name for the Web Services for J2EE specification is the Java Specification Requirements (JSR) 109. The specification includes open standards for developing and implementing Web services.

The Web Services for J2EE specification focuses on Extensible Markup Language (XML) remote procedure call (RPC) and the Java language, including representing XML-based interface definitions in the Java language; Java language definitions in XML-based definition languages, such as SOAP, and assembling.

The J2EE technology can be integrated with Web services in a variety of ways. J2EE components, for example, JavaBeans and enterprise beans, can be exposed as Web services. These services can be accessed by clients written in Java code or by existing Web service clients that are not written in Java code. J2EE components can also act as Web service clients.

The Web Services for J2EE specification is the preferred platform for Web-based programming because it provides open standards allowing 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 Web Services Description Language (WSDL) file and the Java application must exist. The mapping is defined by the Java API for XML-based RPC (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 encompassed is based on the Web Services for J2EE specification.

The specification brings with it the `webservices.xml` deployment descriptor specifically for Web services. 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
- 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. To learn more about the EJB 2.1 specification see Enterprise beans: Resources for learning.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

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 required part of the J2EE 1.4 platform. The J2EE 1.4 platform allows you to develop portable Web services. Web services can also be developed and deployed on J2EE 1.3 containers.

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 allows 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 throw `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.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

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.

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 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.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

SOAP with Attachments API for Java interface

SOAP with Attachments API for Java (SAAJ) interface is used for SOAP messaging that works behind the scenes in the Java API for XML-based RPC (JAX-RPC) implementation. You can map XML to Java types with standards supported by the JAX-RPC specification, but there are limited XML schema types, therefore you can use the SOAPElement interface to create a *custom data binder*.

Web services use XML messages to exchange messages. These messages conform to XML schema and when developing We 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. To manipulate the XML schema types, you need to map the XML schema types to Java types with a *custom data binder*.

Web services uses SOAP messages to represent remote procedure calls between the client and the server. In normal JAX-RPC flows, the SOAP message is deserialized into a series of Java value-type business objects that represent the parameters and return values. In addition, JAX-RPC 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 JAX-RPC, the specification provides the SAAJ data model as an extension to manipulate the message.

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 SAAJ 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 Document Object Model (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 Service Data Object (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.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

Web services SOAP/JMS protocol

The Web services engine supports the use of a Java Message Service (JMS)-compliant messaging transport as an alternative to HTTP for communicating SOAP messages between clients and servers.

You can use SOAP/JMS if you need to provide implementations for the client or server components, and 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 either a JMS TextMessage, for example, javax.jms.TextMessage, or a BytesMessage, for example, javax.jms.BytesMessage, to transmit the SOAP request message to the server. If the request message contains attachments, a BytesMessage must be used. If the request message does not contain attachments, the client can use a TextMessage or a BytesMessage. The WebSphere product client implementation uses only a BytesMessage 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.
  - **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.
  - **endpointURL** - This property must be set to the JMS endpoint URL associated with 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's setJMSReplyTo method is used for this.
- 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, regardless of the type of JMS message used to transmit the SOAP request. The WebSphere product implementation of the server component responds with the same type of JMS message that is received from the client, unless the response contains attachments and a BytesMessage 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.
  - The **correlation ID** of the JMS reply message should be set to the **message ID** of the original JMS request message. This is done by calling the JMS message's `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's `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 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>
```

Example: SOAP request with attachments

The following example displays the results from calling the JMS message's `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"
0d0a2d2d2d2d2d2d3d5f506172745f305f3234373935333339372e31303932303836303131393730
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's `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"
0d0a2d2d2d2d2d2d2d3d5f506172745f305f313635353030363735342e3130393233131323835343734
350d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d38
...
```

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

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 1.1.

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.

Each of the technology components have 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 using 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.

See how to modify your encoding from UTF-8 to UTF-16 if you need to change from UTF-8 to UTF-16.

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.

RMI-IIOP using JAX-RPC

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 J2EE 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. Learn more about this by reviewing 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.

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.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

Web services: Resources for learning

This topic provides relevant supplemental information about Web services-related topics.

The following topics provide extended reference information about Web services:
- Web services overview
- Developing Web services:

  This topic includes information about developing Web services that based on the Java 2 Platform, Enterprise Edition (J2EE) and Java API for XML-based remote procedure call (JAX-RPC) specifications.
- Performance

  Review this topic for information about key Web sites that discuss performance best practices.
- Universal Description Discovery and Integration (UDDI)

  This topic is an overview about UDDI and information about the UDDI Java API.
- The Web Services Invocation Framework (WSIF)

  This topic includes a look into the Apache Software Foundation and its maintenance of WSIF.
- Web Services-Interoperability (WS-I) Basic Profile

  This topic is an overview about the WS-I Basic Profile.
- SOAP

  This topic is an overview about SOAP, information about the SOAP syntax and processing rules.
- Security

  This topic provides a roadmap to security, the WS-Security specification, best practices, a profile of the OASIS Security Assertion Markup Language (SAML) and more.
- Samples

  This topic is the Samples Gallery for WebSphere Application Server and Samples Central for UDDI and WSIF.

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 an 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.

Web services overview

*   WebSphere Version 6 Web Services Handbook

    This IBM Redbook describes the new concept of Web services from various perspectives. It presents the major building blocks on which Web services rely. Well-defined standards and new concepts are presented and discussed.
*   Web services (r)evolution, Part 1

    This article focuses on the benefits and challenges of building Web services applications. Web services might be an evolutionary step in designing distributed applications, however, the technology is not without problems. Outlined are the difficulties developers face in creating a truly workable distributed system of Web services. This article also outlines author Graham Glass's plan for building peer-to-peer Web applications.

Developing Web services

*   Java Web Services: SOAP with Attachments API for Java (SAAJ)

    This document describes the SOAP with Attachments API for Java (SAAJ) and how this API provides a standard way to send XML documents over the Internet from the Java platform.
*   JSR 109: Implementing Enterprise Web services

    This document describes the J2EE specification.
*   JAX-RPC: Core Web services API in the Java platform

    This document reviews the JAX-RPC specification which enables Java technology developers to develop SOAP-based interoperable and portable Web services.
*   A developer introduction to the JAX-RPC specification, Part 1: Learn the ins and outs of the JAX-RPC type-mapping system. The JAX-RPC specification is an important step forward in the quest for Web services interoperability. This DeveloperWorks WebSphere article explains the mapping between WSDL and XML types and Java types. It explains how the JAX-RPC standard defines this feature and some of the important points on designing an interoperable type system.
*   A developer introduction to JAX-RPC, Part 2: Mine the JAX-RPC specification to improve Web service interoperability. This DeveloperWorks WebSphere article explains how you can achieve the next level of Web service interoperability using the JAX-RPC standard client and server side interface definitions and message processing model. It includes information on developing JAX-RPC handlers and handler chains.
*   Getting Started with JAX-RPC. This article explains the basic JAX-RPC programming concepts. It describes the JAX-RPC client and server programming models and illustrates simple examples. The article is intended to give developers a good grasp of how to use the JAX-RPC specification to develop or use Web services.
*   Web Services Description Language

    This article is a detailed overview of Web Services Description Language (WSDL), which includes programming specifications.

Performance

The following Web sites offer tips and best practices to get the best performance from your Web services applications:

*   Best practices for Web services: Part 1, Back to the basics
*   SOA and Web services: Articles
*   IBM WebSphere Developer Technical Journal: Web Services Architectures and Best Practices
*   Web services programming tips and tricks: How to create a simple JAX-RPC handler
*   Web services programming tips and tricks: Using SOAP headers with JAX-RPC
*   Web services programming tips and tricks: Extend JAX-RPC Web services using SOAP headers
*   Web services programming tips and tricks: Roundtrip issues in Java coding conventions

UDDI

- Universal Description, Discovery and Integration

  This article is a detailed overview of the UDDI registry.

- A new approach to UDDI and WSDL: Introduction to the new OASIS UDDI WSDL Technical Note

  This article is about using WSDL with UDDI. Although it is based on the UDDI Registry in WebSphere Application Server Version 5, it remains a useful description of the recommended approach for use of WSDL with UDDI.

- UDDI Version 3 Features List

  This article is an introduction to the new features in UDDI Version 3.

WSIF

- The Apache Software Foundation. The Apache Software Foundation provides support for the Apache community of open-source software projects. Of particular interest is the Apache Web services project. The WSIF source code is donated by IBM to the Apache Software Foundation, and is maintained here as an Apache project.

WS-I Basic Profile

- Web Services Interoperability Organization This Web site offers resources and guidelines for Web services interoperability. You can also view the latest specification documents for WS-I Basic Profile from the documentation link on the home page.

- UTF and BOM Frequently Asked Questions. This Web site offers general information about UTF-8, UTF-16, UTF-32, along with Byte Order Mark (BOM) in a question and answer format.

SOAP

- SOAP

  This article is a detailed overview of SOAP, which includes the programming specifications.

- SOAP Security Extensions: Digital Signature

  This document specifies the syntax and processing rules of a SOAP header entry to carry digital signature information within a SOAP 1.1 Envelope

Security

- Security in a Web Services World: A Proposed Architecture and Roadmap

  This document describes a proposed model for addressing security within a Web service environment. It defines a comprehensive Web services security model that supports, integrates, and unifies several popular security models, mechanisms, and technologies, including both symmetric and public key technologies. You can enable a variety of systems to securely interoperate in a platform and language-neutral manner. It also describes a set of specifications and scenarios that show how these specifications can be used together.

- Web Services Security (WS-Security)

  The Web Services Security (WS-Security) specifications describe enhancements to SOAP messaging to provide the quality of protection through message integrity, message confidentiality, and single message authentication. Use these mechanisms to accommodate a wide variety of security models and encryption technologies. The WS-Security specification also provides a general-purpose mechanism for associating security tokens with messages. Additionally, the specification describes how to encode binary security tokens. Specifically, the specification describes how to encode X.509 certificates and Kerberos tickets, as well as how to include opaque encrypted keys. It also includes extensibility mechanisms that can be used to further describe the characteristics of the credentials that are included with a message.

- SOAP Security Extensions: Digital Signature

  This document specifies the syntax and processing rules of a SOAP header entry to carry digital signature information within a SOAP 1.1 envelope

- Web Services Security Addendum

This document describes clarifications, enhancements, best practices, and errata of the WS-Security specification.

- WS-Security Profile of the OASIS Security Assertion Markup Language (SAML) Working Draft 04, 10 September 2002

    This document proposes a set of standards for SOAP extensions that are used to increase message confidentiality.
- Web Services Security: SOAP Message Security Working Draft 12, Monday 21 April 2003

    This document describes the support for multiple token formats, trust domains, signature formats, and encyrption technologies.
- JSR 55:Certification Path API

    This document provides a short description of the Certification Path API.
- XML-Signature Syntax and Processing

    This document specifies XML digital signature processing rules and syntax. XML signatures provide integrity, message authentication, or signer authentication services for data of any type, whether it is located within the XML that includes the signature or elsewhere.
- Canonical XML Version 1.0

    This specification describes a method for generating a physical representation or the canonical form of an XML document that accounts for the permissible changes.
- Exclusive XML Canonicalization Version 1.0

    Canonical XML [XML-C14N] specifies a standard serialization of XML that, when applied to a subdocument, includes the subdocument ancestor context including all of the namespace declarations and attributes in the ″xml:″namespace.
- XML Encryption Syntax and Processing

    This document specifies a process for encrypting data and representing the result in XML.
- Decryption Transform for XML Signature

    This document specifies an XML Signature decryption transform that enables XML Signature applications to distinguish between those XML encryption structures that are encrypted before signing, and must not be decrypted, and those that are encrypted after signing, and must be decrypted, for the signature to validate.
- WS-Security

    This document specifies resources for the April 2002 WS-Security specification. The following addenda and drafts are available:
    - http://schemas.xmlsoap.org/ws/2002/07/secext/
    - http://schemas.xmlsoap.org/ws/2002/07/utility/
    - OASIS draft 12 for secext
    - OASIS draft 12 for utility
- Specification: Web Services Security (WS-Security) Version 1.0 05 April 2002
- XML Encryption Syntax and Processing W3C Recommendation 10 December 2002
- XML-Signature Syntax and Processing W3C Recommendation 12 February 2002
- Web Services Security Addendum
- Web Services Security Core Specification Working Draft 01, 20 September 2002
- Web Services Security: SOAP Message Security Working Draft 13, Thursday, 01 May 2003
- Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC3280, April 2002
- OASIS Web Services Security Technical Committee

Samples
- Samples Gallery
- Samples Central. Samples and associated documentation for the following Web services components are available through the Samples Central page of the DeveloperWorks WebSphere Web site:
    - The IBM WebSphere UDDI Registry.
    - The Web Services Invocation Framework (WSIF).

Developing Web services applications

This topic explains how to develop a Web service based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification. Web services are structured in a service-oriented architecture (SOA) that makes integrating your business and e-commerce systems more flexible.

Before you develop the Web services you need to Set up a Web services development and unmanaged client execution environment . You do not have to set up a development environment if you are using Rational Application Developer.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

WebSphere Application Server uses Web services standards developed for the Java language under the Java Community Process (JCP). WebSphere Application Server follows these standards:
- SOAP Version 1.1
- WSDL Version 1.1
- Web Services for J2EE (JSR-109) Version 1.1
- Java API for XML-based remote procedure call (JAX-RPC) (JSR 101) Version 1.1
- SOAP with Attachments API for Java (SAAJ) Version 1.2

WebSphere Application Server provides extensions to the JSR-101 and JSR-109 programming models. See "Extensions to the JAX-RPC and Web Services for J2EE programming models" on page 399 for more information.

You can also use the Rational Application Developer graphical user interface development tools to develop Web services that integrate with WebSphere Application Server.

You can develop Web services in one of four ways:
1. Develop Web services using JavaBeans implementation.
2. Develop Web services using a stateless session enterprise bean.
3. Develop Web services with an existing WSDL file using JavaBeans implementation.
4. Develop Web services with an existing WSDL file using a stateless session enterprise bean.

You have developed a Web service.

Assemble the Web service.

This topic presents what you need to assemble a Web service and in what order you should assemble the parts, for example an enterprise archive (EAR) file.

Example: Developing a Web service from an EJB or JavaBeans implementation

This example takes you through the steps to develop a Web service from an Enterprise JavaBeans (EJB) or JavaBeans implementation. The development process is based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

1. **Select the enterprise bean or JavaBeans implementation that you want to enable as a Web service.**

   The implementation must meet the following Web Services for J2EE 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 service 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.

Review the API Documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

Artifacts used to develop Web services

With *development artifacts* you can develop an enterprise bean or a Java bean module into a Web service. This topic describes artifacts used to develop Web services that are based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

To create a Web service from an enterprise bean or a Java bean 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 WSDL 1.1 World-Wide Web Consortium (W3C) Note.

- **webservices.xml**

  The webservices.xml file contains the J2EE deployment descriptor of the Web servicespecifying how the Web service is implemented. The webservices.xml file is defined in the Web Services for J2EE specification available through Web services: Resources for learning

- **ibm-webservices-bnd.xmi**

  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 J2EE 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-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**

  This file contains WebSphere product-specific deployment information, such as security information.
- **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.

Mapping between Java language, WSDL and XML

This topic contains the mappings between the Java language and extensible Markup Language (XML) technologies, including XML Schema, Web Services Description Language (WSDL) and SOAP, supported by WebSphere Application Server. Most of these mappings are specified by the Java API for XML-based Remote Procedure Call (JAX-RPC) specification. Some mappings that are optional or unspecified in JAX-RPC are also supported.

References to the JAX-RPC specification throughout this topic. Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

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.

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:

```
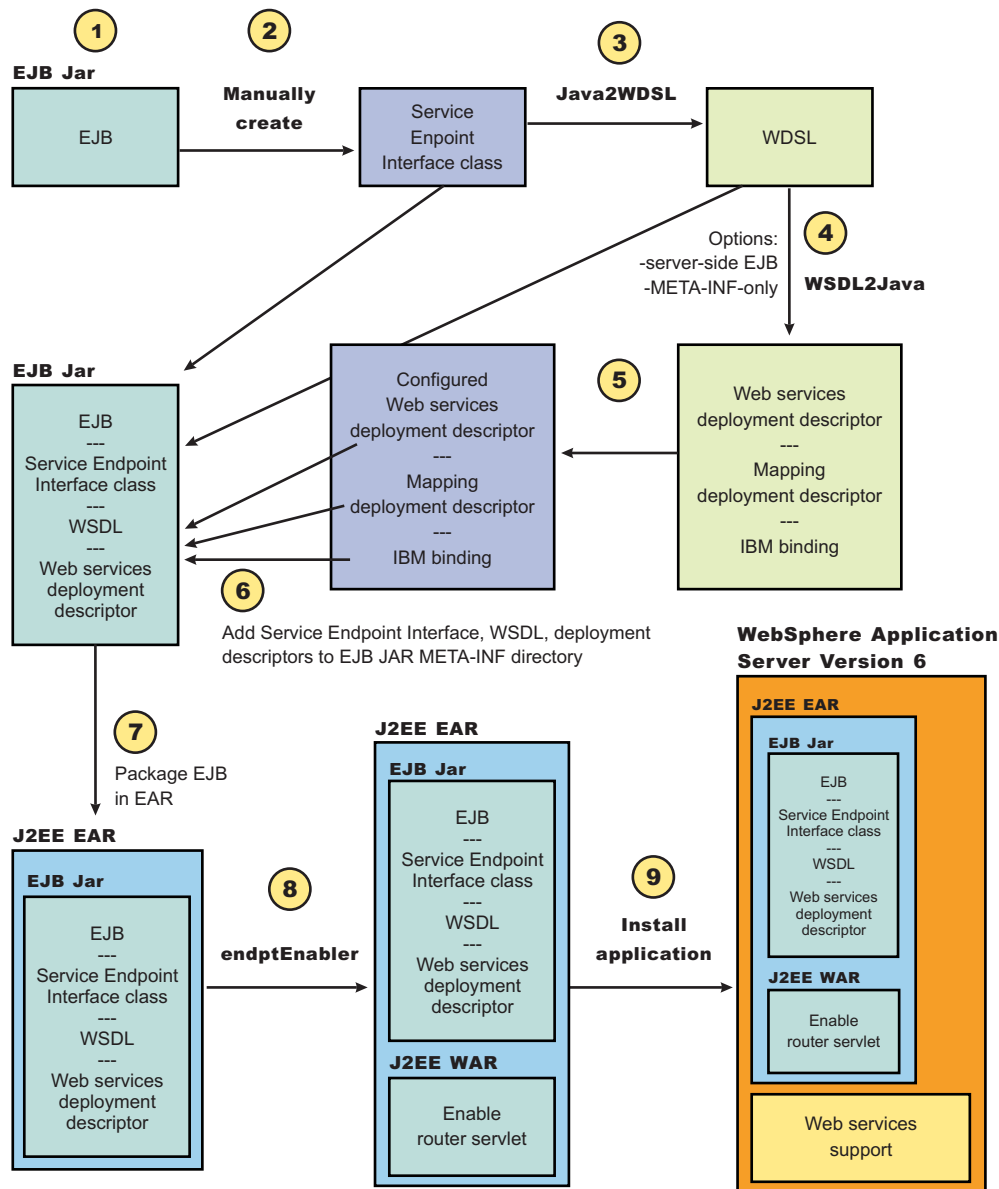public abstract class Base {
```

```
    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.

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 allowed 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</c>
</soap:body>
```

Extensions to the JAX-RPC and Web Services for J2EE programming models

WebSphere Application Server provides extensions to the Java API for XML-based RPC (JAX-RPC) and Web Services for Java 2 Platform, Enterprise Edition (J2EE) client programming models.

These extensions are defined as follows:

- The **REQUEST_SOAP_HEADERS** and **RESPONSE_SOAP_HEADERS** properties can be added to 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.

  To learn how to modify your client code to send or retrieve transport headers see Sending values from implicit SOAP headers or Receiving values from implicit SOAP headers.

- The **REQUEST_TRANSPORT_PROPERTIES** and **RESPONSE_TRANSPORT_PROPERTIES** properties can be added to a JAX-RPC client Stub to enable a Web services client to send or retrieve HTTP transport headers.

  To learn how to modify your client code to send or retrieve transport headers see Sending HTTP transport headers or Receiving HTTP transport headers. To learn more about these properties, see HTTP transport header properties best practices.

- Implementation-specific support for javax.xml.rpc.ServiceFactory.loadService() as described by the JAX-RPC specification. 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. You can review the JAX-RPC specification through Web services: Resources for learning.

  – **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.

- **CustomBinder interface**

WebSphere Application Server defines a CustomBinder interface that you can implement 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:
- 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 run time system and interacts with the Web services runtime using a SOAPElement. They are added to the run time by using custom binding providers. See the related links to learn more about the topics associated with the CustomBinder interface.

See the related links to learn more about the topics associated with these programming model extensions.

To review the documentation used for APIs and SPIs, see Reference: Generated API documentation. Follow the instructions in this topic that lead you to the API and SPI interfaces.

Review the specifications for the standards and APIs used in developing Web services.

Custom data binders:

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
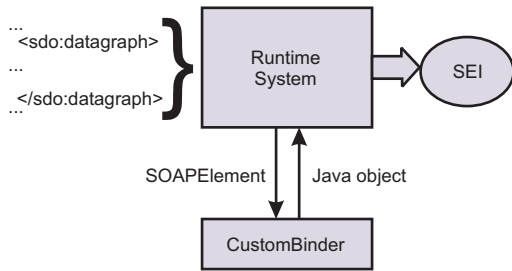<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sdo="commonj.sdo"
  targetNamespace="commonj.sdo">

  <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, see Custom binding providers. The topic Usage patterns for deploying custom data binders provides usage patterns and roles for deploying the custom data binders.

To review the documentation used for APIs and SPIs, see Reference: Generated API documentation. Follow the instructions in this topic that lead you to the API and SPI interfaces.

You can also review the specifications for the standards and APIs used in developing Web services.

Custom binding providers:

A *custom binding provider* is the packaging of custom data binder classes 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 would 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.

To review the documentation used for APIs and SPIs, see Reference: Generated API documentation. Follow the instructions in this topic that lead you to the API and SPI interfaces.

You can also review the specifications for the standards and APIs used in developing Web services.

CustomBinder interface:

WebSphere Application Server defines a CustomBinder interface that you can implement 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" on page 400. To learn how to plug your custom data binders into the **WSDL2Java** command-line tool for development, see Custom binder providers.

To review the documentation used for APIs and SPIs, see Reference: Generated API documentation. Follow the instructions in this topic that lead you to the API and SPI interfaces.

You can also review the specifications for the standards and APIs used in developing Web services.

Usage patterns for deploying custom data binders:

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 Java 2 Enterprise Edition (J2EE) 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 J2EE 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.

To review the documentation used for APIs and SPIs, see Reference: Generated API documentation. Follow the instructions in this topic that lead you to the API and SPI interfaces.

You can also review the specifications for the standards and APIs used in developing Web services.

Implicit SOAP Header property code example:

WebSphere Application Server provides extensions to the Java API for XML-based RPC (JAX-RPC) and Web Services for Java 2 Platform, Enterprise Edition (J2EE) 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.

To review the documentation used for APIs and SPIs, see Reference: Generated API documentation. Follow the instructions in this topic that lead you to the API and SPI interfaces.

Review the specifications for the standards and APIs used in developing Web services.

Sending values in implicit SOAP headers:

This task explains how to enable an existing 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.

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 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.

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.

The client application sets properties on the Stub or Call object to send and receive implicit SOAP headers. You can modify the client code as follows:
1. Create a java.util.HashMap.
2. Add an entry to the HashMap 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 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 are sent in the outgoing message.

   A `JAXRPCException` can occur if any of the following are true:
   - The HashMap contains a key that is not a QName or if the HashMap contains a value that is not a String or a SOAPElement.
   - The HashMap contains a key that represents a SOAP header that is declared protected by the owning component.

You have a Web service client that is configured to send implicit SOAP headers.

Receiving values from implicit SOAP headers:

This task explains how to enable an existing 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.

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 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.

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.

The client application sets properties on the Stub or Call object to send and receive implicit SOAP headers. You can modify the client code as follows:

1. Create a java.util.HashMap.
2. Add an entry to the HashMap 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.

   A `JAXRPCException` 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.

You have a Web service client that is able to receive values from implicit SOAP headers.

HTTP transport header properties best practices: The REQUEST_TRANSPORT_PROPERTIES property and RESPONSE_TRANSPORT_PROPERTIES property can be set on a Java API for XML-based RPC (JAX-RPC) client Stub to enable a Web services client to send or retrieve HTTP transport headers.

REQUEST_TRANSPORT_PROPERTIES best practices

Header values format

The header values format must be written in the following way:
- Each name=value pair 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. 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 HTTP headers, which correspond to the HashMap keys from the incoming HTTP response. An empty HashMap causes all of the HTTP headers and the associated values to be retrieved from the incoming HTTP response

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 a Web service a Web services client can be coded 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 HTTP transport headers:

This task explains how to enable an existing Web services client to send values in HTTP transport headers. By modifying your client code to send transport headers, you can send specific information within the HTTP transport headers of outgoing requests.

You need a Web services client that you can enable to send HTTP transport headers.

Sending transport headers is supported by Web services clients only, and over the HTTP transport only. The Web services client must call the Java API for XML-based RPC (JAX-RPC) APIs directly and not through any intermediary layers, such as a gateway-like function. Sending and retrieving HTTP transport headers on the Web services server-side is done through non-Web services APIs.

The client must set a property on the Stub to send values in HTTP transport headers. Once the property is set, the values are set in all the HTTP requests for subsequent remote method invocations against the Stub until the associated property is set to `null` or the Stub is discarded. To send values in the HTTP transport headers on outbound requests, modify the client code as follows:

1. Create a java.util.HashMap that contains the HTTP header identifiers.
2. Add an entry to the HashMap for each header that you want the client to send.
   a. Set the HashMap entry key to a string that exactly matches the HTTP header identifier. The header identifier can be one that is defined for HTTP, such as Cookie, or it can be user-defined, such as MyHTTPHeader. Certain header identifiers are processed in a special manner, but no other checks are made as to the header identifier value. To learn more about the header identifiers that have special consideration, see HTTP transport header properties best practices.

      Common header identifier string constants, such as HTTP_HEADER_SET_COOKIE can be found in the com.ibm.websphere.webservices.Constants class. The HashMap entry value does not need to be set; it is ignored. An empty HashMap (one that is non-null, but does not contain keys), causes values from all headers in the HTTP response to be received.
   b. Set the HashMap value to a string that contains the header value to send in the HTTP header.
3. Set the HashMap on the Stub by using the property com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES. 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 the HTTP transport header properties see HTTP transport header properties best practices.
4. Issue the remote method calls against the Stub. The headers and the associated values from the HashMap are added to the outgoing HTTP request for each method invocation.

   A `JAXRPCException` can occur if the property is not set correctly. The following requirements must be met:
   - The property value set on the Stub 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.

You have a Web service client that is configured to send HTTP transport headers.

Retrieving HTTP transport headers:

This task explains how to enable an existing Web services client to retrieve values from HTTP transport headers. By modifying your client code, you can retrieve information from incoming HTTP headers responses

You need a Web services client that you can enable to retrieve HTTP transport headers.

Retrieving transport headers is supported only by Web services clients, and only over the HTTP transport. The Web services client must call the Java API for XML-based RPC (JAX-RPC) APIs directly and not through any intermediary layers, such as a gateway-like function. Sending and retrieving HTTP transport headers on the Web services server-side is done through non-Web services APIs.

The client must set a property on the Stub in order to retrieve values from the HTTP transport headers. Once the property is set, values are read from HTTP responses for the subsequent method invocations against that Stub until the associated property is set to `null` or the Stub is discarded. To retrieve values from the HTTP transport headers on inbound responses, modify the client code as follows:

1. Create a java.util.HashMap that contains the HTTP header identifier values to retrieve and the values for those headers on responses.
2. Add an entry to the HashMap for each header that you want the client to retrieve a value from.
   a. Set the HashMap entry key to a string that exactly matches the HTTP header identifier. The header identifier can be defined for HTTP, such as Cookie, or it can be user-defined, such as MyHTTPHeader. Certain header identifiers are processed in a special manner, but no other checks are made to confirm the header identifier value. To learn more about the header identifiers that have special consideration, see HTTP transport header properties best practices. Common header identifier string constants, such as HTTP_HEADER_SET_COOKIE can be found 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 keys, causes values from all headers in the HTTP response to be received.
3. Set the HashMap entry on the Stub 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 the properties used, see HTTP transport header properties.
4. Issue remote method calls against the Stub. The values from the HTTP response headers are placed in the HashMap.

   You might experience API usage errors that result in a JAXRPCException. The following items are checked for during invocation and cause an exception to be thrown if there is an error:
   - The property value that is set on the Stub is either `null` or a HashMap.
   - All the HashMap keys are not non-null and an instance of a String.

You have a Web service that is able to receive HTTP transport headers.

Java2WSDL command
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 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.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

WSDL2Java command

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 2 Platform, Enterprise Edition (J2EE) 1.1 specification defines deployment descriptors that deploy a Web service in a J2EE environment. The **WSDL2Java** command is run against the WSDL file to create Java APIs and deployment descriptor templates according to these specifications.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

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 *j2ee role*

    Specifies the J2EE 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 *j2ee-container*

    Indicates the J2EE 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 J2EE 1.1 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) 1.2 specification.

    The JAX-RPC specification defines 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.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

- -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.
- -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.

Using HTTP to transport Web services requests

This task leads you into developing an HTTP accessible Web service when you already have a JavaBean object to enable as a Web service.

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
```

WebSphere 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 2 platform Enterprise Edition (J2EE) 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.

You have a JavaBean object that uses HTTP to transport Web services client requests.

Publish the WSDL file.

Configuring HTTP outbound transport level security with the administrative console

This topic explains how to configure HTTP outbound transport level security with the administrative console.

This task is one of several ways that you can configure the HTTP outbound transport level security for a Web service acting as a client to another Web service server. You can also configure the HTTP outbound transport level security with an assembly tool or by using the Java properties. If you do not configure the HTTP outbound transport level security, the Web services runtime defers to the Java 2 Platform, Enterprise Edition (J2EE) security runtime in the WebSphere product for an effective Secure Sockets Layer (SSL) configuration. If there is no SSL configuration with the J2EE security runtime in the WebSphere product, the Java Secure Socket Extension (JSSE) system properties are used.

If you choose to configure the HTTP outbound transport level security with the administrative console or an assembly tool, the Web services security binding information is modified. You can use the administrative console to configure the Web services client security bindings if you have deployed or installed the Web services application into WebSphere Application Server. If you have not installed the Web services application, you can configure the HTTP SSL configuration with an assembly tool. This task assumes that you have deployed the Web services application into the WebSphere product.

If you configure the HTTP outbound transport level security using the standard Java properties for JSSE, the properties are configured as system properties. The configuration specified in the binding takes precedence over the Java properties. However, the configurations that are specified by the J2EE security programming model , or that are associated the Dynamic selection, have higher precedence.

Review the topic Secure communications using Secure Sockets Layer for more information.

Configure the HTTP outbound transport level security with the following steps provided in this task section.

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* > **Web Services Client Security Bindings**.
3. Click **HTTP SSL Configuration** to access the HTTP SSL configuration panel. Select the **Centrally-managed** radio button so that the system runtime chooses the SSL configuration that is based on the current context. Select the **Specific to this Web service port** radio button if you want to choose the SSL configuration in the HTTP SSL configuration drop down box.

You have configured the HTTP outbound transport level security for a Web service acting as a client to another Web service with the administrative console.

HTTP SSL Configuration collection:

Use this page to configure transport-level Secure Sockets Layer (SSL) security. You can use this configuration when a Web service is a client to another Web service.

You can use transport-level security to enable HTTP SSL (or HTTPS). Transport-level security can be enabled or disabled independently from message-level security. Because transport-level security provides minimal security, use message-level security when security is essential to the Web service application.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >** *application_instance*.
2. Click **Manage modules >** *URI_file_name* **> Web Services: Client Security Bindings**.
3. Under HTTP SSL Configuration, click **Edit**.

SSL configuration: Select the **Centrally-managed** radio button so that the system runtime chooses the SSL configuration that is based on the current context. Select the **Specific to this Web service port** radio button if you want to choose the SSL configuration in the **HTTP SSL configuration** drop down box.

HTTP SSL configuration: The **HTTP SSL configuration** drop down box lists the SSL configurations used with the HTTP transport for a port. Use this drop down box if you want to select the SSL configuration rather than using the SSL configuration that the runtime automatically selects. To use the drop down box, select the **Specific to the Web service port** radio button that is located in the **SSL configuration** field. After you select the radio button, you can click the drop down box to view and select an SSL configuration.

Configuring HTTP outbound transport level security with an assembly tool
This topic explains how to configure the HTTP outbound transport level security with an assembly tool.

You can configure HTTP outbound transport level security with assembly tools provided with WebSphere Application Server.

This task is one of several ways that you can configure the HTTP outbound transport level security for a Web Service acting as a client to another Web service server. You can also configure the HTTP outbound transport level security with the administrative console or by using the Java properties. If you do not configure the HTTP outbound transport level security, the Web services runtime defers to the Java 2 Platform, Enterprise Edition (J2EE) security runtime in the WebSphere product for an effective Secure Sockets Layer (SSL) configuration. If there is no SSL configuration with the J2EE security runtime in the WebSphere product, the Java Secure Socket Extension (JSSE) system properties are used.

If you configure the HTTP outbound transport level security with assembly tool or with the administrative console, the Web services security binding information is modified. If you have not yet installed the Web services application into WebSphere Application Server, you can configure the HTTP SSL configuration with an assembly tool. This task assumes that you have not deployed the Web services application into the WebSphere product.

If you configure the HTTP outbound transport level security using the standard Java properties for JSSE, the properties are configured as system properties. The configuration that is specified in the binding takes precedence over the Java properties. However, the configurations that are specified by the J2EE security programming model, or are associated with the Dynamic selection, have a higher precedence.

Review the topic Secure communications using Secure Sockets Layer for more information.

Configure the HTTP outbound transport level security with the following steps provided in this task section.

1. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.

2. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.

3. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.

4. Configure the HTTP outbound transport level security. See ″Enabling Web service endpoints″ in the Application Server Toolkit documentation for more information.

You have configured the HTTP outbound transport level security for a Web Service acting as a client to another Web service with an assembly tool.

Configuring HTTP outbound transport-level security using Java properties

This topic explains how to configure the HTTP outbound transport level security for a Web service using Java properties

This task is one of three ways that you can configure HTTP outbound transport-level security for a Web service that is acting as a client to another Web service. You can also configure the HTTP outbound transport level security with the administrative console or an assembly tool. However, you can also use this task to configure the HTTP outbound transport-level security for a Web service client.

If you choose to configure the HTTP outbound transport-level security with the administrative console or an assembly tool, the Web services security binding information is modified.

If you configure the HTTP outbound transport-level security using Java properties, the properties are configured as system properties. However, the configuration specified in the binding takes precedence over the Java properties.

You can configure the HTTP outbound transport-level security using WebSphere SSL properties or JSSE SSL properties. However, the WebSphere SSL properties take precedence over the JSSE SSL properties.

Configure the HTTP outbound transport-level security with the following steps provided in this task section.

1. Create a property file that includes the following properties:
```
com.ibm.ssl.protocol
com.ibm.ssl.keyStoreType
com.ibm.ssl.keyStore
com.ibm.ssl.keyStorePassword
com.ibm.ssl.trustStoreType
com.ibm.ssl.trustStore
com.ibm.ssl.trustStorePassword
```

2. Set the `com.ibm.webservices.sslConfigURL` Java system property to the absolute path of the created property file. If no WebSphere SSL properties are defined, the JSSE SSL properties are used. Set the JSSE SSL properties as JVM custom properties. See "Secure transports with JSSE and JCE programming interfaces" on page 964 for more information about setting the JSSE SSL properties.

You have configured the HTTP outbound transport-level security for a Web service acting as a client to another Web service.

Transport level security

Transport level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP.

Transport level security can be used to secure Web services messages. However, transport-level security functionality is independent from functionality that is provided by WS-Security or HTTP Basic Authentication.

SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections. To run with HTTPS, the service port address must be in the form `https://`.

The integrity and confidentiality of transport data, including SOAP messages and HTTP basic authentication, is confirmed when you use SSL and TLS.

Web services applications can also use Federal Information Processing Standard (FIPS) approved ciphers for more secure TLS connections.

WebSphere Application Server uses the Java Secure Sockets Extension (JSSE) package to support SSL and TLS.

HTTP basic authentication

HTTP basic authentication uses a user name and password to authenticate a service client to a secure endpoint.

WebSphere Application Server can have several resources, including Web services, protected by a Java 2 Platform, Enterprise Edition (J2EE) security model.

HTTP basic authentication is orthogonal to the security support provided by WS-Security or HTTP Secure Sockets Layer (SSL) configuration.

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint using HTTP basic authentication. The basic authentication is encoded in the HTTP request that carries the SOAP message. When the application server receives the HTTP request, the user name and password are retrieved and verified using the authentication mechanism specific to the server.

Although the basic authentication data is base64-encoded, sending data over HTTPS is recommended. The integrity and confidentiality of the data can be protected by the SSL protocol.

In some cases, a firewall is present using a pass-thru HTTP proxy server. The HTTP proxy server forwards the basic authentication data into the J2EE application server. The proxy server can also be protected. Applications can specify the proxy data by setting properties in a stub object.

Configuring HTTP basic authentication with the administrative console

This topic explains how to configure HTTP basic authentication with the administrative console.

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with an assembly tool or by modifying the HTTP properties programmatically.

If you choose to configure HTTP basic authentication with the administrative console or an assembly tool, the Web services security binding information is modified. You can use the administrative console to configure HTTP basic authentication if you have deployed or installed the Web services application into

WebSphere Application Server. If you have not installed the Web services application, then you can configure the security bindings with an assembly tool. This task assumes that you have deployed the Web services application into the WebSphere product.

If you configure HTTP basic authentication programmatically, the properties are configured in the Stub or Call instance. The values set programmatically take precedence over the values defined in the binding. However, you only can programmatically configure HTTP proxy authentication.

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

Configure HTTP basic authentication with the following steps provided in this task section.

Open the administrative console.
1. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* > **Web services: Client security bindings**.
2. Click **HTTP Basic Authentication** to access the HTTP basic authentication panel. Enter the values in the HTTP Basic Authentication panel.

You have configured the HTTP basic authentication.

HTTP basic authentication collection:

Use this page to specify a user name and password for transport-level basic authentication security for this port. You can use this configuration when a Web service is a client to another Web service.

You can use transport-level security to enable basic authentication. Transport-level security can be enabled or disabled independently from message-level security. Because transport-level security provides minimal security, use message-level security when security is essential to the Web service application.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >** *application_instance*.
2. Click **Manage modules >** *URI_file_name* **> Web Services: Client Security Bindings**.
3. Under HTTP basic authentication, click **Edit**.

Basic authentication ID:

The user name for the HTTP basic authentication for this port is set in this field.

Basic authentication password:

The password for the HTTP basic authentication for this port is set in this field.

Configuring HTTP basic authentication with an assembly tool
This topic explains how to configure HTTP basic authentication with an assembly tool.

You can configure HTTP basic authentication with assembly tools provided with WebSphere Application Server.

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with the administrative console or by modifying the HTTP properties programmatically.

If you choose to configure the HTTP basic authentication with an assembly tool or with the administrative console , the Web services security binding information is modified. You can use an assembly tool to configure HTTP basic authentication before you deploy or install the Web services application into WebSphere Application Server. This task assumes that you have not deployed the Web services application into the WebSphere product.

If you configure HTTP basic authentication programmatically, the properties are configured in the Stub or Call instance. The values set programmatically take precedence over the values defined in the binding. However, you only can programmatically configure HTTP proxy authentication.

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

To configure HTTP basic authentication, use the WebSphere Application Server tools to modify the binding information.
1. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
2. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
3. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.
4. Configure the HTTP basic authentication in the Web Services Client Port Binding page for a Web service or a Web service client. The Web Services Client Port Binding page is available after double-clicking the client deployment descriptor file.

Configuring HTTP basic authentication programmatically

This topic explains how to configure HTTP basic authentication by programmatically modifying HTTP properties.

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with an assembly tool or with the administrative console.

If you programmatically configure HTTP basic authentication, the properties are configured in the Stub or Call instance. If you choose to configure HTTP basic authentication with the administrative console or an assembly tool, the Web services security binding information is modified. The values that are set programmatically take precedence over the values defined in the binding. However, you can only configure HTTP proxy authentication programmatically.

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

Configure HTTP basic authentication programmatically with the following steps provided in this task section.
1. Set the properties in the Stub or Call instance for a Web service or a Web service client You can set the following properties:
   ```
   javax.xml.rpc.Call.USERNAME_PROPERTY
   javax.xml.rpc.Call.PASSWORD_PROPERTY
   javax.xml.rpc.Stub.USERNAME_PROPERTY
   javax.xml.rpc.Stub.PASSWORD_PROPERTY
   ```
2. Set the properties in the Stub or Call instance to configure the HTTP proxy authentication.

a. You can set the following properties for HTTP:

```
com.ibm.wsspi.webservices.HTTP_PROXYHOST_PROPERTY
com.ibm.wsspi.webservices.HTTP_PROXYPORT_PROPERTY
com.ibm.wsspi.webservices.HTTP_PROXYUSER_PROPERTY
com.ibm.wsspi.webservices.HTTP_PROXYPASSWORD_PROPERTY
```

3. You can set the following properties for HTTPS:

```
com.ibm.wsspi.webservices.HTTPS_PROXYHOST_PROPERTY
com.ibm.wsspi.webservices.HTTPS_PROXYPORT_PROPERTY
com.ibm.wsspi.webservices.HTTPS_PROXYUSER_PROPERTY
com.ibm.wsspi.webservices.HTTPS_PROXYPASSWORD_PROPERTY
```

Configuring additional HTTP transport properties using the JVM custom property panel in the administrative console

This topic explains how to configure additional HTTP transport properties with the JVM custom properties panel in the administrative console.

This task is one of three ways that you can configure additional HTTP transport properties for a Web Service acting as a client to another Web service. You can also configure the additional HTTP transport properties in the following ways:

- Configure the properties with an assembly tool
- Configure the properties using the **wsadmin** command-line tool

If you want to programmatically configure the properties using the Java API XML-based Remote Procedure Call (JAX-RPC) programming model, review the JAX-RPC specification that is available through Web services: Resources for learning.

See Additional HTTP transport properties for Web services applications for more information about the following properties that you can configure:

- com.ibm.websphere.webservices.http.requestContentEncoding
- com.ibm.websphere.webservices.http.responseContentEncoding
- com.ibm.websphere.webservices.http.connectionKeepAlive
- com.ibm.websphere.webservices.http.requestResendEnabled
- http.proxyHost
- http.proxyPort
- https.proxyHost
- https.proxyPort

These additional properties are configured for Web services applications that use the HTTP protocol. The properties affect the content encoding of the message in the HTTP request, the HTTP response, the HTTP connection persistence and the behavior of an HTTP request that is resent after a `java.net.ConnectException` error occurs when there is a read time-out.

Configure the additional HTTP properties with the administrative console with the following steps provided in this task section:

1. Open the administrative console.
   a. Click **Servers** > **Application Servers** > *server* > **Java and Process Management** > **Process Definition** > **Java Virtual Machine** > **Custom Properties**.
2. (Optional) If the property is not listed, create a new property name.
3. Enter the name and value.
4. (Optional) Accept the redirection of the HTTP request to a different URI in HTTPS.

A redirection of the HTTP request to a different URI in HTTPS can occur if the transport guarantee of CONFIDENTIAL or INTEGRAL is configured in the application. To accept the redirection, you can do either of the following tasks:

- Set the `com.ibm.ws.webservices.HttpRedirectEnabled` Java system property to `true`.
- Programmatically set the `com.ibm.wsspi.webservices.Constants.HTTP_REDIRECT_ENABLED` property to `true` in the stub or call object before invoking the service.

You have configured HTTP transport properties for a Web services application.

Configuring additional HTTP transport properties with an assembly tool

This topic explains how to configure additional HTTP transport properties with an assembly tool. The assembly tool is used to configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor binding file.

You can configure additional HTTP transport properties with assembly tools provided with WebSphere Application Server.

This task is one of three ways that you can configure additional HTTP transport properties for a Web Service acting as a client to another Web service. You can also configure the additional HTTP transport properties in the following ways:

- Configuring additional HTTP transport properties using the JVM custom property panel in the administrative console
- Configure the properties using the **wsadmin** command-line tool.

If you want to programmatically configure the properties using the Java API XML-based Remote Procedure Call (JAX-RPC) programming model, review the JAX-RPC specification that is available through Web services: Resources for learning.

See Additional HTTP transport properties for Web services applications for more information about the following properties that you can configure:

- com.ibm.websphere.webservices.http.requestContentEncoding
- com.ibm.websphere.webservices.http.responseContentEncoding
- com.ibm.websphere.webservices.http.connectionKeepAlive
- com.ibm.websphere.webservices.http.requestResendEnabled
- http.proxyHost
- http.proxyPort
- https.proxyHost
- https.proxyPort

These additional properties are configured for Web services applications that use the HTTP protocol. The properties affect the content encoding of the message in the HTTP request, the HTTP response, the HTTP connection persistence and the behavior of an HTTP request that is resent after a `java.net.ConnectException` error occurs when there is a read time-out.

Configure the additional HTTP properties with an assembly tool with the following steps provided in this task section:

1. The assembly tools, Application Server Toolkit (AST) and Rational Web Developer, provide a graphical interface for developing code artifacts, assembling the code artifacts into various archives (modules) and configuring related Java 2 Platform, Enterprise Edition (J2EE) Version 1.2, 1.3 or 1.4 compliant deployment descriptors.
2. Start an assembly tool. See "Starting WebSphere Application Server Toolkit" in the Application Server Toolkit documentation for more information.

3. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.

4. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.

5. Configure the additional HTTP transport properties. Create and specify the name/value pair in the **Web Services Client Port Binding** page for a Web service client. The Web Services Client Port Binding page is available after double-clicking the client deployment descriptor file.

You have configured additional HTTP transport properties for a Web services application.

Configuring additional HTTP transport properties using the wsadmin command-line tool

This topic explains how to configure additional HTTP transport properties with the **wsadmin** command-line tool.

The WebSphere Application Server wsadmin tool provides the ability to run scripts. You can use the wsadmin tool to manage a WebSphere Application Server installation, as well as configuration, application deployment, and server run-time operations. The WebSphere Application Server only supports the Jacl and Jython scripting languages. For more information about the wsadmin tool options, review Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands

This task is one of three ways that you can configure additional HTTP transport properties for a Web Service acting as a client to another Web service. You can also configure the additional HTTP transport properties in the following ways:

- Configure the properties with an assembly tool
- Configuring additional HTTP transport properties using the JVM custom property panel in the administrative console

If you want to programmatically configure the properties using the Java API XML-based Remote Procedure Call (JAX-RPC) programming model, review the JAX-RPC specification that is available through Web services: Resources for learning.

See Additional HTTP transport properties for Web services applications for more information about the following properties that you can configure:

- com.ibm.websphere.webservices.http.requestContentEncoding
- com.ibm.websphere.webservices.http.responseContentEncoding
- com.ibm.websphere.webservices.http.connectionKeepAlive
- com.ibm.websphere.webservices.http.requestResendEnabled
- http.proxyHost
- http.proxyPort
- https.proxyHost
- https.proxyPort

These additional properties are configured for Web services applications that use the HTTP protocol. The properties affect the content encoding of the message in the HTTP request, the HTTP response, the HTTP connection persistence and the behavior of an HTTP request that is resent after a `java.net.ConnectException` error occurs when there is a read time-out.

Configure the additional HTTP properties with the wsadmin tool by following steps provided in this task section:

1. Launch a scripting command.

2. At the **wsadmin** command prompt, enter the command syntax. You can use install, installInteractive, edit, editInteractive, update, and updateInteractive commands.

3. If you are configuring the `com.ibm.websphere.webservices.http.responseContentEncoding` property, use the **WebServicesServerCustomProperty** command option.

4. Configure all other properties using the **WebServicesClientCustomProperty** command option.

5. Save the configuration changes with the **$AdminConfig save** command.

You have configured HTTP transport properties for a Web services application.

The following illustrates an example of the Jython script syntax:

```
AdminApp.edit ( 'PlantsByWebSphere', '[ -WebServicesClientCustomProperty [[PlantsByWebSphere.war ""
service/FrontGate_SEIService FrontGate http.proxyHost+http.proxyPort myhost+80]]]')
AdminConfig.save()

AdminApp.edit ( 'WebServicesSamples', '[ -WebServicesServerCustomProperty
[[AddressBookW2JE.jarAddressBookService AddressBook http.proxyHost+http.proxyPort myhost+80]]]')
AdminConfig.save()
```

The following illustrates an example of the Jacly script syntax:

```
$AdminApp edit PlantsByWebSphere {  -WebServicesClientCustomProperty {{PlantsByWebSphere.war {}
service/FrontGate_SEIService FrontGate  http.proxyHost+http.proxyPort myhost+80 }}}
$AdminConfig save

$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty {{AddressBookW2JE.jar
AddressBookService AddressBook http.proxyHost+http.proxyPort myhost+80}}}
$AdminConfig save
```

To convert these examples from **edit** to **install**, add `.ear` to form a file name, and add any extra keywords for deployment, like `-usedefaultbindings` and `-deployejb`.

Additional HTTP transport properties for Web services applications

This topic defines additional HTTP transport properties for Web services applications. The additional properties can be used to manage the connection pool for HTTP outbound connections, configure the content encoding of the HTTP message, enable HTTP persistent connection, and resend the HTTP request when a timeout occurs.

Properties that manage the connection pool for Web services HTTP outbound connections

For information about how to configure these properties see Configuring additional HTTP transport properties using the administrative console.

Note: These properties can only be configured as JVM custom properties.

Establishing a connection is an expensive operation. Connection pooling improves performance by avoiding the overhead of creating and disconnecting connections. When an application invokes a Web service over an HTTP transport, the HTTP outbound connector for the Web service locates and uses an existing connection from a pool of connections. When the response is received, the connector returns the connection to the connection pool for reuse. The overhead to create and disconnect the connection is avoided.

The following properties are only configured as JVM custom properties that manage the connection pool for HTTP outbound connections for Web services applications:

- **com.ibm.websphere.webservices.http.connectionTimeout**

  This property specifies the interval, in seconds, that a connection request times-out and the `WebServicesFault( "Connection timed out" )` error occurs. You can configure the property only as a JVM custom property. The value affects all of the HTTP connection requests made by the HTTP outbound connector. The wait time is needed when the maximum number of connections in the connection pool is reached. For example, if the property is set to 300 and the maximum number of connections is reached, the connector waits for 300 seconds until a connection is available. After 300 seconds, the `WebServicesFault( "Connection timed out" )` error occurs if a connection is not available. If the property is set to 0 (zero), the connector waits until a connection is available.

  If the `WebServicesFault( "Connection timed out")` error occurs in the application, set the com.ibm.websphere.webservices.http.connectionTimeout property value higher. Also, review the application usage. If the com.ibm.websphere.webservices.http.maxConnection property value is set to 0 (zero), and is enabled for an unlimited number of connections, the com.ibm.websphere.webservices.http.connectionTimeout property value is ignored.

Data type	Integer
Units	Seconds
Default	300
Range	0 (zero) to the maximum integer

- **com.ibm.websphere.webservices.http.maxConnection**

  This property specifies the maximum number of connections that are created in the HTTP outbound connector connection pool. You can configure the property only as a JVM custom property. It affects all of the Web services HTTP connections that are made within one JVM. When the maximum number of connections is reached, no new connection are created and the HTTP connector waits for a current connection to return to the connection pool. If the HTTP connector does not wait for a current connection because of a connection request timeout, the `WebServicesFault( "Connection timed out" )` error occurs. For example, if the property is set to 5, and there are 5 connections in use, the HTTP connector waits for the specified time set in the com.ibm.websphere.webservices.http.connectionTimeout property for a connection to become available.

  If the property is set to 0 (zero), the com.ibm.websphere.webservices.http.connectionTimeout property is ignored. The connector attempts to create as many connections allowed by the system.

Data type	Integer
Default	50
Range	0 (zero) to the maximum integer

- **com.ibm.websphere.webservices.http.connectionPoolCleanUp**

  This property specifies the interval, in seconds, between runs of the connection pool maintenance thread. You can configure the property only as a JVM custom property. This property affects all HTTP connections for Web Services made within one JVM. For example, if the property is set to 180, the pool maintenance thread runs every 180 seconds. When the pool maintenance thread runs, the connector discards any connections remaining idle for longer than the time set in the com.ibm.websphere.webservices.http.connectionIdleTimeout property.

Data type	Integer
Units	Seconds
Default	180
Range	0 (zero) to the maximum integer

- **com.ibm.websphere.webservices.http.connectionIdleTimeout**

  This property specifies the interval, in seconds, after an idle connection is discarded. You can configure the property only as a JVM custom property. For example, if the property is set to 120, the pool

maintenance thread discards any connection that remains idle for 2 minutes. This property affects all Web services HTTP connections made within one JVM.

Data type	Integer
Units	Seconds
Default	5
Range	0 (zero) to the maximum integer

Additional HTTP transport properties

Additional HTTP transport properties can also be configured for Web services applications.

For more information about how to configure these properties see Configuring additional HTTP transport properties using wsadmin, and Configuring additional HTTP transport properties using an assembly tool.

The following are additional HTTP transport properties that can be configured:

* **com.ibm.websphere.webservices.http.requestContentEncoding**

   This property specifies the type of encoding to use in the message of each HTTP outbound request. Supported encoding formats follow the HTTP 1.1 protocol specification including gzip, x-gzip, and deflate. If this property is configured, the headers ″Content-Encoding″ and ″Accept-Encoding″ in the HTTP request are also set to the same value. For example, if the property is set to gzip, the headers become Content-Encoding: gzip and Accept-Encoding: gzip. However, if the property is not set, the HTTP request message is not encoded. The default is no encoding.

   You should check if the target Web server is capable of decoding the configured coding format. For example, if the property is set to gzip, the target Web server must also support the gzip encoding. Otherwise, a failure can occur and a status code of `415 Unsupported Media Type` might display.

   The compress encoding format is not supported and x-gzip encoding is equivalent to gzip encoding.

Data type	String
Valid values	gzip, x-gzip, and deflate

* **com.ibm.websphere.webservices.http.responseContentEncoding**

   This property specifies the type of encoding to be used in the message of each HTTP response. Supported encoding formats follow the HTTP 1.1 protocol specification including gzip, x-gzip, and deflate. If this property is configured, the headers ″Content-Encoding″ in the HTTP response is set to the same value. If the property is not set, the HTTP response message content is not encoded. The default value is `no encoding`.

   If the property is set, the request client must also support the same encoding. Otherwise, a failure can occur and a `WebServicesFault()` error displays.

   The compress encoding format is not supported and x-gzip encoding is equivalent to gzip encoding.

*

Data type	String
Valid values	gzip, x-gzip, or deflate

* **com.ibm.websphere.webservices.http.connectionKeepAlive**

   This property specifies whether the connector should maintain a live or persistent HTTP connection. If the property is set to `true`, the connector keeps the connection in the connection pool and reuses the connection for subsequent HTTP requests. However, the connection is closed if syncTimeout(Read timeout) is reached or the server has dropped the connection. Also, an idle connection is closed by the pool maintenance thread if the idle time has passed the connection idle time-out. If the property is set to `false`, the connection is closed after the HTTP request is sent. If a new request is ready to send and

the connection does not exist, the HTTP connector creates one.

Data type	String
Default	True
Valid values	True, false

- **com.ibm.websphere.webservices.http.requestResendEnabled**

  This property tells the HTTP connector to resend the SOAP message over HTTP request after a `java.net.ConnectException: read timed out` error is logged. The `java.net.ConnectException` is caused by a socket time-out, or when a server shuts down while the request is being sent. If the property is enabled, the connector tries to reconnect one time only and resends the same SOAP message over HTTP. Otherwise, the connector stops sending the SOAP message and a `WebServicesFault` error is logged.

  Problems can occur with the application this property is enabled. The HTTP request that is resent can be received twice by the server and can cause an unexpected result.

Data type	String
Default	False
Valid values	True, false

- **http.proxyHost**

  This property specifies the host name of an HTTP proxy.

Data type	String

- **http.proxyPort**

  This property specifies the port of an HTTP proxy.

Data type	String

- **https.proxyHost**

  This property specifies the host name of an HTTPS proxy.

Data type	String

- **https.proxyPort**

  This property specifies the port of an HTTPS proxy.

Data type	String

Using the Java Message Service API to transport Web services requests

WebSphere Application Server supports use of the Java Message Service (JMS) API to transport Web services requests, as an alternative to HTTP transport. By using the JMS transport, your Web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections. One-way and synchronous two-way requests are supported.

A Web service must be implemented as an enterprise bean for accessibility through the JMS transport.

Review the API documentation for a complete list of API's. You can also review several articles about the development of Web services at Web services: Resources for learning.

The benefits of using JMS include:
- Reliable messaging for 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.

Perform this task after you have developed or implemented a Web service. This task explains how to configure the Web service to use JMS to transport the requests.

To configure a Web service to use JMS as a transport:

1. Add a JMS binding and a SOAP address to the Web Services Description Language (WSDL) file.

   The WSDL file of a Web service must include a JMS binding and a SOAP address, which specifies a JMS endpoint URL string, for accessibility on the JMS transport. A JMS binding is a wsdl:binding element that contains a wsdlsoap:binding element whose transport attribute ends in `soap/jms`, rather than the typical `soap/http` value.

   In addition to the JMS binding, a wsdl:port element referencing the JMS 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 a JMS endpoint URL string.

   The specification of the actual JMS endpoint URL string can be deferred until you configure endpoint URL information for JMS bindings. As you develop Web services, a placeholder such as `file:/unspecified_location` can be used for the endpoint URL string.

2. Decide the names and the types of the JMS objects that your application uses.

   Before your application can be installed, you need to:

   a. Decide whether your Web service receives requests from a queue or a topic.

   b. Decide whether to use a secure destination, like a queue or topic, or a nonsecure destination.

   c. Decide the names for your destination, connection factory and listener port.

      The following list provides examples of the names that can be used for the StockQuote Web service:
      - **Queue:** StockQuote_Q (Java Naming and Directory Interface (JNDI) name: `jms/StockQuote_Q`)
      - **Connection factory:** StockQuote_CF (JNDI name: `jms/StockQuote_CF`)
      - **Listener port:** StockQuoteEJB_ListenerPort

3. 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 various ways to administer JMS resources depending on what type of JMS provider is being used. See Using JMS resources of a generic provider for more information.

4. Add the JMS endpoints to your enterprise archive (EAR) file using the **endptEnabler** command. You must run the **endptEnabler** command to add a JMS endpoint or a router module for each enterprise bean Java archive (JAR) file that is enabled for Web services and is contained in the EAR file. By default, the **endptEnabler** command adds only HTTP endpoints, but the **-transport jms** command option can be used to request the addition of JMS endpoints.

5. Configure security for the JMS connection.

   For a secure JMS connection, add the basicAuth assembly property to the `ibm-webservicesclient-bnd.xmi` deployment descriptor file. Set the user ID and password attributes.

   If the basicAuth property is not provided in the `ibm-webservicesclient-bnd.xmi` deployment descriptor file, the JMS connection can be rejected, depending on the security configuration of the JMS provider.

6. 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 for the enterprise bean JAR file message-driven bean (MDB) listener to use for sending reply messages.

If your Web service contains two-way operations, the MDB listener that is defined inside the JMS endpoint added by **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 queue connection factory for the MDB listener to use for that Web service. You might want to use the same connection factory that you defined for use by clients in step 2.

- The name of the listener port for the MDB listener to use.

A listener port 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 listener port so that messages from the queue or topic are properly delivered to the MDB. During deployment, you can modify the name of the listener port that is associated with each MDB listener. The listener port name contained in the input EAR file is displayed as a default value. If you specify the correct listener port name to the **endptEnabler** command, you can accept the default value. Otherwise, enter the correct listener port name.

Hint: By default, the **endptEnabler** command produces listener port names of the form `<ejb-jar-name>_ListenerPort`. To simplify this step, define the listener ports that follow this naming convention during step 2.

7. Configure endpoint URL information for JMS bindings.

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 published WSDL file can be used by clients that need to invoke the Web service.

You have a Web service that is configured to use JMS to transport the requests.

Publish the WSDL file.

Java Message Service endpoint URL syntax: A Java Message Service (JMS) endpoint URL is used to access 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.

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.

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 appear 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 they can be set 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 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.

You need an EJB that you can directly access as a Web service.

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 J2EE 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.

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

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.

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.

Developing a Web service from a Java bean

This task explains how to develop a Web service from a Java bean.

Set up a Web services development and unmanaged client run-time environment.

This task is one of four ways that you can develop a Web service. You can also develop a Web service from an enterprise bean, develop a Web service with an existing Web Services Description Language (WSDL) file using a Java bean, or develop a Web service with an existing WSDL file using an enterprise bean. In this task, you need develop a new WSDL file.

You can use a Java bean that already exists and then enable the implementation for Web services. Enabling the Java bean for Web services includes developing the service endpoint interface, developing a WSDL file that is the engine of the Web service, generating and configuring the deployment descriptors, assembling all artifacts required for the Web service, and deploying the application into the WebSphere Application Server environment.

Develop a Web service from a Java bean by following the task steps provided in this section.

1. Access an existing Java bean Web archive (WAR) file.
2. Develop a Java bean service endpoint interface. The service endpoint interface defines the methods for a particular Web service. The Java bean must implement methods having the same signature as the methods on the service endpoint interface.
3. Develop a WSDL file. The WSDL file is the engine of a Java 2 Platform, Enterprise Edition (J2EE) Web service; without it there is no Web service.
4. Develop Web services deployment descriptor templates for a JavaBeans implementation. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans implementation.
5. Complete the JavaBeans implementation. When you complete the JavaBeans implementation, you are assembling a Java archive (JAR) file that contains a JavaBeans implementation and supported classes created from the WSDL file.
6. Configure the `webservices.xml` deployment descriptor. Configure the `webservices.xml` deployment descriptor so that WebSphere 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 WebSphere Application Server can process the incoming Web services requests.
8. Assemble a WAR file that is enabled for Web services from Java code. This article explains how to assemble the artifacts required to enable the Web module for Web services are added to the WAR file.
9. Assemble a WAR file that is enabled for Web services into an EAR file. This topic explains how to assemble the artifacts required to enable the Web module for Web services that are added to the EAR file.
10. Deploy the EAR file into WebSphere Application Server.

    This topic presents the steps necessary to deploy the EAR file that has been configured and enabled for Web services.

You have a Web service developed from a Java bean.

After you deploy the EAR file, test the Web service to make sure that it works with WebSphere Application Server.

Developing a service endpoint interface for a JavaBeans implementation:

This task explains how to develop a service endpoint interface if you are developing a Web service from a JavaBeans implementation.

You need to set up a Web services development and unmanaged client run-time environment and access an existing Java bean Web archive (WAR) file.

This task is a required step in developing a Web service from a Java bean.

The service endpoint interface defines the methods for particular 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 Java API for XML-based remote procedure call (JAX-RPC) specification, which is available through Web services: Resources for learning.

You can also create a service endpoint interface by using the 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.

   > **Windows**   Use the **javac** commands.

You have developed a service endpoint interface that you can use to develop Web services.

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.

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 WSDL file:

This topic explains how to develop a Web Services Description Language (WSDL) file.

Depending on your development path, develop a Service Endpoint Interface for a Java bean implementation or develop a Service Endpoint Interface from an EJB remote interface.

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.

   - 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.

A WSDL file that defines the Web services described by the service endpoint interface.

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`.

Depending on your development path, develop Web services deployment descriptor templates for a Java bean implementation or develop Web services deployment descriptor templates for an EJB implementation.

Developing 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 Web services that are developed using the Web services for Java 2 Platform, Enterprise Edition (J2EE).

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.

Completing this task creates the deployment descriptors used to describe how to map the service implementation to a JavaBeans component.

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 Java API for XML-based remote procedure call (JAX-RPC) mapping file are generated in the `WEB-INF` subdirectory.

Develop deployment descriptor templates by running the designated command:

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.

You have deployment descriptor templates that are required to implement or use Web services.

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
```

Now, you need to configure the `webservices.xml` deployment descriptor and configure the `ibm-webservices-bnd.xmi` deployment descriptor so that WebSphere Application Server can process the incoming Web services. After you configure the deployment descriptors, you must assemble the Web services application for deployment.

Completing the JavaBeans implementation:

This task explains how to complete the JavaBeans implementation after you have developed the deployment descriptor bindings and the bindings necessary to develop a Web service.

Develop JavaBeans implementation templates and bindings from a Web Services Description Language (WSDL) file. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans implementation. This task is a required step in developing a Web service from a Java bean.

When you complete the JavaBeans implementation, you are assembling a Java archive (JAR) file that contains a JavaBeans implementation and supported classes created from the WSDL file.

Complete the JavaBeans implementation by following the steps provided in this task section.
1. Edit the JavaBeans implementation template, *binding*Impl.java. Where *binding* is the name of the <wsdl:binding> element in the WSDL file.
   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 when developing implementation templates and bindings from a WSDL file.

You have a Java archive (JAR) file containing the JavaBeans implementation and supported classes created from the WSDL file.

You need to configure the `webservices.xml` deployment descriptor and configure the `ibm-webservices-bnd.xmi` deployment descriptor so that WebSphere Application Server can process the incoming Web services requests.

Developing a Web service from an enterprise bean

This task explains how to develop a Web service from an enterprise bean.

Set up a Web services development and unmanaged client run-time environment.

This task is one of four ways that you can develop a Web service. You can also develop a Web service from a Java bean, develop a Web service with an existing Web Services Description Language (WSDL) file using a Java bean, or develop a Web service with an existing WSDL file using an enterprise bean. In this task, you need develop a new WSDL file.

Enabling the enterprise bean for Web services includes developing the service endpoint interface, locating or developing a WSDL file that is the engine of the Web service, generating and configuring the deployment descriptors, completing the EJB implementation, assembling all the artifacts required for the Web service, enabling the modules and deploying the application into the WebSphere Application Server environment.

To use an enterprise bean as the basis for a Web service implementation, follow these requirements:
- The enterprise bean must be a stateless session bean.
- Web service method parameters must be one of the supported Java API for XML-based remote procedure call (JAX-RPC) types.

These requirements are documented in the JAX-RPC specification available through Web services: Resources for learning.

Create the artifacts that enable the enterprise bean to be a Web service and assemble the artifacts into the enterprise application:

1. Access an existing Java archive (JAR) file to use as a Web service. Make sure that the enterprise bean meets the requirements.
2. Develop an Enterprise JavaBeans (EJB) service endpoint interface. The service endpoint interface defines which enterprise bean methods should be made available as a Web service.
3. Develop a Web Services Description Language (WSDL) file. The WSDL file is the engine of a Java 2 Platform, Enterprise Edition (J2EE) Web service; without it there is no Web service.
4. Develop Web services deployment descriptor templates from an EJB implementation. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the EJB implementation.
5. Complete the EJB implementation.
6. Configure the `webservices.xml` deployment descriptor. Configure the `webservices.xml` deployment descriptor so that WebSphere 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 WebSphere Application Server can process the incoming Web services requests.
8. Assemble a JAR file that is enabled for Web services from an enterprise bean. This article explains how to assemble the artifacts required to enable the EJB module for Web services into the JAR file.
9. Assemble a Web services-enabled enterprise bean JAR file into an enterprise archive (EAR) file. This topic explains how to assemble the artifacts required for Web services into to the EAR file.
10. Enable the EAR file. When the EAR file contains EJB modules, it must have the Web services endpoint Web archive (WAR) file added with the endptEnabler tool before it is deployed.
11. Deploy the EAR file into WebSphere Application Server.

    This topic presents the steps necessary to deploy the EAR file that has been configured, assembled and enabled for Web services.

You have a Web service developed from a stateless session enterprise bean.

Publish the WSDL file.

Developing a service endpoint interface from an EJB:

This topic explains how to develop a service endpoint interface from an Enterprise JavaBeans (EJB).

Set up a Web services development and unmanaged client run-time environment.

This task is a required step in developing a 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, which is available through Web services: Resources for learning.

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 the assembly tools.

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.

   ▶ Windows ◀ Use the **javac** commands.

You have a service endpoint interface that you can use to develop a Web service.

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.

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 Web services deployment descriptor templates for an EJB implementation:

This topic explains how to develop deployment descriptor templates for an Enterprise JavaBeans (EJB) implementation that is enabled for Web services.

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.

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 Web address of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like this: *file:drive:\path\ file_name.*wsdl. If you are using the UNIX or iSeries platform, the URL looks like this: *file:/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.

Develop deployment descriptor templates with the following step provided in this task section.

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.

You have deployment descriptor templates that are required to implement a Web service.

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
   ```

Continue to complete the steps that are necessary to develop a 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 EJB implementation:

This task explains how to complete the Enterprise JavaBeans (EJB) implementation.

Develop EJB implementation templates and bindings from a WSDL file. 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.

When you complete the EJB implementation, you are assembling an enterprise bean Java archive (JAR) file that contains the EJB and supporting classes created from a WSDL file.

Complete the EJB implementation by following the steps provided in this task section.

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. Inspect the *portType*Home.java EJB home interface template. If necessary, modify the template.

3. Edit the *binding*Impl.java EJB implementation template. Where *binding* is the name of the `<wsdl:binding>` element in the WSDL file.

   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.

4. Compile all the Java classes.

5. Assemble an EJB Java archive (JAR) file. Assemble all the Java classes into an enterprise bean JAR file using the typical EJB assembly tools. Include all of the classes generated from running the **WSDL2Java** command tool when developing implementation templates and bindings from a WSDL file.

You have an enterprise bean JAR file containing an EJB and supporting classes created from a WSDL file.

Now that you have gathered the required artifacts for developing a Web service with an enterprise bean, you need to configure the `webservices.xml` deployment descriptor .

Developing a new Web service with an existing WSDL file using JavaBeans technology

This task explains how to develop a new Web service with an existing Web Services Description Language (WSDL) file using the JavaBeans technology.

Locate the Web Services Description Language (WSDL) file that defines the Web service to be implemented. You can develop a WSDL or obtain one from an existing Web service through e-mail, downloading or a Uniform Resource Locator (URL).

This task is one of four ways that you can develop a Web service. You can also develop a Web service from an enterprise bean, develop a Web service from a Java bean, or develop a Web service with an existing WSDL file using an enterprise bean.

Develop a new Web service with an existing WSDL file using JavaBeans technology with the following steps:

1. Develop JavaBeans implementation templates and bindings from a WSDL file. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans implementation.

2. Complete the JavaBeans implementation.

3. Configure the `webservices.xml` deployment descriptor. Configure the `ibm-webservices-bnd.xml` deployment descriptor so that WebSphere Application Server can process the incoming Web services requests.

4. Configure the `ibm-webservices-bnd.xmi` deployment descriptor. Configure the `ibm-webservices-bnd.xml` deployment descriptor so that WebSphere Application Server can process the incoming Web services requests.

5. Assemble a Web archive (WAR) file when starting from a WSDL file. This article explains how to assemble the artifacts required to enable the Web module for Web services are added to the WAR file.

6. Assemble a Web services-enabled WAR into an enterprise archive (EAR) file. This topic explains how to assemble the artifacts required to enable the Web module for Web services that are added to the EAR file.

7. Deploy the enterprise archive (EAR) file into WebSphere Application Server.

   This topic presents the steps necessary to deploy the EAR file that has been configured and enabled for Web services.

You have a new Web service with an existing WSDL file using JavaBeans technology

After you deploy the EAR file, test the Web service to make sure that it works with WebSphere Application Server.

Developing Web services deployment descriptor templates for a JavaBeans implementation:

To develop the JavaBeans implementation templates and bindings from a Web Services Description (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file.

If the WSDL file is a local file and you are running on the Windows platform, the URL looks like this example: file:*drive:\path\file_name*.wsdl. If you are using the Linux, UNIX or i5/OS platform, the URL looks like this example: file:*/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.

Develop JavaBeans implementation templates and bindings from a WSDL file by issuing the proper command:

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.

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.

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.

Complete the Java bean implementation.

Developing new Web services from an existing WSDL file using an EJB implementation

This task explains how to develop a new Web service from an existing Web Services Description Language (WSDL) file using a stateless session enterprise bean.

Set up a Web services development and unmanaged client run-time environment.

Locate the Web Services Description Language (WSDL) file that defines the Web service to implement. The SOAP address URI is not required because it is updated when your new implementation is deployed.

This task is one of four ways that you can develop a Web service. You can also develop a Web service from a JavaBeans implementation, develop a Web service from a stateless session enterprise bean, or develop a Web service with an existing WSDL file using a Java bean.

Create the enterprise bean and artifacts that enable the enterprise bean as Web services and assemble those artifacts into the enterprise application:

1. Develop implementation templates and bindings from a WSDL file. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the enterprise bean implementation.
2. Complete the enterprise bean implementation.
3. Configure the `webservices.xml` deployment descriptor. Configure the `ibm-webservices-bnd.xml` deployment descriptor so that WebSphere Application Server can process the incoming Web services requests.
4. Configure the `ibm-webservices-bnd.xmi` deployment descriptor. Configure the `ibm-webservices-bnd.xml` deployment descriptor so that WebSphere Application Server can process the incoming Web services requests.
5. Assemble a JAR file that is enabled for Web services from an enterprise bean. This article explains how to assemble the artifacts required to enable the Enterprise JavaBeans (EJB) module for Web services into the JAR file.
6. Assemble a Web services-enabled enterprise bean JAR file into an enterprise archive (EAR) file. This topic explains how to assemble the artifacts required for Web services into to the EAR file
7. Enable the EAR file. When the EAR file contains EJB modules, the EAR file must have the Web services endpoint Web archive (WAR) file added with the **endptEnabler** command or with an assembly tool before deployment.
8. Deploy the EAR file into WebSphere Application Server. This topic presents the steps necessary to deploy the EAR file that has been configured and enabled for Web services.

You have an EJB implementation of a Web service that is defined in the WSDL file.

After you deploy the EAR file, test the Web service to make sure that it works with WebSphere Application Server.

Developing EJB implementation templates and bindings from a WSDL file:

This task explains how to develop Enterprise JavaBeans (EJB) implementation deployment descriptor templates and binding from a Web Services Description Language (WSDL) file.

To develop EJB implementation templates and bindings from a WSDL file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like the following example: file:*drive:\path\file_name*.wsdl. If you are using the Linux, UNIX or i5/OS platform, the URL looks like the following example: file:*/path/file_name*.wsdl. You can also specify local files using the absolute or relative file system path.

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.

Develop implementation templates and bindings from a WSDL file:

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.

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.

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.
```

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.

Configuring Web services deployment descriptors

This task is an entry-point to the tasks that explain how to configure the deployment descriptors for a Web services application.

Before you can configure the deployment descriptors you need to complete all tasks required to develop a Web service and create the deployment descriptor templates.

You have developed a Web service that contains all the necessary artifacts and have created the deployment descriptors; now it is time to configure the deployment descriptors so that WebSphere Application Server can process the incoming Web services requests.

After you have finished configuring the deployment descriptors, you need to assemble the Web services application.

Viewing Web services deployment descriptors in the administrative console

This task is explains how to view the Web services client and server deployment descriptors for a deployed Web services application. You can view the bindings in the deployment descriptors.

Before you can view the deployment descriptors you need to complete all tasks required to develop a Web service, create the deployment descriptor templates that were generated by the **WSDL2Java** command-line tool, configure the deployment descriptors and bindings, and deploy the Web service application into WebSphere Application Server.

You have developed a Web service that contains all the necessary artifacts, created the deployment descriptors from the deployment descriptor templates, configured the deployment descriptors, and deployed the Web services application into WebSphere Application Server; now you can view the deployment descriptors and bindings in the administrative console.

To view the Web services client deployment descriptor extension, the Web services server deployment descriptor, and the Web services server deployment descriptor extension through the administrative console:

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules**.
   - Click **View Web services client deployment descriptor extension**.
   - Click **View Web services server deployment descriptor**.
   - Click **View Web services server deployment descriptor extension**.
3. Click **Expand All** to view the deployment descriptor contents.
4. Verify deployment descriptor and bindings configurations.

You have viewed and verified the deployment descriptors and bindings for the Web services application.

View Web services client deployment descriptor extension:

The bindings for an EJB or Web module are displayed when you use this page to view the Web services client deployment descriptor extension.

To view this administrative console page, click **Applications** >**Enterprise Applications** > *application_instance* > **Manage Modules** > *EJB or Web module*.

Deployment descriptors contain the information that is needed by a Web services client to communicate with the server for which the Web services is installed. This information is added to the deployment descriptor templates after a Web service is developed or an existing Web service is located. The data that you can view in the deployment descriptor includes the following:

- The Web service description including the name, WSDL file, WSDL file location and the mapping file.
- The port description, including the port component name, the WSDL port, the service endpoint interface that indicate the service's bindings, and the Enterprise Java Bean (EJB) that is used to implement the Web service.

View Web services server deployment descriptor:

The bindings for an EJB or Web module are displayed when you use this page to view the Web services server deployment descriptor extension.

To view this administrative console page, click **Applications** >**Enterprise Applications** > *application_instance* > **Manage Modules** > *EJB or Web module*..

Deployment descriptors contain the information that is needed by a Web services client to communicate with the server for which the Web services is installed. This information is added to the deployment descriptor templates after a Web service is developed or an existing Web service is located. The data that you can view in the deployment descriptor includes the following:

- The Web service description including the name, WSDL file, WSDL file location and the mapping file.

- The port description, including the port component name, the WSDL port, the service endpoint interface that indicate the service's bindings, and the Enterprise Java Bean (EJB) that is used to implement the Web service.

View Web services server deployment descriptor extension:

The bindings for an EJB or Web module are displayed when you use this page to view the Web services server deployment descriptor extension.

To view this administrative console page, click **Applications** >**Enterprise Applications** > *application_instance* > **Manage Modules** > *EJB or Web module.*.

Deployment descriptors contain the information that is needed by a Web services client to communicate with the server for which the Web services is installed. This information is added to the deployment descriptor templates after a Web service is developed or an existing Web service is located. The data that you can view in the deployment descriptor includes the following:
- The Web service description including the name, WSDL file, WSDL file location and the mapping file.
- The port description, including the port component name, the WSDL port, the service endpoint interface that indicate the service's bindings, and the Enterprise Java Bean (EJB) that is used to implement the Web service.

Configuring the webservices.xml deployment descriptor

This topic explains how to configure the `webservices.xml` deployment descriptor with an assembly tool.

To configure the client deployment descriptor see ″Client Deployment Descriptor editor″ in the Application Server Toolkit documentation for more information.

Before you can configure the `ibm-webservices-bnd.xml` deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

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 a Java bean or an enterprise bean:
- Develop Web services Java bean 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. The Eclipse assembly tools, Application Server Toolkit (AST) and Rational Web Developer, provide a graphical interface for developing code artifacts, assembling the code artifacts into various archives (modules) and configuring related Java 2 Platform, Enterprise Edition (J2EE) Version 1.2, 1.3 or 1.4 compliant deployment descriptors.
2. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
3. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.

4. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.

5. Configure the deployment descriptor. See ″Editing Web services″ in the Application Server Toolkit documentation for more information.

You have a `webservices.xml` deployment descriptor that is configured.

Now, you must configure the `ibm-webservices-bnd.xmi` deployment descriptor.

Configuring the ibm-webservices-bnd.xmi deployment descriptor

This topic explains how to configure the `ibm-webservices-bnd.xml` deployment descriptor.

To configure the client deployment descriptor see ″Client Deployment Descriptor editor″ in the Application Server Toolkit documentation for more information.

Before you can configure the `ibm-webservices-bnd.xml` deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

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 a Java bean or an enterprise bean:
- Develop Web services Java bean 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. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
2. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
3. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.
4. Configure the client deployment descriptor. See ″Editing Web services″ in the Application Server Toolkit documentation for more information.

The `ibm-webservices-bnd.xmi` deployment descriptor is configured for the Web service implementation module.

If you are developing a Web service from a Java bean, 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.

ibm-webservices-bnd.xmi assembly properties:

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.

Note:

The following user-defined assembly properties are supported:
* **wsDescNameLink**

   Attribute of the wsdescBindings element that specifies the link to the corresponding <webservice-description-name> elemetn 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.

```
<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>
```

Configuring the webservices.xml deployment descriptor for handler classes

This topic explains how to use and assembly tool to configure the `webservices.xml` deployment descriptor for user-provided handler classes.

You can configure deployment descriptors with assembly tools provided with WebSphere 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.

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 J2EE specification and Chapter 12 of the JAX-RPC specification available through Web services: Resources for learning. The application modules must contain the `webservices.xml` deployment descriptor.

Configure a handler in the `webservices.xml` deployment descriptor by following the listed steps:

1. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
2. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
3. Migrate the Web archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.
4. Configure the client deployment descriptor. See ″Editing Web services″ in the Application Server Toolkit documentation for more information.

Configuring the Web services client deployment descriptor with an assembly tool

This topic explains how to configure the client deployment descriptor with an assembly tool.

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.

Complete this task if you are developing a managed client that runs in the J2EE client container. This task is done after you assemble the EJB or Web module.

Configure the client deployment descriptor with an assembly tool by following the steps provided:

1. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation.for more information.
2. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
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 an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.
4. Configure the client deployment descriptor. See ″Client Deployment Descriptor editor″ in the Application Server Toolkit documentation for more information.

You have a client deployment descriptor that is configured. Now you can test the Web services client to make sure it works in the WebSphere Application Server run time environment.

Test the Web services client. This task explains how to test an unmanaged client JAR file and an unmanaged client application.

Configuring the client deployment descriptor for handler classes with an assembly tool

This topic explains how to use an assembly tool to configure the client deployment descriptor for user-provided handler classes.

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 2 Platform, Enterprise Edition (J2EE) specification and chapter 12 of the Java API for XML-based remote procedure call (JAX-RPC) specification available through Web services: Resources for learning. The application modules must contain the `webservices.xml`(for server) and the client deployment descriptors.

Configure a handler in the client deployment descriptor by following the steps provided:

1. Start an assembly tool. See ″Starting WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
2. If you have not done so already, configure the assembly tool to work on J2EE modules. Make sure that the **J2EE** and **Web** categories are enabled. See ″Configuring WebSphere Application Server Toolkit″ in the Application Server Toolkit documentation for more information.
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 an AST or Rational Web Developer assembly tool. To migrate files, import your WAR files to the assembly tool. See ″Importing Web archive (WAR) files″ in the Application Server Toolkit documentation for more information.
4. Configure the client deployment descriptor. See ″Creating Web services handlers″ in the Application Server Toolkit documentation for more information.

You have a client deployment descriptor that is configured.

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: 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 2 Platform, Enterprise Edition (J2EE) technology descriptor field.

Display name

Standard J2EE technology descriptor field.

Small icon

Standard J2EE technology descriptor field.

Large icon

Standard J2EE 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: Resources for learning, 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: Resources for learning, 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

   %JAVA_HOME%\bin\java -extdirs %WAS_EXT_DIRS% ClientHandler.java ServerHandler.java (on Windows systems)

   $JAVA_HOME/bin/java -extdirs $WAS_EXT_DIRS ClientHandler.java ServerHandler.java (on Linux and Unix systems)

2. Open an assembly tool and import the two sample enterprise archive (EAR) files:

   - %WAS_HOME%\samples\lib\WebServicesSamples\WebServicesSamples.ear on Windows systems or $WAS_HOME/samples/lib/WebServicesSamples/WebServicesSamples.ear on Linux and Unix systems.
   - %WAS_HOME%\samples\lib\WebServicesSamples\ApplicationClients.ear on Windows systems or $WAS_HOME/samples/lib/WebServicesSamples/ApplicationClients.ear on Linux and Unix systems.

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.

Developing Web services clients

This topic explains how to develop a Web services client based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

You need a Web Services Description Language (WSDL) file to use Web services. Before you begin this task, locate the WSDL file that defines 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.

For a Java application to act as a Web service client, a mapping between the Web Services Description Language (WSDL) file and the Java application must exist. The mapping is defined by the Java API for XML-based RPC (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 J2EE specification. The JAX-RPC specification defines the mapping between a WSDL file, Java code and XML Schema types.

Create the client code and artifacts that enable the application client to access a Web service by following the steps provided:

1. Develop client bindings from a WSDL file. The client-side bindings and deployment descriptors are generated.
2. Complete the client implementation.
3. (Optional) Assemble a Web services-enabled client Java archive (JAR) file. Complete this step if you are developing a managed client that runs in the J2EE client container.
4. (Optional) Assemble a Web services-enabled client Web archive (WAR) file. Complete this step if you are developing a managed client that runs in the J2EE client container.
5. (Optional) Configure the client deployment descriptor. Complete this step if you are developing a managed client that runs in the J2EE client container.
6. Test the Web services-enabled client application. This task explains how to test an unmanaged client JAR file and an unmanaged client application.

You have created and tested a Web services client application. For step-by-step information see Example: Developing Web services clients.

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. You can change the endpoint with the administrative console or the wsadmin scripting tool.

Developing client bindings from a WSDL file

This topic explains how to develop client bindings from a Web Services Description (WSDL) file.

To develop the client bindings from a WSDL file, 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 it is a local file and you are running the Windows platform, the URL looks like the following example: file:*drive:\path\file_name*.wsdl. If you are using the Linux or UNIX platform, the URL looks like the following example: file:*/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.

Develop client bindings from a WSDL file by running the appropriate command:

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.

You have the bindings and deployment descriptors needed by a client to use a Web service.

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.
```

Complete the client implementation.

Assemble a Web services-enabled client JAR and EAR file.

Setting up a development and unmanaged client run-time environment for Web services

WebSphere Application Server provides command-line tools to develop Web services clients and implementations that are based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification. This topic explains how to set up your development and unmanaged client run environment in order to start the development process with these tools.

Before you can set up a Web services development and unmanaged client execution environment within WebSphere Application Server, you must Install WebSphere Application Server.

Set up a Web services development and thin client environment by following the listed actions:
1. Set up the environment.

   You can set **WAS_USER_SCRIPT** to *install_root*\profiles\<*application_server*>\bin\ `setupCmdLine.bat`, which has the same effect as running the **setupCmdLine** command.

Run the **setupCmdLine.bat** command if you are using a Windows platform.

Linux and UNIX-based platforms *source* the script to the parent shell to inherit the exported variables by running this command:

. ./setupCmdLine.sh   (Notice the space between the periods.)

from the/*install_root*/profiles/*<application_server>*/bin directory or *. /install_root*/profiles/ *<application_server>*/bin/setupCmdLine.sh, if the command is executed from anywhere else.

2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:

   ▶ Windows   On Windows platforms:

   `set PATH=%WAS_PATH%;%PATH%`

3. Configure CLASSPATH. You can add `com.ibm.ws.webservices.thinclient__6.1.0.jar`, `com.ibm.ws.wccm__6.1.0.jar` and *<your_application_jars>* to CLASSPATH by typing:

   On Windows platforms:

   ```
   set CLASSPATH=.;%WAS_HOME%\runtimes\com.ibm.ws.webservices.thinclient_6.1.0.jar;
     %WAS_HOME%\plugins\com.ibm.ws.wccm__6.1.0.jar;%;<your_application_jars>;
     %WAS_CLASSPATH%;%CLASSPATH%
   ```

   On Unix platforms:

   ```
   export CLASSPATH=.:$WAS_HOME/runtimes/com.ibm.ws.webservices.thinclient_6.1.0.jar:
     $WAS_HOME/plugins/com.ibm.ws.wccm__6.1.0.jar:<your_application_jars>:
     $WAS_CLASSPATH:$CLASSPATH
   ```

4. Run the client application using the Java command to call the main class directory as follows: On Windows platforms:

   `%JAVA_HOME%\bin\java <your_client_application>`

   On Unix platforms:

   `$JAVA_HOME/bin/java <your_client_application>`

You have set up a development and unmanaged client run-time environment so that you can develop Web services. If you you get a `NullPointerException` error when the HTTP basic authentication fails, you can fix the problem by including `<JAVA_HOME>\jre\lib\ext` in the classpath at the command-line, or by manually editing it at WAS_EXT_DIRS.

Develop Web services. This topic is a good starting point in learning about how to develop a J2EE Web service.

Example: Developing Web services clients

This example takes you through the steps to develop a Web services client. The development process is based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) and the Java API for XML-based remote procedure call (JAX-RPC) specification.

You need a Web Services Description Language (WSDL) file to use Web services. Before you begin this task, locate the WSDL file that defines the Web service that you want to access. You can locate the WSDL from the services provider through e-mail, downloading, or through a Uniform Resource Locator (URL).

For a Java application to act as a Web service client, a mapping between the Web Services Description Language (WSDL) file and the Java application must exist. The mapping is defined by the Java API for XML-based RPC (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 J2EE specification. The JAX-RPC specification defines the mapping between a WSDL file, Java code and XML Schema types.

Create the client code and artifacts that enable the application client to access a Web service by following the steps provided.

Steps for this task

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 your WSDL file.

   The **WSDL2Java** command-line tool is run against your WSDL file to develop client bindings.

   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. Implement the client.

   See Chapter 4 of the JSR-109 specification. You can access the specification through Web services: Resources for learning.

   Note: If an application creates a number of threads in the JSR-109 client, the meta data (including the WebSphere Application Server configuration) is not copied to the thread, and the Global Security Handler is not called.

   You can also review the GetQuote client in the WebServicesSamples application available in the Samples Gallery.

4. Assemble the module.

   Assemble the client JAR file into an EAR file or assemble the client WAR file into an EAR file.

5. Configure the deployment descriptors.

   Configure the client deployment descriptor.

6. Test the Web services client.

   You should test the client to make sure it correctly operates and binds to the Web service.

Configuring Web service client bindings

When a Web service application is deployed into WebSphere Application Server, an instance is created for each application or module. The instance contains deployment information for the Web module or enterprise JavaBean (EJB) module, including client bindings.

Deploy the Web service into WebSphere Application Server.

To complete this task, you need to know the topology of the URL endpoint address of the Web services servers and which Web service the client depends upon. You can view the deployment descriptors in the administrative console to find topology information. See the article View Web services server deployment descriptors for more information.

The client bindings define the Web Services Description Language (WSDL) file name and preferred ports. The relative path of a Web service in a module is specified within a compatible WSDL file that contains the actual URL to be used for requests. The address is only needed if the original WSDL file did not contain a URL, or when a different address is needed. For a service endpoint with multiple ports, you need to define an alternative WSDL file name.

The following steps describe how to edit bindings for a Web service after these bindings are deployed on a server. When one Web service communicates with another Web service, you must configure the client bindings to access the downstream Web service.

You can also configure client bindings with **wsadmin**.

To configure client bindings through the administrative console:

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* > **Web services client bindings**.
3. Find the Web service you want to update.

   The Web services are listed in the **Web Service** field.
4. Select the WSDL file name from the drop down box in the WSDL file name field.
5. Click **Edit** in the Preferred port mappings field to configure the default port to use.

   a. Specify the port type and the preferred ports in the Port type and Preferred ports fields.

      Configuring the preferred port enables you to select an optimal port implementation use non-SOAP protocols. See RMI-IIOP Web services using JAX-RPC for more information about using non-SOAP protocols.

   b. Click **Apply** and **OK**.
6. Click **Edit** in the Port information field to configure the request timeout, the overridden endpoint, and the overridden binding namespace for a port.

   Configuring the request timeout accommodates complex topologies that can have multiple cascaded Web services that involve multiple hops or long-running services.

   Timeout values can be configured based on observed behavior of the overall system as integration proceeds. For example, a Web service client might time out because of changing network conditions or the performance of an external Web service. When you have applications containing Web services clients that timeout, you can change the request time out values for the clients.

   a. Click **Apply** and **OK**.

Your Web service client bindings are configured.

Now you can finish any other configurations, start or restart the application, and verify the expected behavior of the Web service.

Web services client bindings

The client bindings define the Web Service Description Language (WSDL) file name, preferred ports and other port information. Use this page to specify the client bindings and the port mappings for the Web services in a module.

A Web service can specify the relative path within the module of a compatible WSDL file containing the actual URL to be used for requests. The relative path only needs to be specified if the original WSDL file does not contain a URL or when a different URL is needed. For a service endpoint with multiple ports defined, a preferred mapping specifies the default port to use for a port type.

Web service:

Identifies the name of this Web service. A module can contain one or more Web services.

EJB:

Identifies the name of the EJB for the EJB modules.

WSDL file name:

Specifies the WSDL file name, which is relative to the module. Locate the WSDL file name in the drop down menu.

Preferred port mappings:

Specifies and manages the preferred port type mapping for a Web service when a particular port type is requested.

Click **Edit** to edit the preferred port mapping information on the **Preferred port mappings** panel.

Port information:

Specifies additional configuration information for the ports of this Web service.

Click **Edit** to edit the port information on the **Port information** panel. You can set a request timeout, override an endpoint and override a binding namespace for each client port.

Preferred port mappings:

Use this page to view and manage a preferred portType mapping for a Web service.

When you have multiple ports that reference the same portType (service endpoint interface), a preferred port specifies the port to use when the `Service.getPort(Class SEI)` method is called with only the service endpoint interface.

To view this administrative console page, click **Applications** >**Enterprise Application** > *application_instance* > **Manage Modules** > *module_instance* >**Web services client bindings** > **Edit** > *preferred_port_instance*.

portType:

Specifies the portType.

The preferred port and the portType values are both of the type `java.xml.namespace.QName`.

Preferred port:

Specifies the preferred port to be associated with a particular portType. The `Service.getPort(Class)` method returns the preferred port associated with the specified service endpoint interface class (portType).

The preferred ports available are listed, as well as a value of `None`, which indicates no preferred port is selected.

Web services client port information:

Use this page to specify a request timeout, override an endpoint, and override a binding namespace for a Web services client port.

A Web service can have multiple ports. You can view and configure the port attributes for each defined Web service port. The Web services are listed on the Web services client bindings panel.

To view this page, click **Applications** >**Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* >**Web services client bindings** > **Edit**.

For EJB modules, click **Applications** >**Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* >**Web services client bindings** > **Edit**.

Port:

Specifies the name of a port.

Request timeout:

Specifies the time, in seconds, that a Web service client waits for a request to complete on this port. If a timeout is not specified, the default request timeout for the client to wait is `360` seconds. If the value is set at `0` (zero), the client's request does not timeout.

A typical use for this setting is to customize the client's behavior when it is configured to use a JMS transport to access a Web service to make it wait longer for an expected completion. Depending upon network conditions, or the nature of a Web service implementation, it might be necessary to tune the timeout.

Overridden endpoint:

Specifies the name of an endpoint that is used to override the current endpoint. A client invoking a request on this port uses this endpoint instead of the endpoint specified in the WSDL file.

If an assembled application contains a Web service client that is statically bound, the client is locked into using the implementation (service end point) identified in the WSDL file used during development. Overriding the endpoint is an alternative to configuring the deployed WSDL attribute.

The overridden endpoint URI attribute is specified on a per port basis. It does not require an alternative WSDL file within the module. The overridden endpoint URI takes precedence over the deployed WSDL attribute. The client uses this value for the service end point URI or SOAP address, instead of the value in the static client bindings.

Overridden binding:

Specifies the WSDL file binding namespace URI to use with this port, instead of the namespace in the WSDL file. This binding does not need to exist in the WSDL file. A client invoking a request on this port uses this binding instead of the binding specified in the WSDL file. An overridden binding namespace cannot be specified unless an overridden endpoint is specified.

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 required to create a Web service that is accessed using a WS-Addressing endpoint reference.

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.

1. To perform the basic WS-Addressing development activities 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 API: Creating an application that uses endpoint references" on page 478.

2. To perform more advanced WS-Addressing functions, such as setting or retrieving message addressing properties, refer to"Using the WS-Addressing SPI: Performing more advanced Web Service Addressing tasks" on page 485.

3. To configure a service or a client to use the WS-Addressing support, refer to "Enabling Web Services Addressing support" on page 493.

Web Services Addressing support

The Web Services Addressing (WS-Addressing) support in WebSphere Application Server provides the environment for Web services that use the W3C WS-Addressing specifications. This family of specifications provide transport-neutral mechanisms to address Web services and to facilitate end-to-end addressing.

You do not normally need to be aware of the underlying WS-Addressing support as WebSphere Application Server will ensure that your Web service applications are WS-Addressing compliant when required. Read this topic, and other WS-Adressing and Web Services Resource Framework (WSRF) topics, 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 APIs to create endpoint references within an application, and then use these references to target Web service resource instances. For example, a WSRF application developer.
- A system programmer who needs to use the WS-Addressing SPIs to perform more advanced WS-Addressing operations, such as specifying message addressing properties on Web services messages.

Features of the WS-Addressing support

The WS-Addressing support in WebSphere Application Server provides the following features:

For core WS-Addressing application development using the API

- Java representations of WS-Addressing endpoint references.
  - You can easily create Java endpoint reference instances at run time based on the application's deployment environment. You do not have to specify the URI of the endpoint reference. Additionally, endpoint references can represent highly available or workload managed objects.
  - Runtime and tooling to provide appropriate mapping between the Java and XML representations of an endpoint reference. This ensures that endpoint references appearing on application Web service interfaces are appropriately serialized and deserialized to or from SOAP automatically.
  - You can configure client JAX-RPC Stub or Call objects with a WS-Addressing endpoint reference. Future invocations on the client Stub or Call object are targeted at the endpoint represented by the endpoint reference. The invocations also automatically conform to the WS-Addressing specification (namespace) associated with that endpoint reference.
- 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.

For extended WS-Addressing system development using the SPI

- 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 this 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, WebSphere Application Server supports the W3C WS-Addressing 1.0 Core and SOAP Binding specifications identified by the http://www.w3.org/2005/08/addressing namespace. Unless otherwise stated, WS-Addressing semantics described in this documentation refer to these specifications.

For interoperability, other levels of the WS-Addressing specification are supported in this version of WebSphere Application Server; in particular, the WS-Addressing W3C submission with the namespace http://schemas.xmlsoap.org/ws/2004/08/addressing.

In addition, WebSphere Application Server supports the following features from the WS-Addressing WSDL binding specification:

- The wsaw:UsingAddressing extensibility element, on the WSDL Binding element only. The supported namespace for this element is the http://www.w3.org/2006/02/addressing/wsdl namespace.
- The wsaw:Action extensibility element. The supported namespaces for this element are the http://schemas.xmlsoap.org/ws/2004/08/addressing namespace and the http://www.w3.org/2006/02/addressing/wsdl namespace.

Web Services Addressing overview:

Web Services Addressing (WS-Addressing) is a 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 8. Information model for endpoint references

Abstract Property name	Property type	Multiplicity	Description
[address]	xs:anyURI	1..1	The absolute URI specifying 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 above table.

Table 9. 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/2006/02/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 (MAPs)

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 replies to the message should be directed, or information about the relationship that the message has with other messages. The MAPs defined by the WS-Addressing specification are summarized in the following table.

Table 10. 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 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 will default to the anonymous URI 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 above tables are used to refer to the MAPs throughout this documentation.

The following example 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 W3C Web Services Addressing (WS-Addressing) specification explicitly defines the WS-Addressing core properties for the message exchange patterns (MEPs) 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 is a straightforward one-way message defined in WSDL 1.0 as an input-only operation. The WSDL for this operation is of the 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 WebSphere Application Server runtime, to ensure compliance with the WS-Addressing specification.

Tip: You can override these values using the WS-Addressing SPIs.

Table 11.

Abstract WS-Addressing MAP name	Default value for one-way input message
[action]	The WS-Addressing [action] 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 runtime automatically sets this.

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 related messages should be sent. In order to propagate a [reply endpoint] or [fault endpoint], associate the appropriate property with the JAX-RPC Stub or Call object as described in "Web Services Addressing SPI" on page 487, to override the defaults.

Two-way request-response

This is request-response 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 WSDL definition for a request-response operation is, therefore, of one of the following forms:

```
<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 WebSphere Application Server client runtime ensures that the SOAP header of the outgoing request message contains the relevant WS-Addressing message information headers, in other words, the calling application does not have to set the WS-Addressing headers. In this case, a response is expected therefore a [reply endpoint] or [fault endpoint] must be specified in the request message.

Note: In the 2005/08 specification, an unspecified [reply endpoint] is valid as it defaults to an endpoint reference containing the anonymous URI.

The following table summarizes the MAPs that WebSphere Application Server sets by default on a Web service request that uses WS-Addressing. You can override or specify other MAPs using the WS-Addressing SPIs.

Table 12.

Abstract WS-Addressing MAP name	Default Value for input message of request-response operation
[action]	The WS-Addressing [action] 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 WebSphere Application Server on a WS-Addressing response or fault message.

Table 13.

Abstract WS-Addressing MAP name	Default Value for input message of request-response operation
[action]	The WS-Addressing [action] 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] passed in the request message.
[message id]	A uniquely generated message identifier; although not mandated by the spec, the WebSphere Application Server runtime automatically sets this property.

Synchronous request-response

By default, in other words if you do not use the 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.
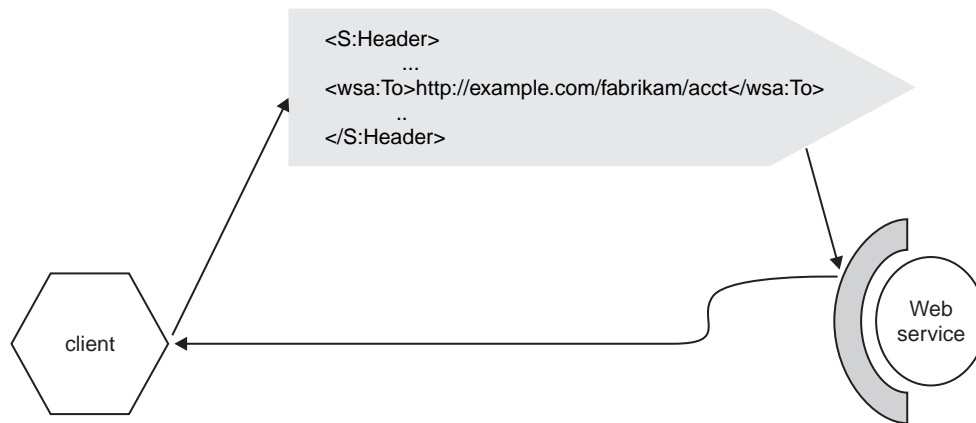
Asynchronous request-response

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 directed at endpoints hosted on a 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.
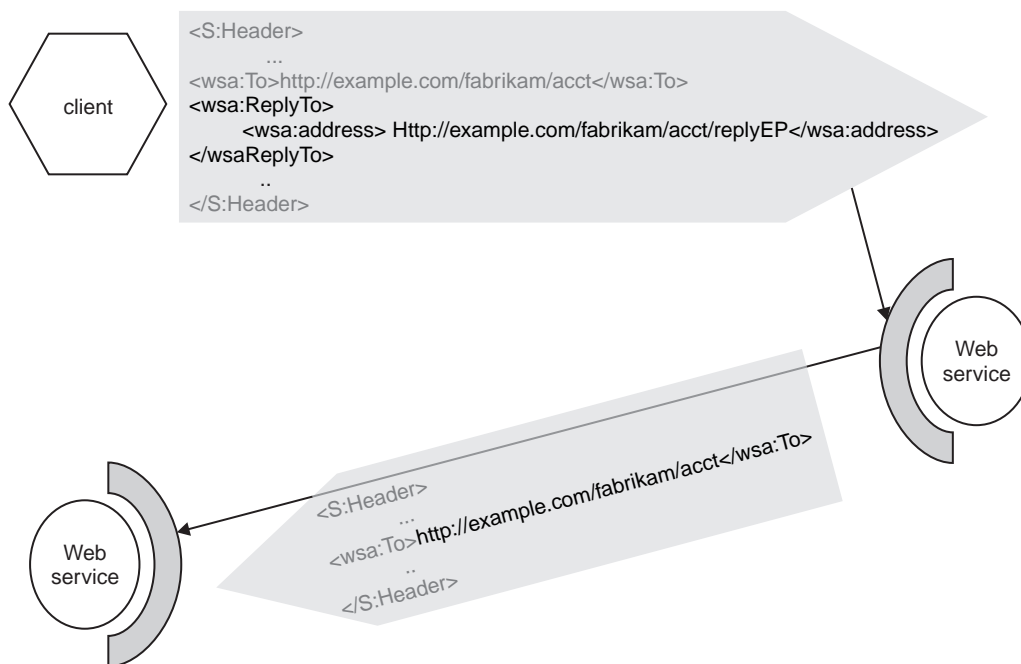
Web Services Addressing version interoperability:

The Web Services Addressing (WS-Addressing) support in WebSphere Application Server can interoperate with various versions of the WS-Addressing specification.

Table 14. 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.
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. This is the default namespace used by WebSphere Application Server for the WSDL parts of the WS-Addressing specification.
http://schemas.xmlsoap.org/ws/2004/08/addressing	http://www.w3.org/Submission/ws-addressing/	W3C WS-A Submission 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 WebSphere Application Server interoperates with each of the WS-Addressing specifications defined in the table above. This interoperability results in the following behavior:

- Incoming Web service messages containing WS-Addressing message addressing properties (MAPs) are appropriately bound to SOAP, and WS-Addressing SOAP elements appropriately de-serialized to their WS-Addressing programming model representations according to the namespace in use.
- WS-Addressing programming model artefacts are appropriately serialized into SOAP elements, and the MAPs 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 found. The WebSphere Application Server 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 issued from this version of WebSphere Application Server adopt the namespace associated with the destination endpoint reference. If this namespace is unknown, the message adopts the default WS-Addressing namespace.

WebSphere Application Server provides an SPI to change the namespace associated with an endpoint reference to any namespace in the supported set.

Which WS-Addressing specification should I use?

best-practices: In most cases, use the default WS-Addressing specification supported by WebSphere Application Server. 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 WebSphere Application Server.
- 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.

Web Services Addressing application programming model:

The Web Services Addressing (WS-Addressing) specification defines an endpoint reference that is represented in XML by an EndpointReferenceType which 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 and nothing more, 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 figure below illustrates a Web service, at http://www.example.com/service, and three resources, A, B and C, that are accessed through the Web service. There are thus three WS-Resources illustrated in the figure:

A WS-Resource is referenced by a WS-Addressing endpoint reference which 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 example above WS-Resource-C is the combination of the Web service and the resource identified by "C", and a reference to WS-Resource-C might appear as follows:

```
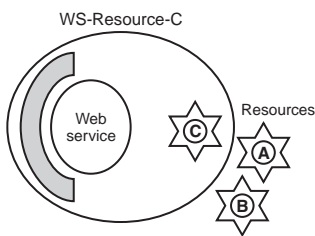<wsa:EndpointReference>
    <wsa:Address>
        http://www.example.com/service
    </wsa:Address>
    <wsa:ReferenceParameters>
        <tns:SomeDisambiguatorElement>C</tns:SomeDisambiguatorElement>
    </wsa:ReferenceParameters>
    ...
</wsa:EndpointReference>
```

The WS-Addressing API provides the appropriate interfaces for implementing the above pattern.

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.

Signing of WS-Addressing message addressing properties

You can use an assembly tool to specify the message addressing properties that require signing, or whose signature should be verified 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, lack of a verifiable signature that is associated with the specified inbound message addressing properties causes the message to be rejected 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.

Web Services Addressing, firewalls and intermediary nodes:

Using the Web Services Addressing (WS-Addressing) support in WebSphere Application Server, you can create endpoint references which can be distributed across firewalls and intermediary nodes. The topology of your system can have an affect the type of endpoint reference that is generated. For example, if the endpoint reference is created in a cluster environment, the generated endpoint reference might represent an endpoint that is workload managed.

There are situations where, as an application developer, you require specific behavior of an endpoint reference regardless of the topology into which the application is deployed. In particular, the WS-Addressing programming model allows you to indicate that an endpoint reference represents a stateful session bean that should not be workload managed because it maintains in-memory state. Additionally, if high availability for stateful session beans is enabled and the endpoint reference is created using the appropriate programming model, the endpoint reference remains valid even if the stateful session bean is failed over.

The topologies that are available to you are as follows:

Direct connection

> No intermediary node exists in this topology. The client communicates directly with the WebSphere Application Server on which the target endpoint resides.

> Your application can create, on the WebSphere Application Server, endpoint references that represent endpoints located on that server. You can then export these endpoint references to the Web service client, for example as the return value of a Web service method. The WS-Addressing API automatically generates endpoint references that contain the appropriate address information.

> If the endpoint reference represents a highly available stateful session bean, the endpoint reference remains valid following stateful session bean failover only if the client targeting the WebSphere Application Server is a WebSphere Application Server client, at Version 6.1 or later.

Proxy Server for IBM WebSphere Application Server

In this topology, the client communicates with a Proxy Server for IBM WebSphere Application Server, which dynamically routes the client requests to the appropriate server within a WebSphere Application Server cluster.

Your application can create, on the WebSphere Application Server, endpoint references that represent endpoints located on that server. You can then export these endpoint references to the Web service client, for example as the return value of a Web service method. In this topology, the address of the endpoint reference is the address of the proxy server.

If the endpoint reference represents a highly available stateful session bean, the endpoint reference remains valid after stateful session bean failover. This high availability behavior occurs only if the proxy targeting the WebSphere Application Server is a Proxy Server for IBM WebSphere Application Server, at Version 6.1 or later.
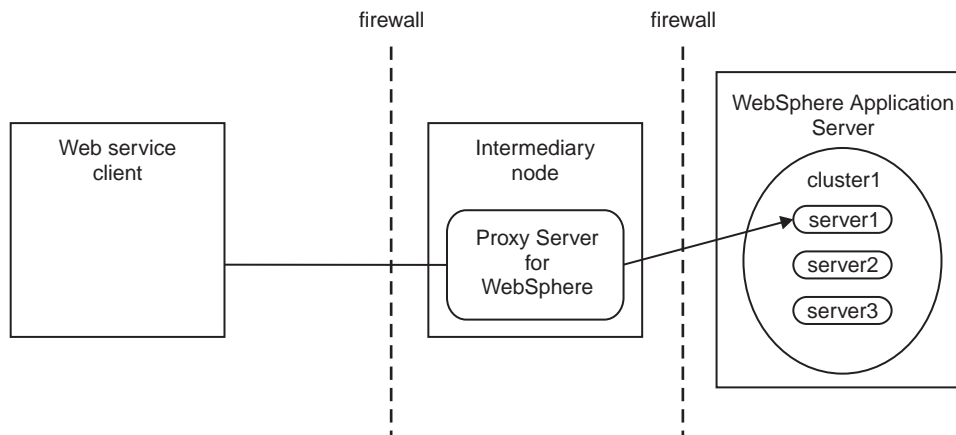
HTTP server, such as IBM HTTP Server

In this topology, the client communicates with an HTTP server which always routes the client requests to a specific server. You can configure the HTTP server to specify the WebSphere Application Server to route requests to.

Your application can create and export endpoint references as described in the previous topology. In this topology, the address of the endpoint reference is the appropriate virtual host of the HTTP server's configuration. For the WS-Addressing API to automatically generate endpoint references containing this address, configure the address for the endpoint in your Web service deployment information.

If the endpoint reference represents a highly available stateful session bean, the endpoint reference ceases to be valid after stateful session bean failover.
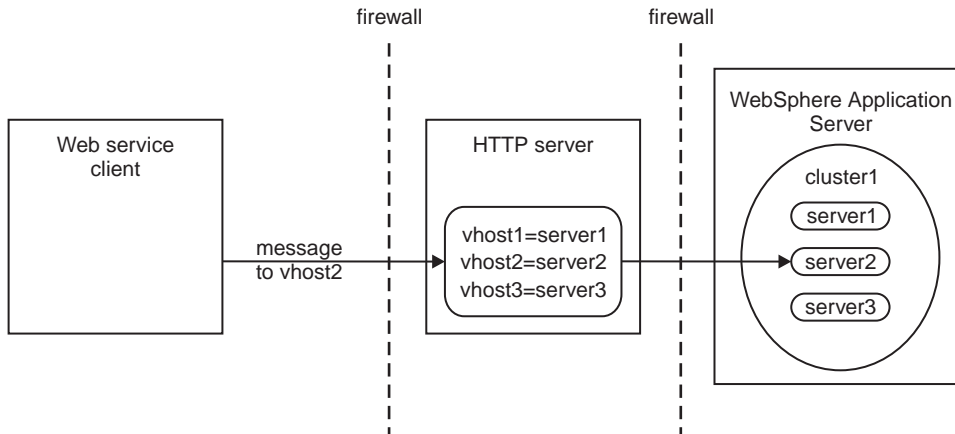
If you want the endpoint reference to remain valid after stateful session bean failover, use a Proxy Server for IBM WebSphere Application Server in addition to your HTTP server, as described in the following section.

HTTP server in conjunction with a Proxy Server for IBM WebSphere Application Server

In this topology, the client communicates with an HTTP server which you configure to forward the client's requests to a Proxy Server for IBM WebSphere Application Server. The proxy then dynamically routes the requests to the appropriate server.

This topology is similar to the previous topology, but if the endpoint reference represents a highly available stateful session bean, the endpoint reference remains valid after stateful session bean failover. This high availability behavior occurs only if the proxy targeting the WebSphere Application Server is a Proxy Server for IBM WebSphere Application Server, at Version 6.1 or later.
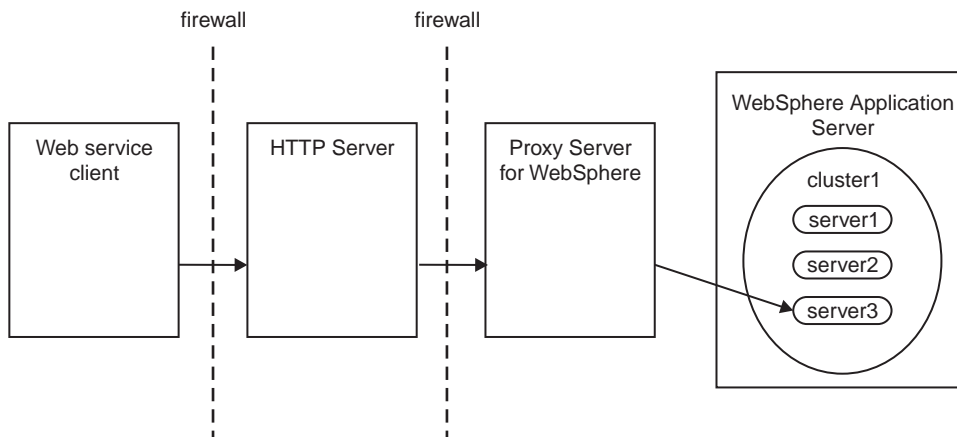
Web Services Addressing and the service integration bus:

If you are using the Web Services Addressing (WS-Addresing) 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 end point to which replies will be sent, to the service integration bus. As this is a one way request, 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 specified in the request.
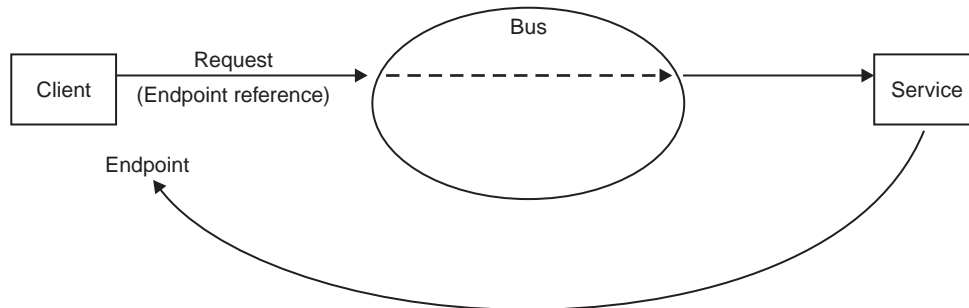
This scenario works if messages are able to flow directly from the Web service to the endpoint. If you have a configuration that does not allow direct message flow, for example if you have a firewall, you must create handlers that can redirect the message as required.

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 end point to which replies will be 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 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.
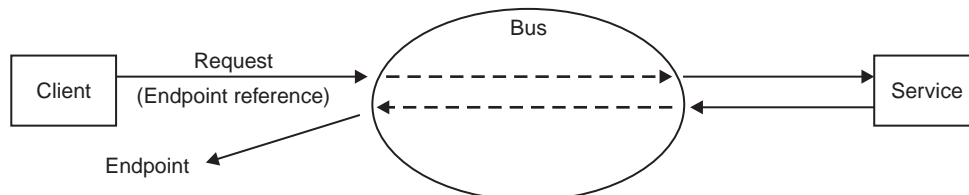
Using the Web Services Addressing API: Creating an application that uses endpoint references

WebSphere Application Server provides application programming interfaces for applications that need to create endpoint references and use those endpoint references to target Web service endpoints.

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

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, using the instructions in "Creating a Web service application that is referenced through a Web Services Addressing endpoint reference."

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 495.

Creating a Web service application that is referenced through a Web Services Addressing endpoint reference:

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 required to create a 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.

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

Perform this task if you are creating a Web service that complies with the WS-Addressing interoperability protocol, in other words one that is addressed through an endpoint reference.

1. Provide a Web service interface, by creating or generating a 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 Web Services Addressing support" on page 480.

    b. **Optional:** If your interface involves a Web service that fronts a resource instance, create or lookup 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 itself.

    Note: Do not put sensitive information in the resource identifier, as 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 will 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 produced by the service in the first step to send messages to the endpoint.

a. Obtain a Stub object in the normal J2EE fashion by looking up the Printer service in JNDI, or alternatively create an empty Call object.

b. Use the Stub or Call object's setProperty(String property_name, Object value) method to associate the endpoint reference with the Stub or Call object, using the WS-Addressing constant WSADDRESSING_DESTINATION_EPR as the property name, and the endpoint reference itself 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 interface and endpoint metadata (portType and port elements) associated with the endpoint reference.

Note: For Stub objects, if the endpoint defined by the endpoint reference conflicts with the Stub object's configuration, for example if it rendpoint referenceesents a different interface, a JAXRPCException 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, defined by the endpoint reference. When an invocation occurs, WebSphere Application Server adds appropriate message addressing properties, such as a reference parameter contained within the endpoint reference that identifies a target resource, to the message header.

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 Web Services Addressing API to access a generic Web service resource instance" on page 481.

1. Refer to "Web Services Addressing security considerations" on page 475 for information about security with WS-Addressing.

2. 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-Adressing property on the client. For more information, and for other scenarios which might require additional steps, see "Enabling Web Services Addressing support" on page 493.

Creating endpoint references using the Web Services Addressing support:

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. WebSphere Application Server provides interfaces for you to create endpoint references, and specify the behavior of those endpoint references within a cluster environment.

This task is part of "Creating a Web service application that is referenced through a Web Services Addressing endpoint reference" on page 479.

Perform this task if you are writing an application that uses the WS-Addressing support. 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 this can change when the application is deployed. Using the WS-Addressing support, you can either specify the endpoint address, or allow WebSphere Application Server to generate it for you at run time.

- To create an endpoint reference with an address that you specify directly, use the WS-Addressing EndpointReferenceManager.createEndpointReference(URI address) SPI method. This method is useful in test scenarios, where the address of the service will not change.

- To create an endpoint reference with an address that is automatically generated by WebSphere Application Server, 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. 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) API method. This method ensures that requests that are targeted at the endpoint are not workload managed. Additionally, if high availability for stateful session beans is specified, the endpoint reference remains valid even if the stateful session bean is failed over.

  3. If you are creating an endpoint reference to represent any other object, create the endpoint reference using the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) API method. Because the endpoint reference might be workload managed at a later date, ensure that the endpoint does not contain any in-memory state.

  When the application invokes either of the above two methods, WebSphere Application Server 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 have 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 above methods will use the overridden value to generate the address URI for the endpoint reference.

Continue with "Creating a Web service application that is referenced through a Web Services Addressing endpoint reference" on page 479.

Example: Creating a Web service that uses the 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 Web Services Addressing (WS-Addressing) APIs 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 example above 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 below 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 returns the EndpointReference object, which now represents the new printer.

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
import com.ibm.websphere.wsaddressing.EndpointReference;
...
public Constants
{

// 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 EndpointReference createPrinter(java.lang.Object createPrinterRequest)
{
   public static final QName PRINTER_SERVICE_QNAME = new QName("example.printer.com", "printer", "...");
   public static final String PRINTER_ENDPOINT_NAME = new String("PrinterService");

 // 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 epr;
}
```

Extending the target service to match incoming messages to Web service resource instances

Because of the Web service implementation described earlier, the printer resource instance now has a unique identifier embedded in its endpoint reference. This identifier appears as a reference parameter in the SOAP header of subsequent messages 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 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 associated with the target endpoint from the MessageContext object, as illustrated below:

```
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 obtains a JAX-RPC Stub object in the normal J2EE fashion by looking up the Printer service in JNDI. The client then associates the EndpointReference object obtained previously with the Stub object, as illustrated below.

```
import javax.xml.rpc.Stub;
...
 // Associate the endpoint reference that represents the new printer with the Printer stub.
 Printer p = (Printer)((new PrinterServiceLocator()).getPort(Printer.class));
 Stub printerStub = (javax.xml.rpc.Stub)p;
 printerStub._setProperty(
        "com.ibm.websphere.wsaddressing.WSAConstants.
                        WSADDRESSING_DESTINATION_EPR ", epr);
```

The Stub 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 Stub object, WebSphere Application Server adds appropriate message addressing properties to the message header, in this case a reference parameter contained within the endpoint reference that identifies the target printer resource.

Alternatively, the client can use a Call object, which the client configures to represent the new printer as follows:

```
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 client's perspective 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 represented by the endpoint reference, as shown.

Web Services Addressing APIs:

WebSphere Application Server 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 these interfaces in conjunction with the Web Services Addressing (WS-Addressing) system programming interfaces.

The programming interfaces in this topic are described in more detail in the WS-Addressing API documentation.

The application programming interfaces are contained in the com.ibm.websphere.wsaddressing package and are summarized in the following diagram.
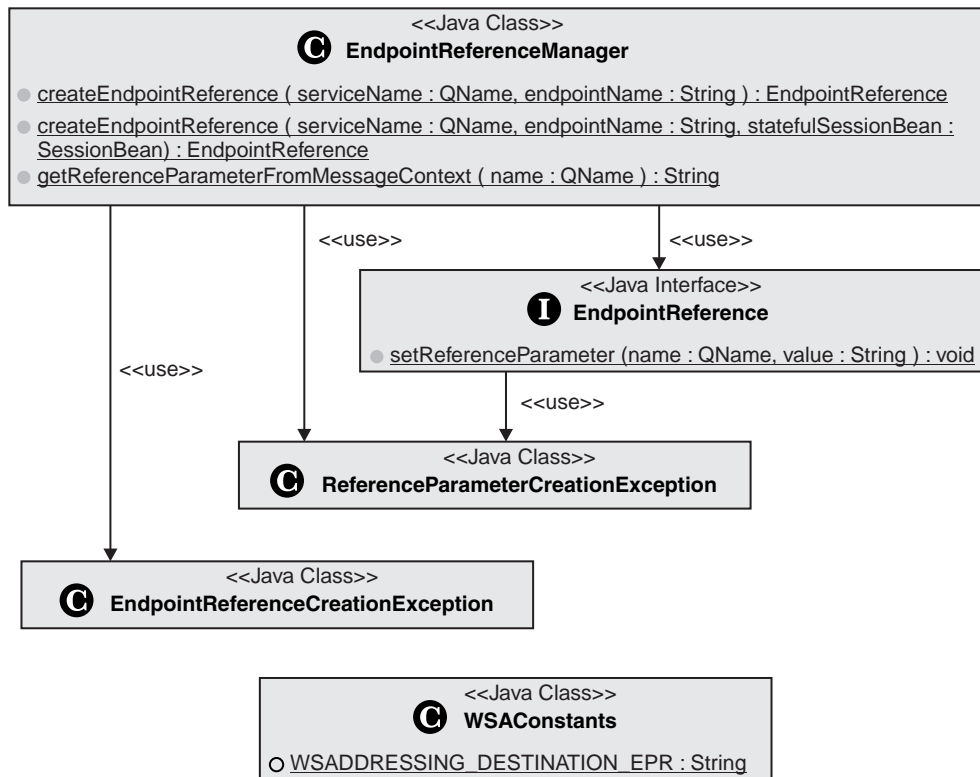
These interfaces provide the following features:

- A mechanism to create a com.ibm.websphere.EndpointReference instance to represent a WS-Addressing endpoint reference using the com.ibm.websphere.EndpointReferenceManager.createEndpointReference interface.

- An interface com.ibm.websphere.EndpointReference.setReferenceParameter to enable the association of reference parameters with the EndpointReference created above.

- An interface to enable a client to configure its Stub or Call object based on the EndpointReference instance created above. All invocations on the Stub or Call object will subsequently be targeted at the endpoint represented by the EndpointReference. To achieve this behavior, set the com.ibm.websphere.wsaddressing.WSAConstants.WSADDRESSING_DESTINATION_EPR property on the Stub or Call object to the EndpointReference in question.

- A mechanism to acquire individual reference parameters 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 WS-Addressing SPI: Performing more advanced Web Service Addressing tasks

WebSphere Application Server provides system programming interfaces for more advanced Web Services Addressing (WS-Addressing) tasks, that involve the WS-Addressing message addressing properties that are passed in the SOAP header of a Web service message. The SPI also provides interfaces to enable you to choose a WS-Addressing specification level other than the WebSphere Application Server default.

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

Perform this task in order 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 WebSphere Application Server.

- To manipulate message addressing properties, follow the instructions in "Specifying and acquiring message addressing properties using the Web Services Addressing SPI"
- 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 486.

Specifying and acquiring message addressing properties using the Web Services Addressing SPI:

The Web Services Addressing (WS-Addressing) SPI enables you to add WS-Addressing message addressing properties (MAPs) to the SOAP headers of an outbound message, through properties on the JAX-RPC Stub or Call interface. When the target endpoint receives the message, the SPI enables the endpoint to acquire the MAPs through properties on the message context.

Perform this task if you are a Web service developer using the WS-Addressing support, or a system programmer using the 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 detailed, along with the Java type of property instances, in "Web Services Addressing SPI" on page 487. 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 containing 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 that you are replying to. 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 Stub or Call object as described in "Creating a Web service application that is referenced through a Web Services Addressing endpoint reference" on page 479.
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 com.ibm.websphere.wsaddressing.EndpointReference, to use as the WSADDRESSING_REPLYTO_EPR property.

3. Set the required properties by associating them with the Stub or Call object, using the Stub or Call object's setProperty(String property_name, Object value) method. The following example sets a destination endpoint reference and a reply endpoint reference on a Call object:

```
import javax.xml.rpc.Call;
...
 // Associate the endpoint reference for the Web service. This property is required for the message
 // to be WS-Addressing compliant.
 call.setProperty(com.ibm.websphere.wsaddressing.WSAConstants.
                           WSADDRESSING_DESTINATION_EPR, destinationEpr);
 // Associate the endpoint reference that represents the reply to endpoint reference
 call.setProperty(com.ibm.wsspi.wsaddressing.WSAConstants.
                           WSADDRESSING_REPLYTO_EPR, replyToEpr);
```

When an invocation occurs on the Stub or Call object, WebSphere Application Server adds the appropriate MAPs to the message header.

4. On the server, retrieve the MAPs from the inbound message through the message context, using the MessageContext.getProperty(String propertyName) method. 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:

```
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();

   // 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 WebSphere Application Server. 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.

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 sending or receiving messages to or from an endpoint that supports only the 2004/08 namespace, you do not have to perform any additional steps for interoperability. WebSphere Application Server 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 that adhered to by the message. Any discrepancy will result in a 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, 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 WSDL document for your Web service, as follows: change `http://www.w3.org/2006/02/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 runtime, for export to an endpoint that supports the 2004/08 namespace only, perform the following extra steps.

  1. Create the endpoint reference to be exported.

  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 WS-Addressing SPI to retrieve the WSADDRESSING_INBOUND_NAMESPACE property from the inbound message context. This property specifies the Core WS-Addressing specification namespace of the of the incoming message.

  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 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
    }
  ```

Web Services Addressing SPI:

The Web Services Addressing (WS-Addressing) system programming interface extends the application programming interface 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.

The programming interfaces in this topic are described in more detail in the WS-Addressing SPI documentation.

Creating, refining and reasoning about the contents of endpoint references

The SPIs for creating, refining and reasoning about the contents of endpoint references are contained in the com.ibm.wsspi.wsaddressing package and are summarized below (the first two interfaces are API interfaces that are extended by the SPIs):

<<Java Interface>>
EndpointReference (I)

setReferenceParameter (name : QName, value : String) : void

<<Java Class>>
EndpointReferenceManager (C)

- createEndpointReference (serviceName : QName, endpointName : String) : EndpointReference
- createEndpointReference (serviceName : QName, endpointName : String, statefulSessionBean : SessionBean) : EndpointReference
- getReferenceParameterFromMessageContext (name : QName) : String

---

<<Java Class>>
EndpointReferenceManager (C)

- createEndpointReference (soapElement : SOAPElement) : EndpointReference
- createEndpointReference (address : URI) : EndpointReference
- createEndpointReference (address : AttributedURI) : EndpointReference
- getSOAPElementReferenceParameterFromMessageContext (name : QName) : SOAPElement

---

<<Java Interface>>
EndpointReference (I)

- getAddress ( ) : AttributedURI
- setAddress (address : AttributedURI) : void
- getReferenceParameter (referenceParameterName : QName) : String
- getReferenceParameter (name : QName, value : SOAPElement) : void
- getSOAPElementReferenceParameter (referenceParameterName : QName) : SOAPElement
- getMetadata ( ) : Metadata
- setMetadata (metadata : Metadata) : void
- getSOAPElement (rootNode : SOAPElement) : SOAPElement
- setNamespace (namespace : String) void
- clone ( ) : Object
- getNamespace ( ) : String

<<Java Interface>>
AttributedURI (I)

- getURI ( )
- setURI ( )
- clone ( )

<<Java Class>>
WSAddressingFactory (C)

- createAttributedQName ( )
- createAttributedURI ( )
- createRelationship ( )
- createMetadata ( )
- createServiceName ( )
- generateMessageID ( )

<<Java Interface>>
Metadata (I)

- getInterfaceName ( )
- setInterfaceName ( )
- getServiceName ( )
- setServiceName ( )

<<Java Interface>>
ServiceName (I)

- getQName ( )
- setQName ( )
- getEndpointName ( )
- setEndpointName ( )
- clone ( )

<<Java Class>>
WSAConstants (C)

- WSAADDRESSING_NAMESPACE_2005_08 : String
- WSAADDRESSING_NAMESPACE_2004_08 : String
- WSAADDRESSING_NAMESPACE : String

The SPI extends the 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 in order to access this additional functionality.

Similarly, the SPI com.ibm.wsspi.wsaddressing.EndpointReferenceManager extends the functionality provided in the com.ibm.websphere.wsaddressing.EndpointReferenceManager API.

The additional methods provided by the EndpointReference and EndpointReferenceManager SPIs allow you to perform the following actions:

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 provided at the API level, in that they do not automatically generate the URI for the EndpointReference. 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 de-serialize SOAP elements of type EndpointReferenceType into their corresponding EndpointReference Java representation, by using the EndpointReference.createEndpointReference(SOAPElement) operation. You might find these serialization and de-serialization interfaces useful if you are creating custom binders for types which contain EndpointReference instances.

Use more complex reference parameter types

The interfaces 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 which are not of type String from the incoming message.

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 artefacts making up an endpoint reference: Metadata, AttributedURI, and ServiceName. You create instances of these interfaces using operations provided by the WSAddressingFactory class.

Acquire and change the supported namespace

The WS-Addressing support in WebSphere Application Server supports multiple namespaces. The setNamespace and getNamespace operations provided on the EndpointReference interface enable you to change and acquire the namespace 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-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 WS-Addressing SPI provides a number of constants that identify JAX-RPC properties to set WS-Addressing MAPs on outbound messages, and message context properties to retrieve MAPs on inbound messages. These constants are shown in the diagram below in the class com.wsspi.wsaddresssing.WSAConstants. The diagram also shows the interfaces required for generating instances of the appropriate property value types AttributedURI and Relationship. The first WSAConstants interface is an API interface.

<<Java Class>>
C WSAConstants

○ WSADDRESSING_DESTINATION_EPR : String

<<Java Class>>
C WSAConstants

○ WSADDRESSING_FROM_EPR : String
○ WSADDRESSING_REPLYTO_EPR : String
○ WSADDRESSING_FAULTTO_EPR : String
○ WSADDRESSING_RELATIONSHIP_SET : String
○ WSADDRESSING_MESSAGE_ID : String
○ WSADDRESSING_ACTION : String
○ WSADDRESSING_INBOUND_TO : String
○ WSADDRESSING_INBOUND_ACTION : String
○ WSADDRESSING_INBOUND_MESSAGEID : String
○ WSADDRESSING_INBOUND_RELATIONSHIP_SET : String
○ WSADDRESSING_INBOUND_FROM_EPR : String
○ WSADDRESSING_INBOUND_REPLYTO_EPR : String
○ WSADDRESSING_INBOUND_FAULTTO_EPR : String
○ WSADDRESSING_INBOUND_NAMESPACE : String
○ WSADDRESSING_OUTBOUND_NAMESPACE : String

<<Java Class>>
C WSAddressingFactory

● createAttributedQName ( )
● createAttributedURI ( )
● createRelationship ( )
● createMetadata ( )
● createServiceName ( )
● generateMessageID ( )

<<use>>   <<use>>

<<Java Interface>>
I Metadata

● getInterfaceName ( )
● setInterfaceName ( )
● getServiceName ( )
● setServiceName ( )

<<Java Interface>>
I AttributedURI

● getURI ( )
● setURI ( )
● clone ( )

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-RPC Stub or Call interface prior to invoking a message in the Stub or Call object. The table below summarizes the relevant properties and their types.

Table 15. Outbound properties that you can set on the Stub or Call object, their Java types and equivalent abstract WS-Addressing MAP name or names.

JAX-RPC Stub or Call property name (of type String)	Java type of property value	Abstract WS-Addressing MAP name or names	Default value when the Stub or Call object property is not set
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]	Not set, unless the message is a one-way message with no reply
WSADDRESSING_ FAULT_EPR	com.ibm.websphere.wsaddressing. EndpointReference	[fault endpoint]	Not set
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
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 defined in the WSAConstants class. The following table summarizes the available inbound properties. (You can acquire reference parameters from the message context using the EndpointReferenceManager.getReferenceParameter interface.)

Table 16. 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]

Table 16. Inbound properties that you can acquire from the message context, their Java types and equivalent abstract WS-Addressing MAP name. (continued)

Message context property name (of type String)	Java type of property value	Abstract WS-Addressing MAP name
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

Web Services Addressing (WS-Addressing) is a W3C specification that aids interoperability between Web services, by defining a standard way to address Web services and provide addressing information in messages. To enable the WS-Addressing support, you must either configure the Web Services Description Language (WSDL) file for a service that runs on WebSphere Application Server, or use the WS-Addressing API or SPI to add WS-Addressing properties in a WebSphere Application Server client.

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.

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, specifying that WS-Addressing information is accepted but not required in incoming messages, or true, specifying that WS-Addressing information is required in incoming messages. The default value is false. Messages from WebSphere Application Server Version 6.1 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 WebSphere Application Server). 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 17. The behavior of the WS-Addressing support in WebSphere Application Server

	The WSDL for the service specifies UsingAdressing required = ″false″	The WSDL for the service specifies UsingAdressing required = ″true″	The WSDL for the service does not specify UsingAdressing
A client sends a message that contains WS-Addressing information	The WS-Addressing information is processed by WebSphere Application Server.	The WS-Addressing information is processed by WebSphere Application Server.	The WS-Addressing information is processed by WebSphere Application Server.
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 WebSphere Application Server.	The message automatically contains the mandatory WS-Addressing information as defined in the WS-Addressing specification. The information is processed by WebSphere Application Server.	WS-Addressing information is not added by WebSphere Application Server. The message is accepted.

- To enable WS-Addressing support from the server, 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:

    ```
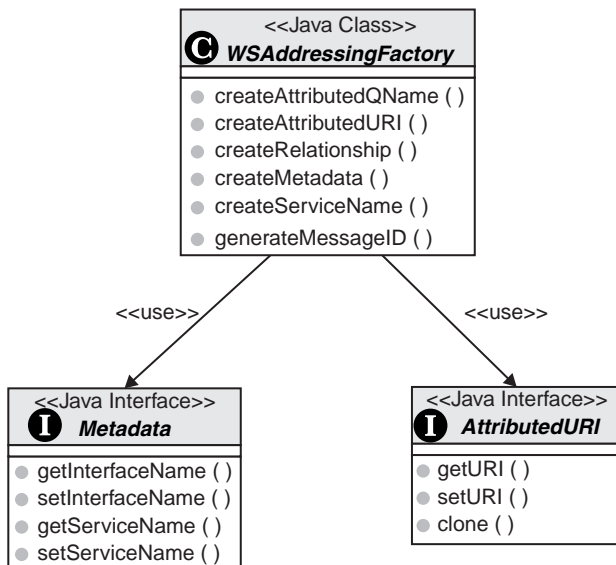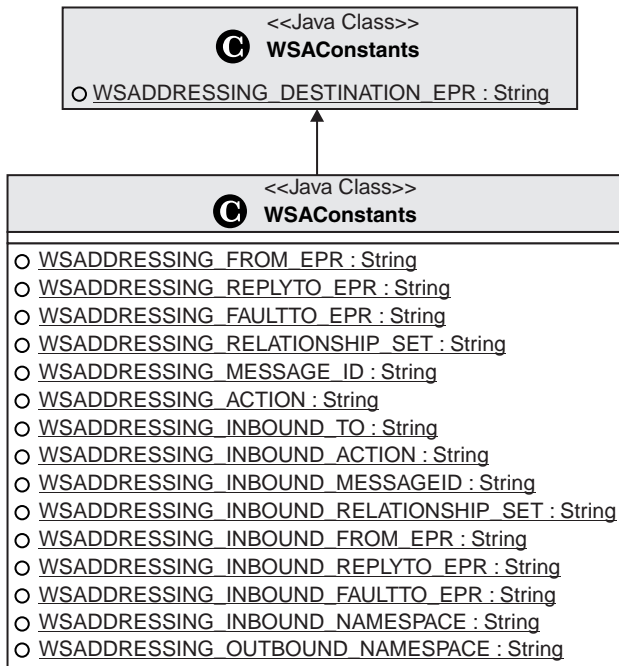    <wsdl:binding name="TestServiceSoapBinding" type="intf:TestService">
        <wsaw:UsingAddressing wsdl:required="false"
                                xmlns:wsaw="http://www.w3.org/2006/02/addressing/wsdl"/>

        <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 understands WS-Addressing information, but 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, 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 whose binding specifies the UsingAddressing element.

- To enable WS-Addressing support from a WebSphere Application Server client, use the WS-Addressing API or SPI provided, 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 receiving 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) which do not support the UsingAddressing element, or when the target endpoint's WSDL file is not available to the client.

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 an a WS-Addressing endpoint reference. You develop WS-Resources in the same way as ordinary Web services using the same tools, however there are some additional tasks that you must perform, as described in this topic.

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 that the WS-Resource provides access for. This resource component can be an existing legacy system or entity, or it can be a new component. There are no constraints on how you implement the resource; it can be a simple Java class, a stateless session EJB, 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 the WebSphere Application Server Toolkit, 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" on page 441 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 505 for an example of a CreatePrinter operation that returns an EndpointReference to 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 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/2006/02/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 Web service application that is referenced through a Web Services Addressing endpoint reference" on page 479 to create the implementation of the WS-Resource, enable the client to access the WS-Resource using an endpoint reference, and deploy the application.

Review "Example: Creating a Web service that uses the Web Services Addressing API to access a Web Services Resource (WS-Resource) instance" on page 505 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 execution of the succeeding ones. 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 made 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 figure below illustrates a Web service, at http://www.example.com/service, and three resources, A, B and C, that are accessed through the Web service. There are thus three WS-Resources illustrated in the figure:

A WS-Resource is referenced by a WS-Addressing endpoint reference which 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 example above WS-Resource-C is the combination of the Web service and the resource identified by "C", and a reference to WS-Resource-C might appear 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 which interact with the WS-Resource.

For example, consider a Printer WS-Resource which 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 would declare 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 it 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 expected of a WS-Resource when it processes these messages; the specifications do not prescribe the means to implement WS-Resources. 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 makes use of WS-Addressing endpoint references and, for the most part, the programming model for WS-Resources is the same as for any Web service that uses WS-Addressing; this is described in "Web Services Addressing application programming model" on page 474.

WSRF extends the WebSphere Application Server WS-Addressing programming model in two ways, which therefore 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 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 desired to represent the stateful resource and to process the WSRF messages; you can implement a resource using a simple Java class, a stateless session EJB, a entity bean backed by a relational database, a Service Data Object (SDO), or anything else.

• 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 is useful in complex systems where faults night 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 which can be used to indicate the originator of the fault.

• Other elements which 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 which 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>
```

WebSphere Application Server provides Java code mappings for all the base fault element types 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 itself 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, it must implement the
java.io.Serializable interface. To meet this requirement, all properties of a BaseFault instance must
themselves be serializable. Because the javax.xml.soap.SOAPElement class is not serializable,
WebSphere Application Server provides a 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 of type PrinterFaultType
shown above, you must provide a custom binder to define how the Web services runtime 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, located in the /META-INF/services directory of the JAR. 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 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 below.

Table 18. Principle WSRF operations 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.

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.

Table 18. Principle WSRF operations supported by WS-Resources (continued)

Operation	Description
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/> ```
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 be deleted. **Response format** ``` <wsrf-rp:DeleteResourcePropertiesResponse/> ```

Table 18. Principle WSRF operations supported by WS-Resources (continued)

Operation	Description

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 18. Principle WSRF operations 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
	<pre><wsrf-rl:SetTerminationTime>
	  [<wsrf-rl:RequestedTerminationTime>
	    xsd:dateTime
	  </wsrf-rl:RequestedTerminationTime>]
	       |
	  [<wsrf-rl:RequestedLifetimeDuration>
	    xsd:duration
	  </wsrf-rl:RequestedLifetimeDuration>]
	</wsrf-rl:SetTerminationTime></pre>
	where the termination time is either an absolute time or a relative duration.
	
	Response format
	<pre><wsrf-rl:SetTerminationTimeResponse>
	  <wsrf-rl:NewTerminationTime>
	    xsd:dateTime
	  </wsrf-rl:NewTerminationTime>
	  <wsrf-rl:CurrentTime>
	    xsd:dateTime
	  </wsrf-rl:CurrentTime>
	<wsrf-rl:SetTerminationTimeResponse></pre>
	This response contains the time, from the WS-Resource's perspective, when the WS-Resource will destroy itself. The response also contains the WS-Resource's value of the current time.
	
	There are a variety of ways in which a WS-Resource can implement scheduled destruction. For example, a WS-Resource that is implemented as an EJB might use the EJB 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. As described in the WS-Resource specification, 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.

Creating a resource properties schema document for the WS-Resource

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 below.

```
<?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 Web Services Addressing API to access a generic Web service resource instance" on page 481, with the addition of a ResourceProperties attribute on the wsdlPortType element. This attribute declares that the portType 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 operation GetResourceProperty; this 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 Web Services Addressing API to access a generic Web service resource instance" on page 481. 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.

Getting started with the UDDI registry

This section covers the basic knowledge you need to get started either as an administrator of a UDDI registry or as a user of a UDDI registry that has already been set up.
- Getting started for UDDI Administrators
- Getting started for UDDI users

Getting started for UDDI Administrators

Use this topic if you are involved in installing (setting up and deploying), customizing, or managing a UDDI registry.

This section contains a list of some of the topics that you will need to refer to as an administrator of a UDDI registry:
- UDDI registry Terminology introduces some terms with which you will need to be familiar in order to administer a UDDI registry
- Setting up and deploying a new UDDI registry explains how to install a UDDI registry node by setting up the resources that it will use, and deploying the UDDI registry application.

- Migrating the UDDI registry explains how to use the scripts provided to migrate UDDI registries from the previous version of WebSphere Application Server.
- Migrating to Version 3 of the UDDI registry explains how to migrate UDDI Version 2 registries to UDDI Version 3, introduced in WebSphere Application Server Version 6.0, by creating a migration datasource.
- Configuring UDDI registry security explains how to configure different levels of security for the UDDI registry, and how other UDDI node settings can influence security.
- Managing the UDDI registry explains how to use the UDDI pages in the administrative console, or the UDDI registry Administrative interface, to administer a UDDI registry node. It also covers how to back up and restore your UDDI registry data.
- UDDI node settings explains how to view and set UDDI properties and policies, and how to manage UDDI publishers, tiers and user entitlements.
- UDDI registry Management Interfaces covers details of programmatic interfaces that you can use to administer a UDDI registry node (UDDI registry Administrative (JMX) Interface), to add custom Value Set data to a UDDI registry (User Defined Value Set Support), and to export and import UDDI version 2 entities (UDDI Utility Tools).
- Removing and reinstalling the UDDI registry

UDDI registry troubleshooting might be useful if you encounter any problems or unexpected behavior while using the UDDI registry, and Troubleshooter reference: Messages links to messages that you might see.

Getting started for UDDI users

Use this topic if you use a UDDI registry to publish or find UDDI entities either through a user interface or by writing UDDI client applications.

This section contains a list of some of the topics that you might want to refer to as a user of a UDDI registry:
- UDDI registry Terminology introduces some terms with which you might need to be familiar in order to use a UDDI registry.
- Using the UDDI registry user interface explains how to access the UDDI User Console, which is a user interface that allows you to find UDDI entities and carry out simple UDDI publish operations.
- UDDI registry SOAP Service End Points contains details for accessing the UDDI version 3 Inquiry, Publish, Security, and custody transfer APIs, as well as the UDDI version 1 and version 2 APIs.
- UDDI registry Client Programming explains how to write UDDI client application programs. The recommended client programming interface is the UDDI version 3 Client for Java.
- IBM JAXR Provider for the UDDI registry is for users who want to use the Java API for XML Registries to access UDDI.
- User Defined Value Set Support in the UDDI registry explains how to add custom value set data to a UDDI registry.

UDDI registry troubleshooting might be useful if you encounter any problems or unexpected behavior while using the UDDI registry, and Troubleshooter reference: Messages links to messages that you might see.

Using the UDDI registry user interface

Configure the application server hosting the UDDI registry for UTF-8 encoding support, as described in refer to Configuring application servers for UTF-8 encoding.

This topic describes the UDDI registry user interface (also referred to as the UDDI registry user console), which you can use to explore the UDDI registry.

The user console provides a graphical user interface to the majority of the UDDI Version 3 API. It is not intended to support the full API set. There is some focus on inquiry operations, as the main purpose of the

UDDI user console is to allow you to issue inquiry requests and to familiarize yourself with general UDDI concepts. This section documents the areas for which support through the user console is not provided, and other known restrictions to the user console.

- General
  - Help is provided in the form of explanatory text on the screens.
  - Maximum rows cannot be specified on finds. The single maximum rows value for the registry can be set through the *Maximum inquiry result set size* general property on the administrative console.
- Find business
  - The identifier feature is not supported.
- Find technical model (tModel)
  - The identifier feature is not supported.
- Add business
  - There is no support for adding Discovery URLs, Identifiers or Digital Signatures.
- Add technical model (tModel)
  - There is no support for adding Identifiers or Digital Signatures.
- Business Relationships
  - There is no support for Business Relationships

Use the related links below to find the appropriate topic for the task you want to perform using the UDDI registry user interface.

Displaying the UDDI registry user interface

Use this task to perform actions on UDDI information through the UDDI user console. The exact behavior of the user console depends on several configurable factors, such as:

- 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 19.

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. |

Table 19. (continued)

WebSphere Application Server security status	URL used to access the UDDI user console	Behavior of the UDDI user console
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

The user console displays the default frameset containing the following items:
* The header frame.
* The navigation frame showing find options.
* The details frame.

You can now use the UDDI user console to find, edit or publish UDDI information.

Finding an entity using the UDDI registry user interface

Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in Displaying the user interface.

This topic 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

Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in Displaying the user interface.

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

Make sure that the UDDI registry application is started, then display the UDDI registry user interface as described in Displaying the user interface.

This topic 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, including adding 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*.

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.

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 Displaying the user interface.

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 510.

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.

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.

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 Displaying the user interface.

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.

Setting up and deploying a new UDDI registry

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.

A UDDI registry node consists of the UDDI registry application (a J2EE 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). The 'Setting up' sub-topics describe how to create the 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 534.

Proceed to one of the following topics:

- "Setting up a default UDDI node"
- "Setting up a customized UDDI node" on page 524

Database considerations for production use of the UDDI registry

The UDDI registry fully supports a number of databases (see An overview of the Version 3 UDDI registry for details) and can be used for development and test purposes, however you should be aware of the following factors when considering which database is appropriate for your anticipated UDDI registry production use.

It is important to consult the information supplied by your chosen database vendor for advice, but additionally you need to consider the likely size and volume of requests, and whether the general performance and scalability of the UDDI registry is important to you.

For example, while Cloudscape supports the full function of the UDDI registry, it is not an enterprise level database and consequently it does not have the same characteristics (for example, scaling or performance) as enterprise databases such as DB2.

If you need multiple connections to the UDDI registry database (for example if you are using the UDDI registry in a cluster configuration) and Cloudscape is your preferred database, you will need to use the network Cloudscape option as embedded Cloudscape has a limitation of allowing only one Java virtual machine to access or load a database instance at any one time (in other words two application servers cannot access the same Cloudscape database instance at the same time).

Note: The UDDI registry can support multiple users even if there is a single database connection.

More information on Cloudscape is available in this information center.

Setting up a default UDDI node

Use this task to create a UDDI node with predetermined property values. You will not be able to change the mandatory node properties, such as node ID, either during the creation of the node or afterwards. Such a node is suitable for initial evaluation of the UDDI registry and for development and test purposes. If you want to choose the mandatory node properties yourself, set up a customized node as detailed in "Setting up a customized UDDI node" on page 524.

Note: If you are deploying the UDDI registry application into a cluster, and you want to use Cloudscape for your database, you must use the network version of Cloudscape. Embedded Cloudscape 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 516
   - "Creating a DB2 for z/OS database for the UDDI registry" on page 518
   - "Creating a Cloudscape database for the UDDI registry" on page 519
   - "Creating an Oracle database for the UDDI registry" on page 520

Note: If you are creating the UDDI node in a cluster, it is assumed that a single database will be used for all members of the cluster.

2. Set up a datasource for the UDDI registry application to use to access the database, as described in "Creating a data source for the UDDI registry" on page 521.

3. Deploy the UDDI registry application, as described in "Deploying the UDDI registry application" on page 523.

4. `Linux` If you are using DB2, stop the server if it is running. Edit the user profile for the DB2 user that will start the node agent or server. Modify the user profile to run the db2profile script which is located in the root directory of the DB2 userid (for example /home/db2inst1/sqllib/db2profile). Alternatively you can run the db2profile script manually by entering the following command, however you will need to do this every time you restart the node agent or server :

   ```
   . /home/db2inst1/sqllib/db2profile
   ```

   Note: In the above example, notice that the '.' is followed by a single space character.

5. Click **Applications** → **Enterprise Applications** to display the installed applications. Start the UDDI registry application by selecting the check box next to it and clicking **Start**. Alternatively start the application server if it is not already running; this will automatically start the UDDI registry application. The UDDI node is now active.

   Note: Restarting the UDDI application, or the application server, will always result in the reactivation of the UDDI node, even if the node was previously deactivated.

6. 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.

As 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 533 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.

The steps below use a number of variables for which you need to enter appropriate values. You should decide the values that you will use before you start. The variables used, and suggested values are:

<DataBaseName>
   is the name of the UDDI registry database. A recommended value is UDDI30, and this name is assumed throughout the UDDI documentation. If you use some other name, then you should substitute that name whenever you see 'UDDI30' in other sections of the documentation.

<DB2UserID>
   is a DB2 userid with administrative privileges.

<DB2Password>
   is the password for the DB2 userid.

<BufferPoolName>
   is 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, as the buffer pool is created as part of this task.

<TableSpaceName>
   is the name of a table space. A suggested value is uddits, but any name can be used.

<TempTableSpaceName>
is the name of a temporary table space. A suggested value is udditstemp, but any name can be used, as the temporary table space is created as part of this task.

You only need to perform this task once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

1. Change directory to *app_server_root*/UDDIReg/databaseScripts.
2. Start the DB2 Command Line Processor by entering `db2` at the command prompt (for Windows platforms, enter `db2cmd` and then enter `db2` in the new DB2 window).
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. Run this step only if you want your database to be used as a default UDDI node. Enter the following command:

   ```
   db2 -tvf uddi30crt_70_insert_default_database_indicator.sql
   ```

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.

In order to connect from a machine running a distributed operating system to a remote DB2 database on the z/OS operating system, you must have DB2 Version 8.2 or later. You must also have a DB2 Connect license (see the DB2 documentation for more information).

You only need to perform this task once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

1. Copy the createddl.sh script 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 you have installed WebSphere Application Server in a non default location, update the root_dir constant to reflect this (note that the UDDIReg directory must remain at the end of the path).

   c. Update the temp_dir constant to a temporary directory of your choice, if you do not want to accept the default.

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
   > This is the name which will be used when defining the required DB2 tables and other components. The default is UDDI30.

   tablespace_name
   > This is the tablespace in which the database's tables will be defined. The default is UDDI30TS.

   hlq   This is the high level qualifier under which the SQL and JCL partitioned datasets (PDS) will be created. The default is IBMUSER.

   The script generates the partitioned data sets *hlq*.UDDI.SQL and *hlq*.UDDI.JCL, containing members that are required for subsequent steps.Using the default parameters listed above, a successful execution of the script results in 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.

Continue with setting up and deploying your UDDI registry node.

Creating a Cloudscape database for the UDDI registry:

Perform this task if you want to use Cloudscape (embedded or network) as the database store (either local or remote) for your UDDI registry. You need only perform this task once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

Before you begin

The commands below use a number of arguments for which you need to enter appropriate values. You should decide the values that you will use before you start. The arguments used, and suggested values, are:

arg1   is the path of the SQL files, which on a standard installation will be *app_server_root*/UDDIReg/databasescripts

arg2   is the path to the location where you want to install the Cloudscape database.

        For example, *app_server_root*/profiles/*profile_name*/databases/com.ibm.uddi

arg3   is the name of the Cloudscape database. A recommended value is UDDI30, and this name is

assumed throughout the UDDI documentation. If you use another name, you should substitute that name whenever you see 'UDDI30' in other sections of the UDDI documentation.

arg4   is an optional argument, and must either be omitted or be the string 'DEFAULT'. DEFAULT should only be specified if you want your database to be used as a default UDDI node. Note that this argument is case sensitive.

1. Run the following Java -jar command from the *app_server_root*/UDDIReg/databaseScripts directory, to create a UDDI Cloudscape database using UDDIDerbyCreate.jar. `Windows`

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib;app_server_root/java/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   `AIX`   `Linux`   `HP-UX`   `Solaris`

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib:app_server_root/java/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   If the Cloudscape database already exists, you will be asked if you want to recreate it. If you choose to recreate the database, your existing database will be deleted and a new one created in its place. If you choose not to recreate the database, the command will exit and a new database will not be created.

   Note: If the application server already accessed the existing Cloudscape 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. If you are using a remote database (which requires network Cloudscape), or you want to use network Cloudscape for other reasons, for example if you want to use Cloudscape with a cluster, configure the Cloudscape Network Server framework as described in the managing derby network server section of the Cloudscape information center.

Continue with setting up and deploying your UDDI registry node.

Creating an Oracle database for the UDDI registry:

Note: Only Version 9i[1] and Oracle 10g[2]

Perform this task if you want to use Oracle as the database store (either local or remote) for your UDDI registry data. You need only do this once for each UDDI registry, as part of Setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

1.

   Restrictions:
   discoveryURL (Business) maximum 4000 bytes, UDDI specification 4096 characters; accessPoint (bindingTemplate) maximum 4000 bytes, UDDI Specification 4096 characters; instanceParms (tModelInstanceInfo) maximum 4000 bytes, UDDI specification 8192 characters; overviewURL (tModelInstanceInfo) maximum 4000 bytes, UDDI specification 4096 characters; Digital Signature maximum 4000 bytes.

2.

   Restrictions:
   discoveryURL (Business) maximum 4000 bytes, UDDI specification 4096 characters; accessPoint (bindingTemplate) maximum 4000 bytes, UDDI Specification 4096 characters.

Before you begin

This task creates three new schemas: ibmuddi, ibmudi30 and ibmuds30. You will be unable to complete this task if you already have existing schemas with these names.

The steps below use a number of variables for which you need to enter appropriate values. You should decide the values that you will use before you start. The variables used are:

<OracleUserID>
   is the Oracle userid to be used to create the database.

<OraclePassword>
   is the password for the Oracle userid.

1. Run the following commands:
   a. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_10_prereq_oracle.sql`
   b. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_20_tables_generic.sql`
   c. Complete one of the following actions depending on your level of Oracle:
      - For Oracle 9i:
        `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_25_tables_oracle_pre10g.sql`
      - For Version 10g and later:
        `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_25_tables_oracle.sql`
   d. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_30_constraints_generic.sql`
   e. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_35_constraints_oracle.sql`
   f. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_40_views_generic.sql`
   g. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_45_views_oracle.sql`
   h. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_50_triggers_oracle.sql`
   i. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_60_insert_initial_static_data.sql`
2. This last command should only be run if you want the database to be used as a default UDDI node.
   `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_70_insert_default_database_indicator.sql`

Continue with setting up and deploying your UDDI registry node.

Creating a data source for the UDDI registry:

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.

Note: If you are connecting to a remote DB2 database on the z/OS operating system, you must have installed a DB2 Connect license (See the DB2 documentation for more information).

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 (not required for embedded Cloudscape, but required for network Cloudscape):
   a. Click **Security** → **Secure administration, applications, and infrastructire** → **[Authentication] Java Authentication and Authorization Service** → **J2C authentication data**.
   b. Click **New** to create a new J2C authentication data entry
   c. Fill in the details as follows:

      Alias   a suitable (short) name, such as ″UDDIAlias″

      Userid

the database userid (such as db2admin for DB2 or IBMUDDI for Oracle), which is used to read and write to the UDDI registry database. For network Cloudscape the userid can be any value.

If you are using a remote DB2 database on the z/OS operating system, the userid must be one that is valid on the remote system.

Password
the password associated with the userid specified above. For network Cloudscape the password can be any value.

Description
a suitable description to describe the chosen userid.

Click **Apply** and 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
Oracle	Oracle JDBC Driver	Connection Pool data source
Embedded Cloudscape	Derby JDBC Driver	Connection Pool data source
Network Cloudscape	Derby Network Server JDBC Driver provider	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. Create the data source for the UDDI registry by following these steps:

   a. Click **Resources** → **JDBC** → **JDBC Providers**.

   b. Select the desired 'scope' of the JDBC provider you selected or created earlier. For example, select:

      `Server: yourservername`

      to show the JDBC providers at the server level.

   c. Select the JDBC provider created earlier.

   d. Under **Additional Properties**, select **Data sources** (not 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:

   g. Click **Next**.

   h. On the database specific properties page of the wizard, enter the following data:

   i. Click **Next**, then check the summary and click **Finish**.

   j. Click the data source to display its properties, and add the following information:

   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**. You will see a message similar to ″Test Connection for datasource UDDI Datasource on server server1 at node MyNode was successful″. If you do not see this message investigate the problem with the help of the error message.

Continue with setting up and deploying your UDDI registry node.

Deploying the UDDI registry application:
You must have already created the database and datasource for the UDDI registry.

Use this task as part of "Setting up a default UDDI node" on page 515 or "Setting up a customized UDDI node" on page 524.

Run the uddiDeploy.jacl script as shown below, from the *app_server_root*/bin directory. This script deploys the UDDI registry to a server (or cluster) specified by you.

▶ Linux

Note: If you are using either the UNIX or Linux operating systems, add the `.sh` suffix to the wsadmin command.
```
wsadmin [-conntype none] [-profileName profile_name] -f uddiDeploy.jacl
            {node_name server_name | cluster_name}
```

where
- '-profileName *profile_name*' is the deployment manager profile. If you do not specify a profile, the default profile will be used.
- '-conntype none' is optional, and is only needed if the application server is not running.
- *node_name* is the name of the WebSphere node on which the target server runs. Note that the node name is case sensitive.
- *server_name* is the name of the target server on which you wish to deploy the UDDI registry, such as server1. Note the server name entered is case sensitive.
- *cluster_name* is the name of the target cluster into which you wish to deploy the UDDI registry. Note that the cluster name entered is case sensitive.

For example, to deploy UDDI on node 'MyNode' and server 'server1' on a Windows system, (assuming that server1 is already started):
```
wsadmin -f uddiDeploy.jacl MyNode server1
```

To deploy UDDI into cluster 'MyCluster' on a Windows system:
```
wsadmin -f uddiDeploy.jacl MyCluster
```

You can also deploy the UDDI application (the uddi.ear file) using the administrative console, in the normal way. However, some steps that are performed automatically by the uddiDeploy.jacl script do not take place in this scenario. If you use the administrative console to install the UDDI application, you must perform some actions manually, according to the following steps:
1. Install the application.
2. Click **Applications** → **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**.

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.
```

Continue with setting up the UDDI node.

Setting up a customized UDDI node

Use this task to set up a UDDI node with property values that are chosen by you. You will not be able to 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 are deploying the UDDI registry application into a cluster, and you want to use Cloudscape for your database, you must use the network version of Cloudscape. Embedded Cloudscape is not supported for cluster configurations.

1. Review the information in "Database considerations for production use of the UDDI registry" on page 515 to help you decide which database system to use, then create a database schema to hold the UDDI registry 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 516
   - "Creating a DB2 for z/OS database for the UDDI registry" on page 518
   - "Creating a Cloudscape database for the UDDI registry" on page 519
   - "Creating an Oracle database for the UDDI registry" on page 520

   Note: If you are creating the UDDI node in a cluster, it is assumed that a single database will be used for all members of the cluster.

2. Set up a datasource for the UDDI registry application to use to access the database, as described in "Creating a data source for the UDDI registry" on page 521.

3. Deploy the UDDI registry application, as described in "Deploying the UDDI registry application" on page 523.

4. ▶ `Linux` If you are using DB2, stop the server if it is running. Edit the user profile for the DB2 user that will start the node agent or server. Modify the user profile to run the db2profile script which is located in the root directory of the DB2 userid (for example /home/db2inst1/sqllib/db2profile). Alternatively you can run the db2profile script manually by entering the following command, however you will need to do this every time you restart the node agent or server :

   ```
   . /home/db2inst1/sqllib/db2profile
   ```

   Note: In the above example, notice that the '.' is followed by a single space character.

5. Click **Applications** → **Enterprise Applications** to display the installed applications. Start the UDDI registry application by selecting the check box next to it and clicking **Start**. Alternatively start the application server if it is not already running; this will automatically start the UDDI registry application. The UDDI node is now active.

   Note: Restarting the UDDI application, or the application server, will always result in the reactivation of the UDDI node, even if the node was previously deactivated.

As you have chosen a user customized UDDI node, you will 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 532 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.

The steps below use a number of variables for which you need to enter appropriate values. You should decide the values that you will use before you start. The variables used, and suggested values are:

<DataBaseName>
  is the name of the UDDI registry database. A recommended value is UDDI30, and this name is

assumed throughout the UDDI documentation. If you use some other name, then you should substitute that name whenever you see 'UDDI30' in other sections of the documentation.

<DB2UserID>
> is a DB2 userid with administrative privileges.

<DB2Password>
> is the password for the DB2 userid.

<BufferPoolName>
> is 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, as the buffer pool is created as part of this task.

<TableSpaceName>
> is the name of a table space. A suggested value is uddits, but any name can be used.

<TempTableSpaceName>
> is the name of a temporary table space. A suggested value is udditstemp, but any name can be used, as the temporary table space is created as part of this task.

You only need to perform this task once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

1. Change directory to *app_server_root*/UDDIReg/databaseScripts.
2. Start the DB2 Command Line Processor by entering `db2` at the command prompt (for Windows platforms, enter `db2cmd` and then enter `db2` in the new DB2 window).
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. Run this step only if you want your database to be used as a default UDDI node. Enter the following command:

    `db2 -tvf uddi30crt_70_insert_default_database_indicator.sql`

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.

In order to connect from a machine running a distributed operating system to a remote DB2 database on the z/OS operating system, you must have DB2 Version 8.2 or later. You must also have a DB2 Connect license (see the DB2 documentation for more information).

You only need to perform this task once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

1. Copy the createddl.sh script 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 you have installed WebSphere Application Server in a non default location, update the root_dir constant to reflect this (note that the UDDIReg directory must remain at the end of the path).

    c. Update the temp_dir constant to a temporary directory of your choice, if you do not want to accept the default.

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
        This is the name which will be used when defining the required DB2 tables and other components. The default is UDDI30.

tablespace_name
> This is the tablespace in which the database's tables will be defined. The default is UDDI30TS.

hlq This is the high level qualifier under which the SQL and JCL partitioned datasets (PDS) will be created. The default is IBMUSER.

The script generates the partitioned data sets *hlq*.UDDI.SQL and *hlq*.UDDI.JCL, containing members that are required for subsequent steps.Using the default parameters listed above, a successful execution of the script results in 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.

Continue with setting up and deploying your UDDI registry node.

Creating a Cloudscape database for the UDDI registry:

Perform this task if you want to use Cloudscape (embedded or network) as the database store (either local or remote) for your UDDI registry. You need only perform this task once for each UDDI registry, as part of setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

Before you begin

The commands below use a number of arguments for which you need to enter appropriate values. You should decide the values that you will use before you start. The arguments used, and suggested values, are:

arg1   is the path of the SQL files, which on a standard installation will be *app_server_root*/UDDIReg/ databasescripts

arg2   is the path to the location where you want to install the Cloudscape database.

   For example, *app_server_root*/profiles/*profile_name*/databases/com.ibm.uddi

arg3   is the name of the Cloudscape database. A recommended value is UDDI30, and this name is assumed throughout the UDDI documentation. If you use another name, you should substitute that name whenever you see 'UDDI30' in other sections of the UDDI documentation.

arg4   is an optional argument, and must either be omitted or be the string 'DEFAULT'. DEFAULT should only be specified if you want your database to be used as a default UDDI node. Note that this argument is case sensitive.

1. Run the following Java -jar command from the *app_server_root*/UDDIReg/databaseScripts directory, to create a UDDI Cloudscape database using UDDIDerbyCreate.jar. ▶ Windows

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib;app_server_root/java/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   ▶ AIX   ▶ Linux   ▶ HP-UX   ▶ Solaris

   ```
   java -Djava.ext.dirs=app_server_root/derby/lib:app_server_root/java/lib/ext -jar UDDIDerbyCreate.jar
   arg1 arg2 arg3 arg4
   ```

   If the Cloudscape database already exists, you will be asked if you want to recreate it. If you choose to recreate the database, your existing database will be deleted and a new one created in its place. If you choose not to recreate the database, the command will exit and a new database will not be created.

   Note: If the application server already accessed the existing Cloudscape 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. If you are using a remote database (which requires network Cloudscape), or you want to use network Cloudscape for other reasons, for example if you want to use Cloudscape with a cluster, configure the Cloudscape Network Server framework as described in the managing derby network server section of the Cloudscape information center.

Continue with setting up and deploying your UDDI registry node.

Creating an Oracle database for the UDDI registry:

Note: Only Version 9i[3] and Oracle 10g[4]

3.

> **Restrictions:**
>
> discoveryURL (Business) maximum 4000 bytes, UDDI specification 4096 characters; accessPoint (bindingTemplate) maximum 4000 bytes, UDDI Specification 4096 characters; instanceParms (tModelInstanceInfo) maximum 4000 bytes, UDDI specification 8192 characters; overviewURL (tModelInstanceInfo) maximum 4000 bytes, UDDI specification 4096 characters; Digital Signature maximum 4000 bytes.

4.

Perform this task if you want to use Oracle as the database store (either local or remote) for your UDDI registry data. You need only do this once for each UDDI registry, as part of Setting up and deploying a UDDI registry.

If you are creating a remote database, use the database product documentation to familiarize yourself with the relevant capabilities of the product before proceeding with this task.

Before you begin

This task creates three new schemas: ibmuddi, ibmudi30 and ibmuds30. You will be unable to complete this task if you already have existing schemas with these names.

The steps below use a number of variables for which you need to enter appropriate values. You should decide the values that you will use before you start. The variables used are:

<OracleUserID>
        is the Oracle userid to be used to create the database.

<OraclePassword>
        is the password for the Oracle userid.

1. Run the following commands:

   a. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_10_prereq_oracle.sql`

   b. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_20_tables_generic.sql`

   c. Complete one of the following actions depending on your level of Oracle:

      - For Oracle 9i:

        `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_25_tables_oracle_pre10g.sql`

      - For Version 10g and later:

        `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_25_tables_oracle.sql`

   d. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_30_constraints_generic.sql`

   e. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_35_constraints_oracle.sql`

   f. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_40_views_generic.sql`

   g. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_45_views_oracle.sql`

   h. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_50_triggers_oracle.sql`

   i. `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_60_insert_initial_static_data.sql`

2. This last command should only be run if you want the database to be used as a default UDDI node.

   `sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_70_insert_default_database_indicator.sql`

Continue with setting up and deploying your UDDI registry node.

Creating a data source for the UDDI registry:

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.

Note: If you are connecting to a remote DB2 database on the z/OS operating system, you must have installed a DB2 Connect license (See the DB2 documentation for more information).

Restrictions:
        discoveryURL (Business) maximum 4000 bytes, UDDI specification 4096 characters; accessPoint (bindingTemplate) maximum 4000 bytes, UDDI Specification 4096 characters.

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 (not required for embedded Cloudscape, but required for network Cloudscape):

   a. Click **Security** → **Secure administration, applications, and infrastructire** → **[Authentication] Java Authentication and Authorization Service** → **J2C authentication data**.

   b. Click **New** to create a new J2C authentication data entry

   c. Fill in the details as follows:

   Alias a suitable (short) name, such as ″UDDIAlias″

   Userid

   > the database userid (such as db2admin for DB2 or IBMUDDI for Oracle), which is used to read and write to the UDDI registry database. For network Cloudscape the userid can be any value.

   > If you are using a remote DB2 database on the z/OS operating system, the userid must be one that is valid on the remote system.

   Password

   > the password associated with the userid specified above. For network Cloudscape the password can be any value.

   Description

   > a suitable description to describe the chosen userid.

   Click **Apply** and 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
Oracle	Oracle JDBC Driver	Connection Pool data source
Embedded Cloudscape	Derby JDBC Driver	Connection Pool data source
Network Cloudscape	Derby Network Server JDBC Driver provider	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. Create the data source for the UDDI registry by following these steps:

   a. Click **Resources** → **JDBC** → **JDBC Providers**.

   b. Select the desired ′scope′ of the JDBC provider you selected or created earlier. For example, select:

   `Server: yourservername`

   to show the JDBC providers at the server level.

   c. Select the JDBC provider created earlier.

   d. Under **Additional Properties**, select **Data sources** (not 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:

   g. Click **Next**.

    h.   On the database specific properties page of the wizard, enter the following data:

    i. Click **Next**, then check the summary and click **Finish**.

    j. Click the data source to display its properties, and add the following information:

    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**. You will see a message similar to ″Test Connection for datasource UDDI Datasource on server server1 at node MyNode was successful″. If you do not see this message investigate the problem with the help of the error message.

Continue with setting up and deploying your UDDI registry node.

Deploying the UDDI registry application:

You must have already created the database and datasource for the UDDI registry.

Use this task as part of "Setting up a default UDDI node" on page 515 or "Setting up a customized UDDI node" on page 524.

Run the uddiDeploy.jacl script as shown below, from the *app_server_root*/bin directory. This script deploys the UDDI registry to a server (or cluster) specified by you.

> Linux

Note: If you are using either the UNIX or Linux operating systems, add the .sh suffix to the wsadmin command.

```
wsadmin [-conntype none] [-profileName profile_name] -f uddiDeploy.jacl
            {node_name server_name | cluster_name}
```

where

- '-profileName *profile_name*' is the deployment manager profile. If you do not specify a profile, the default profile will be used.
- '-conntype none' is optional, and is only needed if the application server is not running.
- *node_name* is the name of the WebSphere node on which the target server runs. Note that the node name is case sensitive.
- *server_name* is the name of the target server on which you wish to deploy the UDDI registry, such as server1. Note the server name entered is case sensitive.
- *cluster_name* is the name of the target cluster into which you wish to deploy the UDDI registry. Note that the cluster name entered is case sensitive.

For example, to deploy UDDI on node 'MyNode' and server 'server1' on a Windows system, (assuming that server1 is already started):

```
wsadmin -f uddiDeploy.jacl MyNode server1
```

To deploy UDDI into cluster 'MyCluster' on a Windows system:

```
wsadmin -f uddiDeploy.jacl MyCluster
```

You can also deploy the UDDI application (the uddi.ear file) using the administrative console, in the normal way. However, some steps that are performed automatically by the uddiDeploy.jacl script do not take place in this scenario. If you use the administrative console to install the UDDI application, you must perform some actions manually, according to the following steps:

1. Install the application.

2. Click **Applications** → **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**.

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.
```

Continue with setting up the UDDI node.

Initializing the UDDI registry node:

Use this topic to initialize a UDDI registry node after set up or migration.

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.

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.

   Important: 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.

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)

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 "Displaying the UDDI registry user interface" on page 509).
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.

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

After you have deployed the UDDI registry application, you might want to change its environment. For example, you might perform initial evaluation of the UDDI registry using a Cloudscape 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 516
   - "Creating a DB2 for z/OS database for the UDDI registry" on page 518
   - "Creating a Cloudscape database for the UDDI registry" on page 519
   - "Creating an Oracle database for the UDDI registry" on page 520

   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** → **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 516
      - "Creating a DB2 for z/OS database for the UDDI registry" on page 518
      - "Creating a Cloudscape database for the UDDI registry" on page 519
      - "Creating an Oracle database for the UDDI registry" on page 520
   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 521.
   f. Restart the UDDI registry application.
   g. Check that you can access your UDDI data, then delete the old database.

Web Services Invocation Framework (WSIF): Enabling Web services

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)).

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 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.

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:
- Learning about WSIF.
- Using WSIF to invoke Web services.
- WSIF system management and administration.
- 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)).

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" on page 536.
  1. "WSIF - Web services are more than just SOAP services" on page 536.
  2. "WSIF - Tying client code to a particular protocol implementation is restricting" on page 536.
  3. "WSIF - Incorporating new bindings into client code is hard" on page 536.
  4. "WSIF - Multiple bindings can be used in flexible ways" on page 537.
  5. "WSIF - Enabling a freer Web services environment promotes intermediaries" on page 537.
- "WSIF: Overview" on page 537.
  1. "WSIF architecture" on page 538.
  2. "WSIF and Web services that offer multiple bindings" on page 538.
  3. "WSIF and WSDL" on page 539.
  4. "WSIF usage scenarios" on page 539.
  5. "Dynamic invocation" on page 540.

For more information about working with WSIF, visit the Web sites listed in Web services: Resources for Learning.

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.
- Tying client code to a particular protocol implementation is restricting.
- Incorporating new bindings into client code is hard.
- Multiple bindings can be used in flexible ways.
- A freer Web services environment enables intermediaries.

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.

WSIF - 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 2 platform, Enterprise Edition (J2EE) 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.

WSIF - 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.

WSIF - 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.

WSIF - 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.

WSIF - Enabling a freer Web services environment promotes 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 "The goals of WSIF".

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 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 and Web services that offer multiple bindings
- WSIF usage scenarios
- Dynamic invocation

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.
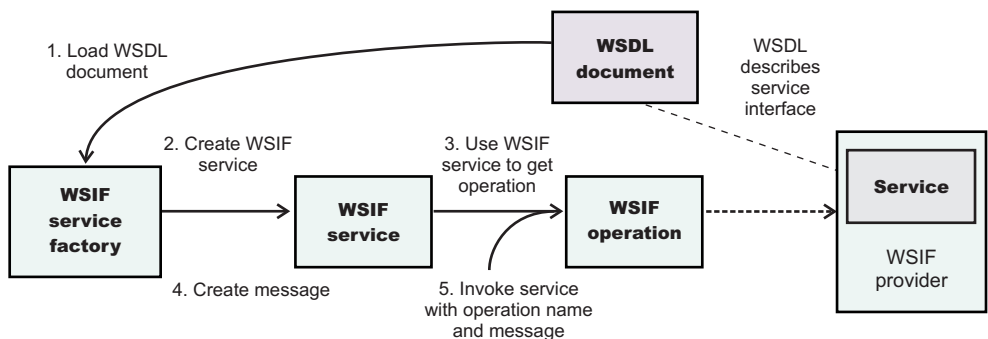
The components of this architecture include:

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 Finding a port factory or service

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 WSIF API reference: Using ports.

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 Web services that offer multiple bindings:

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.

For more information about how to configure a client to dynamically select between multiple bindings, see Developing a WSIF 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 implicit and 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.

For more information about WSDL, see Web services: Resources for learning.

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.

Dynamic invocation:

The Web Services Invocation Framework (WSIF) can provide runtime support for Web services, and for WSDL extensions and bindings, that were not known at build time.

WSIF supports WSDL extensions and bindings that were not known at build time through the use of providers. The providers support Web services that were not known at build time by using the WSDL description to access the target service.

Using WSIF to invoke Web services

You invoke a Web service dynamically by using the WSIF API directly.

You only specify the location of the WSDL file for the service, the name of the operation to invoke, and any operation arguments. All the 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.
- Using complex types.
- Using WSIF to bind a JNDI reference to a Web service .
- Passing SOAP messages with attachments using WSIF.
- Interacting with the J2EE container in WebSphere Application Server.
- Running WSIF as a client.

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.

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 2 platform, Enterprise Edition (J2EE) programming model to utilize J2EE 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:
- Linking a WSIF service to a SOAP over HTTP service.
- Linking a WSIF service to a JMS-provided service (SOAP over JMS, and native JMS).
- Linking a WSIF service to a local Java application.
- Linking 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.

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 J2EE 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 J2EE.

If you have a Web service that you expect multiple clients to use connecting 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.

For an example of the sort of code changes that need to be made in the WSDL file for a SOAP provider, see the following topics:
- The SOAP over JMS provider - writing the WSDL extension.
- SOAP messages with attachments - Writing the WSDL extensions.

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.

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 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, or you can change the WSIF default provider back to Apache SOAP as described in this topic.

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_6.1.0.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_6.1.0.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_6.1.0.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.

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.

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 JMS.

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.

See also the following topics:
• Linking a WSIF service to a SOAP over JMS service
• Linking a WSIF service to a service provided at a JMS destination
• The JMS providers - Configuring the client and server

Linking a WSIF service to a SOAP over JMS service:

If a SOAP message contains only XML, it can be carried on the Java Message Service (JMS) transport mechanism. Here are links to a set of topics that explain how a WSIF service can access a SOAP service that uses the JMS transport.

For information about working with the Java Message Service (JMS) API, see Linking a WSIF service to a JMS-provided service.

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 detailed implementation information, see the following topics:
* The SOAP over JMS provider - writing the WSDL extension.
* The JMS providers - Configuring the client and server.

The SOAP over JMS provider - Writing the WSDL extension:

Here is detailed information, and associated code fragments, to help you to write the WSDL extension that enables your WSIF service to access a SOAP service that use the Java Message Service (JMS) as its transport mechanism.

If a SOAP message contains only XML, then it can be carried on the JMS transport with the JMS message body type **TextMessage**.

The WSDL binding extension for SOAP over JMS varies only slightly from 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.

Here is an 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

Property name	Property description	Corresponding JMS address value
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>
```

Linking a WSIF service to a service provided at a JMS destination:

Using the native JMS provider, WSIF clients can treat a service that is available at a JMS destination as a Web service.

For information about working with the Java Message Service (JMS) API, see Linking a WSIF service to a JMS-provided service.

For detailed implementation information, see the following topics:
- The native JMS provider - Writing the WSDL extension
- The JMS providers - Configuring the client and server

The native JMS provider - Writing the WSDL extensions:

Here is detailed information, and associated code fragments, to help you to write the WSDL extensions that enable your WSIF service to access an underlying service at a Java Message Service (JMS) destination.

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>
```

The JMS providers - Configuring the client and server:

Here is a description of the ways in which the Web Services Invocation Framework (WSIF) interacts with the Java Message Service (JMS), and of the steps you need to take to enable a service to be invoked through JMS by a WSIF client application.

This topic assumes that you installed 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, install one now as described in Installing and configuring a JMS provider.

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 Tips for troubleshooting the Web Services Invocation Framework.

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.

    This task is 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.

Linking a WSIF service to a local Java application:

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.

The Web Services Invocation Framework (WSIF) Java provider is not intended for use in a Java 2 platform, Enterprise Edition (J2EE) 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.

For examples of the code changes that need to be made in the WSDL file, see The Java provider - Writing the WSDL extension.

The Java provider - Writing the WSDL extension:

The Java provider supports the invocation of a method on a local Java object.

To use the Java provider, you need the following binding specified in the WSDL:

```
<!-- 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>
```

Linking a WSIF service to a service implemented as an enterprise bean:

Using the EJB provider, WSIF clients can invoke enterprise beans.

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 Remote Method Invocation over Internet Inter-ORB Protocol (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.

For examples of the code changes that need to be made in the WSDL file, see The EJB provider - Writing the WSDL.

The EJB provider - Writing the WSDL extension:

The EJB provider supports the invocation of an enterprise bean through Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP).

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.

To use the EJB provider, you need the following binding specified in the WSDL:

```
<!-- 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.

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. Develop 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:
   * Developing the WSIF client - the Address Book Sample 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.
   * WSIF API 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.

Developing the WSIF client - the Address Book Sample:

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;
    }
```

```java
    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:

```java
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
<?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 specify this mapping manually or automatically as described in the following sections:
*   Manual mapping of complex types.
*   Automatic mapping of complex types.

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.

Manual mapping of complex types

The method to use when you create these mappings manually depends on the provider that is used. 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.

Automatic mapping of 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 <tt>QName</tt> to a package name, and converting the local part to a class name. WSIF does this mapping using the WSIFUtils methods <tt>getPackageNameFromNamespaceURI</tt> and <tt>getJavaClassNameFromXMLName</tt>.

Using WSIF to bind a JNDI reference to a Web service

This example task shows you how to use WSIF to bind a reference to a Web service, then look up the reference using JNDI.

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 following example shows how, using the Web Services Invocation Framework (WSIF), you can bind a reference to a Web service then look up the reference using JNDI.

Specifying 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

Binding 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.
    }
[...]
```

Looking 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.
    }
[...]
```

Passing SOAP messages with attachments using WSIF

Use the 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.

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.

The Web Services Invocation Framework (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.

For more information, see the following topics:
* SOAP messages with attachments - Writing the WSDL extensions.
* SOAP messages with attachments - Passing attachments to WSIF.
* SOAP messages with attachments - Working with types and type mappings.

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.

SOAP messages with attachments - Writing the WSDL extensions:

These example code fragments show you how to write the WSDL extensions for SOAP messages with 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.

SOAP messages with attachments - Passing attachments to WSIF:

These example code fragments show you how to use WSIF to pass SOAP messages with attachments.

The following code fragment can invoke the service described by the example WSDL in the topic writing the WSDL extensions:

```
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 messages with attachments - Working with types and type mappings:

By default, attachments are passed into the Web Services Invocation Framework (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.

Interacting with the J2EE container in WebSphere Application Server

A description of how, and to what extent, WSIF interacts with the J2EE container that is provided in WebSphere Application Server.

Interaction with a container is limited to the following aspects:

*   Using 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 WSIF system management and administration. As part of the definition of a service, the administrator might define a "preferred port".
*   Using 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.
*   Using WSIF providers to access Java 2 platform, Enterprise Edition (J2EE) services. For example using the EJB provider to access the Java Naming and Directory Interface (JNDI) and make calls to remote enterprise beans.

- Using 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

The Web Services Invocation Framework (WSIF) runs 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.

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.

This 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.

The WSIF API supports the invocation of services defined in WSDL. 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: Pick a port that supports the port type needed, then 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 WSIF API main interfaces are described in the following topics:
- Creating a message for sending to a port (the WSIFMessage interface).
- WSIF API reference: Finding a port factory or service (the WSIFService interface and the WSIFServiceFactory class).
- WSIF API reference: Using ports (the WSIFPort interface and the WSIFOperation interface).

Note: You must ensure that your application uses only one thread to call WSIF.

For additional technical details of the WSIF API, see the generated API information.

WSIF API reference: Creating a message for sending to a port

For message management (that is, message construction and parsing) the underlying API is modeled on WSDL semantics. There is a simple and direct mapping from the WSDL model to the Web Services Invocation Framework (WSIF) classes.

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. A WSIFMessage class is a container for a set of named parts. The WSIFMessage interface separates the actual representation of the data from the abstract type defined by WSDL. WSDL defines messages as XML schema types. 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.

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.

WSIFMessage classes can be sent between Java Virtual Machines (JVMs).

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.

WSIF API reference: Finding a port factory or service

To find a port you use the WSIFService interface, which is a factory for ports.

The port factory 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 from the user. The Web Services Invocation Framework (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. Using this service you can hide and modify the set of available ports at run time.

Here is the WSIFService interface.

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.

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.

WSIF API reference: Using ports

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 the Web Services Invocation Framework (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.

Here is the WSIFPort interface.

Here is the WSIFOperation interface.

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.

UDDI registry client programming

This topic covers programmatic use of the UDDI APIs. The first subtopics describe some standard aspects of the UDDI APIs:

- 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 explains about the support for digital signing of UDDI entities, and for validation of signatures.
- UDDI registry Application Programming Interface contains a summary of the UDDI Version 3 APIs as defined in the UDDI Version 3 specification.

There are several ways in which you can programmatically access the UDDI APIs. The recommended client API is the UDDI Version 3 Client for Java, 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) for further details.
- EJB Interface for the UDDI registry (Deprecated) 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. 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 for details.

UDDI registry Version 3 Entity Keys
Entity Keys, UDDI v1/2 uuid and UDDI v3 uddi keys

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 generated or assigned when first published in the UDDI registry. Within a particular registry, a key MUST be unique. The UDDI version 3 specification expands the space available for keys; it is not limited to a UUID as in versions 1 and 2. Entity keys can now be any URI (Universal Resource Identifier) that follows the recommended UDDI scheme.

Another difference introduced by the UDDI Version 3 specification is that 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 in 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 (Universal Resource Identifier) and is limited to 255 characters. The following diagram shows the different types of keys within the UDDI key scheme:
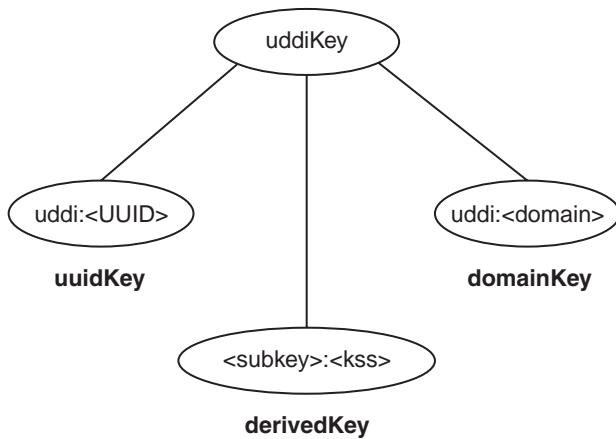
All keys are composed of a set of tokens that are separated by ':'. The first token for all keys that follow the UDDI scheme is "uddi". There are three types of keys:

1. The uuidKeys contain two tokens, the mandatory "uddi" and a <UUID>. These keys assure uniqueness through the UUID algorithm.

2. The domainKeys also contain 2 tokens but its second token is a Domain Name. These keys are intended for creating additional mutually-exclusive key spaces.

3. The derivedkeys 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 is only allowed to save entities using keys from a certain key space if it owns the key generator that represents the key space. This aids in securing unique keys. The key generator is a tModel entity, which key is in the form <subkey>:keyGenerator. By owning this tModel, a publisher can assign keys in the form <subkey>:<kss>. The publisher can also publish new tModel key generators of the form <subkey>:<kss>:keygenerator.

Key uniqueness and registry root key space

Instances of UDDI registry can be configured to be 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 the registry manages. All keys the registry generates are within this key space, and, if allowed by Policy, publishers may request sub-divisions of this key space by publishing new tModel key generators of the form <rootkeygenerator>:<subdivisionIdentifier>:keygenerator and then may include publisher-supplied-keys in subsequent publish requests which are within 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 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 only generates new keys within the key space defined by its own root key generator, and only accepts publisher-supplied-keys which are within a subdivisions of key space owned by the publisher (as the result of a prior 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

As identified above, depending on Policy, (whether the registry supports publisher supplied keys and whether a particular publisher's User entitlements allow the publisher to submit requests for key space) a publisher can submit a request for a (top-level) subdivision of the root registry's key space for its own use.

In addition to 'top-level' subdivisions of the root registry's key space, a publisher can also create further subdivisions of key space for its own use.

A simple example of this is (continuing the example above):

```
uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:keygenerator
```

This request for a further subdivision 'a' is successful when requested by the publisher who previously requested (and hence owns) the tModel for the 'level above' (in this case uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:keygenerator).

Publishing with a 'publisher supplied' key

Having successfully requested a subdivision of a root registry's key space, a publisher must establish and maintain their own scheme for ensuring that the keys generated to be used as publisher-supplied-keys in subsequent publish requests are unique within the subdivision.

Valid schemes need to generate keys which are unique derived keys within the allocated key space subdivision, for example including a unique (incremented) numeric index.

A simple example of this is (continuing the example above):

For key space subdivision resulting from 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 have been 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, the retrieving client is responsible for validating the signature and to provide his own mechanism for ensuring the signer's certificate is signed by a Certificate Authority (CA) that the clients approves and trusts. If a signature is decrypted successfully by using the signer's public key, it is an indication that only the owner of the corresponding private key could have signed and published this element.

Generating a signature

Because an element's attributes are included in the generation of an element's signature, all entity keys must be available at the time that the signature is generated. Publishers are recommended to generate publisher-assigned-keys for all of an element's keys before signing. Alternatively, publishers can publish the element without keys; this 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 the one in the top level element that is returned by a call to getXXDetails(). It is the client's responsibility to perform the validation. The client must have previously imported the publishers X509.3 certificate and validated it based on the CA it trusts. This way the client will have access to the publisher's public validation key that corresponds to the private signing key that the publisher used to sign the entity before publishing it.

The UDDI Version 3 Client can be used to construct JAX-RPC objects and to invoke the UDDI Version 3 WebService. As part of this client a helper class, *com.ibm.uddi.v3.client.apilayer.xmldig.SignatureUtilities*, can be used to create and validate digital signatures on the UDDI Version 3 Entities that support them. See the API documentation page for details of API of this class and its Exception *SignatureUtilitiesException*.

An example of how to use this class can be found at Samples for WebSphere Application Server called UDDIv3ClientSignedBusinessSample.java.

Note: For UDDI, digital signatures are being used to sign the data and **not** to authenticate the SOAP message.

UDDI registry Application Programming Interface

The UDDI Version 3 registry supports multiple versions of UDDI. In addition to UDDI Version 3, it supports UDDI Version 1 and Version 2.

For details of the Version 1 and Version 2 API, visit `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2`.

For details of the UDDI Version 3.0.2 API, visit `http://uddi.org/pubs/uddi_v3.htm`.

The UDDI registry information in this information center defines the support provided by the UDDI registry 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: There is a known restriction within DB2 for zSeries Version 7 that limits the length of publish and inquiry strings to 255 characters. Exceeding this limit will result in an Error 10500 (E_Fatal) error being returned. If you use a character set that uses multiple byte characters, this limit can very easily be exceeded. Care must be taken when using these character sets.

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 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 allows people 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 and then selecting more specific information for drill-down.

The UDDI API specifications accommodate the browse pattern by way of the *find_xx* API calls. These calls form the search capabilities provided by the API and 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 a particular business you know about has any information registered. This sequence would start with a call to *find_business*, perhaps passing the first few characters of a business name that you already know. This returns a *businessList* result. This result is overview information (keys, names and descriptions) derived from the registered businessEntity information, matching on the name fragment that you provided. If you spot the business you are looking for within this list, you can drill down into the corresponding businessService information, looking for particular technical models (for example purchasing, shipping, and so on) using the *find_service* API call. Similarly, if you know the technical *fingerprint* (tModel signature) of a particular software interface and want to see if the business you have chosen 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 managed by a UDDI registry, you can use that key to access the full registered details for 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 of the data items returned by all of the *find_x* return sets is key information. In the case of the business we were interested in, the businessKey value returned within the contents of a businessList structure can be passed as an argument to *get_businessDetail*. The successful return to this message is a *businessDetail* message containing the full registered information for the entity whose key value was passed. This 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 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 should cache this information and use it to contact the service at the registered address whenever the calling application needs to communicate with the service instance. In previously popular remote procedure technologies tools have automated the tasks associated with caching (or hard coding) location information. Problems arise however when a remote service is moved without any knowledge on the part of the callers. Moves occur for a variety of reasons, including server upgrades, disaster recovery, and service acquisition and business name changes.

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 returned is different from the cached information, the service invocation should automatically retry the invocation using the fresh information. If the result of this retry is successful, the new information should replace the cached information.

By using this pattern with Web services, a business using a UDDI registry can automate the recovery of a large number of partners without undue communication and coordination costs. For example, if a business has activated a disaster recovery site, most of the calls from partners 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:

The inquiry API set allows you to locate and obtain details about entries in a UDDI registry. The API is split into a number of functions (see below), each requiring a variety of optional and mandatory arguments.

Use the UDDI Version 3 Client for Java (see UDDI Version 3 Client) to access programmatically all API calls and arguments supported by the UDDI Version 3 registry. You can also access the API functions graphically by using the UDDI user interface, however not all of the functions are available with this method.

The UDDI Version 3 registry supports the following Inquiry API calls:
find_binding
> Locates specific bindings within a registered businessService. Returns a bindingDetail message that contains zero or more bindingTemplate structures matching 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 containing results that match the conditions specified in the arguments.

find_service
> Locates specific services within a registered businessEntity. Returns a serviceList message that matches the conditions specified in the arguments.

find_tModel

Locates a list of tModels that match a set of specified criteria. The response will be a list of abbreviated information about registered tModel data that matches the criteria specified. The result will be 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 matching the businessKey values specified. 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 syntax of the above queries, 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 may contain multiple findQualifier values. Below is a list of the findQualifier short names with a brief description and which find function is applicable.

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:

The UDDI Publish API allows you to publish, delete and update information contained in a UDDI registry. The messages defined in this section all behave synchronously.

Use the UDDI Version 3 Client for Java (see UDDI Version 3 Client) to access programmatically all API calls and arguments supported by the UDDI Version 3 registry. You can also access the API functions graphically by using the UDDI user interface, however not all of the 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 syntax of the above queries, 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:

The UDDI Custody and Ownership Transfer API allows you to transfer custody or ownership of one or more entities contained in a UDDI Version 3 registry.

Use the UDDI Version 3 Client for Java (see UDDI Version 3 Client) to access programmatically all API calls and arguments supported by the UDDI Version 3 registry. You can also access the API functions graphically by using the UDDI user interface, however not all of the functions are available with this method.

Note: The UDDI Version 3 registry supports only intra-node ownership transfer; it does **not** support inter-node custody transfer.

The UDDI Version 3 registry supports the following Custody and Ownership Transfer API calls:

discard_transferToken
Discards a transferToken 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. No actual transfer takes place with the invocation of the API. 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 full details of the syntax of the above queries, 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:

In UDDI Version 1 and Version 2 the security API was part of the Publish API. In UDDI Version 3 the security API is independent.

Use the UDDI Version 3 Client for Java (see UDDI Version 3 Client) to access programmatically all API calls and arguments supported by the UDDI Version 3 registry. You can also access the API functions graphically by using the UDDI user interface, however not all of the functions are available with this method.

The UDDI Version 3 registry supports the following Security API calls:

discard_authToken
> Used to inform a node that a previously obtained authentication token is no longer required and should be considered invalid if used after this message is received. The token is to be discarded and the session is effectively ended.

get_authToken
> Used to request an authentication token in the form of an authInfo element from a UDDI node.

For full details of the syntax of the above queries, refer to the UDDI Version 3 API specification at `http://www.uddi.org/pubs/uddi_v3.htm`.

UDDI Version 3 Client

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.

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 them.

HTTP GET Services for UDDI registry data structures

The UDDI registry offers an HTTP GET service for access to the XML representations of the UDDI data structures businessEntity, businessService, bindingTemplate and tModel. The URL at which these are accessible uses the entity key as a URL parameter. The XML element returned will be a businessDetail, serviceDetail, bindingDetail or tModelDetail, according to the type of entity key supplied. XML for both UDDI version 2 and 3 can be retrieved, at different URLs.

The formats of the URLs to send the HTTP GET requests to are as follows:

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″, then the uddi-org:types tModel can be accessed at the following URLs:

UDDI v2:
> http://myserver.com:9080/uddisoap/get?tModelKey=uuid:c1acf26d-9672-4404-9d70-39b756e62ab4

UDDI v3:
> http://myserver.com:9080/uddiv3soap/get?tModelKey=uddi:uddi.org:categorization:types

There are a number of UDDI property and policy settings that relate to the HTTP GET services:
- Version 3 HTTP GET for UDDI entities
  - Node supports HTTP GET
  - URL Prefix for V3 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 and, depending on WebSphere Application Server security settings and UDDI SOAP service user data constraint transport guarantee settings, supports different end points for different services.

The default context root and URL values listed in this topic apply when you have enabled WebSphere Application Server security and have not changed the default supplied security settings. If you use a non

default security set up, the context root and URL values may differ (see Configuring UDDI registry security for more information about the various security settings).

In the URLs listed below, 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, then services with default end point URLs with HTTPS and SSL port require their data to be transported confidentially. Requests that are not using HTTPS will be 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, then 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 within 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 within 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, providing you still send requests compliant to the SOAP specification. Valid UDDI requests should conform to the UDDI schema, and be as detailed within 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)

Note: Note that 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 within the com.ibm.uddi_1.0.0.jar file. This file contains classes which 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 will require a WebSphere Application Server authenticated user ID, a client program using UDDI4J should 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)

The UDDI EJB interface is deprecated in WebSphere Application Server Version 6.0 and later versions, and supports UDDI version 2 API requests only.

This section describes how to use the EJB application programming interface (API) of the UDDI registry component to publish, find and delete UDDI entries.

The client classes that are required for the EJB interface are contained in *app_server_root*/UDDIReg/ clients/uddiejbclient.jar. You can read the Javadoc for these classes at the Javadoc welcome page.

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 correspond to UDDI Version 2 Inquiry API functions, and all the public methods on the PublishBean 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.

Within 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 10 variations of findBusiness methods, with different variations for finding by name, finding by categoryBag and so on.

The arguments for the EJB interface methods are Java objects in the package com.ibm.uddi.datatypes. Roughly speaking, there is a one to one correspondence between classes in this package and elements of the UDDI Version 2 XML schema. Exceptions to this are, for example, where UDDI XML elements can be represented by a single String. See the Javadoc for package com.ibm.uddi.datatypes for more information at Javadoc welcome page

The methods on the EJB InquiryBean map to the EJB Inquiry Role, and those of the EJB PublishBean 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 will require 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 which logs in to the default realm using the user ID and password. Use the `sas.client.props` configuration file to determine how the user ID and password should be specified (see Configuring security with scripting for information on how to do this).

Using the EJB Client

In this section it is assumed that you have installed both WebSphere Application Server and the UDDI registry (and 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 for communicating with WebSphere Application Server:

   ▶ Linux ◀   `. app_server_root/bin/setupCmdLine.sh` (note that there is a space between the '.' and *app_server_root*)

   ▶ Windows ◀   `app_server_root/bin/setupCmdLine.bat`

2. Ensure that your CLASSPATH includes the uddiejbclient.jar (from *app_server_root*/UDDIReg/clients) and the code for your client.

3. Compile your EJB client programs:

   ▶ Linux ◀   `$JAVA_HOME/bin/javac -extdirs $WAS_EXT_DIRS:$JAVA_HOME/jre/lib/ext -classpath $WAS_CLASSPATH:$CLASSPATH yourcode.java`

   ▶ Windows ◀   `$JAVA_HOME/bin/javac -extdirs $WAS_EXT_DIRS:$JAVA_HOME/jre/lib/ext -classpath $WAS_CLASSPATH:$CLASSPATH yourcode.java`

4. Execute the compiled programs:

   ▶ Linux ◀   `$JAVA_HOME/bin/java -Djava.ext.dirs=$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>`

   ▶ Windows ◀   `%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

---

Service integration

Learning about file stores

Use the subtopics to learn about various aspects of file stores, such as different types of files within it and high availability.

- "File stores"
- "File store high availability considerations" on page 593
- Files in a file store
- "File store configuration attributes"
- File store performance

File stores

File stores enable messaging engines to preserve operating information and to persist those objects that messaging engines need for recovery in the event of a failure, using a file system.

A file store is a type of message store which directly uses files in a file system via the operating system. File store mechanisms splits data storage into three levels: the log file, permanent store files and temporary store files. For more information on these files see "File store configuration attributes."
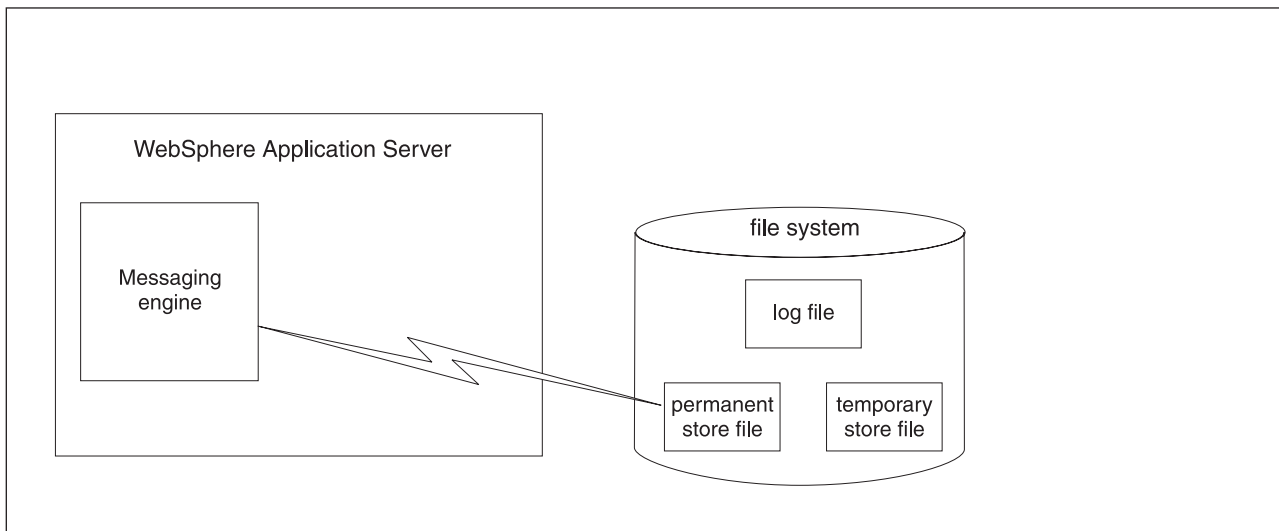
Figure 6. The relationship between a messaging engine and its file store.

File store configuration attributes:

The log file, permanent store file and temporary store file make up a file store. You preserve appropriate amount of space within these three files so that operations and transactions behave predictably.

Data is first written to the log file sequentially, that is, new records are added to the end of the file. When the end of the log file is reached, old records at the beginning of the log file are overwritten by new records and this process repeats. Subsequently data is written to the permanent store file and temporary store file, although extremely short-lived data is only written to the log file.

The permanent store file and temporary store files have a minimum reserved sizes and a maximum size each. When created, the permanent and temporary store files consume their minimum reserved sizes, plus the size of the log. If the maximum size is larger, they grow up to the maximum size as required. The maximum size can be unlimited. It is recommended for production use that the minimum and maximum sizes are the same, as it prevents the store files from growing and shrinking at runtime so that the messaging engine does not gradually fill a file system. Another advantage is that if the file system does fill up while the messaging engine is operating, it is not affected.

The default configuration is intended to be sufficient to be used in typical messaging workloads without any administration. To improve the performance or availability of the log or the two store files, the administrator can modify the file store attributes to control where these files are placed. Similarly, the

administrator can modify the attributes which control the sizes of the log and two store files to handle workloads with a large number of active transactions, large messages or a large volume of message data resident in the messaging engine

Note: This behavior cannot be guaranteed on a compressing file system, for example, NT file system with the **Compress this directory** option selected. You should avoid configuring file store to use a compressing file system for production use.

Table 20. File stores have the following attributes and values

Name	Description	Minimum and Default Values in mega bytes
Log size	Size of the log file, in mega bytes	• *Minimum*: 10 MB • *Default*: 100 MB
Minimum permanent store size	The minimum number of mega bytes reserved by the permanent store file. **Note:** The store files must always be at least as big as the log file.	• *Minimum*: 0 • *Default*: 200 MB
Maximum permanent store size	The maximum size in mega bytes of the permanent store file. **Note:** The store files must always be at least as big as the log file.	• *Minimum*: 50 MB • *Default*: 500 MB
Minimum temporary store size	The minimum number of mega bytes reserved by the temporary store file. **Note:** The store files must always be at least as big as the log file.	• *Minimum*: 0 • *Default*: 200 MB
Maximum temporary store size	The maximum size in mega bytes of the temporary store file. **Note:** The store files must always be at least as big as the log file.	• *Minimum*: 50 MB • *Default*: 500 MB
Unlimited permanent store size	Indicates whether the permanent store file is unlimited in size	• *Default*: false
Unlimited temporary store size	Indicates whether the temporary store file is unlimited in size	• *Default*: false
Log directory	Name of the directory that the log file is in	• *Default*: `${USER_INSTALL_ROOT}/filestores/com.ibm.ws.sib/<me_name>.<me_build>/log`
Permanent store directory	Name of the permanent store file's directory	• *Default*: `${USER_INSTALL_ROOT}/filestores/com.ibm.ws.sib/<me_name>.<me_build>/permanentStore`
Temporary store directory	Name of the temporary store file's directory	• *Default*: `${USER_INSTALL_ROOT}/filestores/com.ibm.ws.sib/<me_name>.<me_build>/temporaryStore`

File store high availability considerations

High availability refers to the capability of failing over messaging engines between servers. File stores can be used in highly available environments.

You can achieve high availability by choosing a file store as the message store of a messaging engine. In order to make a file store highly available, you should use hardware or software facilities to maximize the availability of the file store data, for example, SAN.

Note: It is important to ensure that the directories containing the log file and store files are universally accessible with the same directory name from all members of the cluster.

WebSphere Application Server v6.1 supports two styles of file system access to enable this:

- Cluster-managed file system

  This style of file system access uses high availability clustering and failover of shared disks to ensure that the file store's directories are accessible from the server that is currently running the messaging engine. The file store's directories are located on file systems in the shared disks, and high availability cluster scripts are used to mount the file systems on the node with the server that is running the messaging engine.

- Networked file system

  This style of file system access uses a network file system. The most popular protocols for accessing remote files are Common Internet File System (CIFS) and Network File System (NFS). It is recommended that you adopt Version 4 of NFS, which supports automated failover to ensure access locking. Access locking ensures the integrity of the log files, that is, only a single client process can access the log at a time.

Note: It is important to check that the file system configuration is correct, because it cannot be checked by the WebSphere configuration system or messaging engine. Errors only surface at runtime, so thorough failover testing is recommended.

Further information:
These considerations for enabling access to the file store's directories are similar to those for enabling access to the recovery log in a cluster. For more information see the following article: Transactional high availability and deployment considerations in WebSphere Application Server V6

Exclusive access to file store
A messaging engine has exclusive access to file store by design of file stores.

Each file store contains information which uniquely identifies the messaging engine which created it. A file store can only be used by the messaging engine which created it.

The messaging engine opens the file store's files in exclusive mode to prevent multiple instances of the same messaging engine from simultaneously using the file store, for example, by accidental activation on the messaging engine of multiple servers in a cluster. When a messaging engine stops, either in a controlled fashion or as a result of server failure, the file store's files are closed. Then another instance of the same messaging engine is able to open the file store.

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.

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.

Learning about programming mediations

This topic describes additional mediation concepts that you will need for the mediation programming task.

Using the capabilities of the mediation infrastructure, you can program your own mediations to customize the way the service integration bus handles messages. (For further information about mediations, see Mediations and Mediation handlers.

For example, your mediations can
- reformat messages from the format produced by one application to the format required by another
- route messages based on message content
- distribute messages to more than one destination
- augment messages by adding information to a message from another data source
- transcode messages from one concrete representation to another

There are some topics that you should understand before you start to program your own mediations.
1. Read "Overview of programming process" for an overview of how you can program mediations.
2. Read "SI programming resources" on page 597 for an overview of the programming resources available to you.
3. Read "SDO data graphs" on page 597 for information about SDO data graphs, the abstract representation of data that gives you a consistent interface to different message formats.
4. Read "Coding considerations for mediations" on page 598 for things to consider when writing mediation code. These are hints for successful mediation programming, rather than instructions on the task of programming a mediation, which you can read about in "Programming mediations" on page 599

Overview of programming process

This topic describes developing mediations, from coding through to deployment

Programming mediations involves two types of task that draw on different technical skills: programming and integration. Typically, Java programmers will design and code mediations as components called handlers. Integration involves aggregating the mediation handlers using handler lists, then assembling them into a deployable unit to deploy and install them.

Programming. The basic programming task is to create a mediation handler. This is a wrapper for the mediation code that operates on the message. You can create the handler code using information in "Writing a mediation handler" on page 600. Then you will need to add the function mediation code, see

"Adding mediation function to handler code" on page 601. There are three different APIs you will use to work with messages: the SIMessage and SIMessageContext APIs allow you to manipulate the contents of the message, and the SIMediationSession API gives access to the service integration bus so you can send and retrieve messages. See "SI programming resources" for an overview of the APIs, and see "SIMessageContext" on page 610 for reference information and access to the generated API information.

Integration. Mediations can be "pipelined" to create a more powerful set of operations to be performed on a message. At runtime, each handler in the list is invoked in sequence. Each time a handler returns a value of *True*, the same message context is passed to the next handler. If a handler returns *false*, then the context is not passed to any more handlers, which will result in the message being discarded and it will not be delivered to the destination. You use the Application Server Toolkit (AST) deployment wizard to create handler lists (which may be simply a list of one handler) before deploying the handler list as an Enterprise Archive (EAR file).

SI programming resources

This topic describes the programming APIs that are available for working with messages when you program a mediation.

A mediation handler must implement the MediationHandler interface. This interface defines the method which will be invoked by the mediation runtime. For further information on the MediationHandler API, see "MediationHandler" on page 610

The SIMessage and SIMessageContext APIs allow your mediation to operate on the contents of the message. The SIMediationSession API gives your mediation access to the SI Bus so that the mediation can send and receive messages.

- For further information on the SIMessageContext API, see "SIMessageContext" on page 610
- For further information on the SIMessage API, see "SIMessage" on page 611
- For further information on the SIMediationSession API, see "SIMediationSession" on page 612

SDO data graphs

This topic describes how SDO data graphs are used to represent different types of message information in a standard way giving a simple and powerful model for programming mediations.

SDO data graphs are an important concept for mediation programmers. Service Data Objects (SDO) is a technology designed to simplify and unify the way in which applications handle data. Using SDO, you can uniformly access and manipulate data from diverse data sources, including relational databases, XML data sources, Web services, and enterprise information systems.

For a good introduction to SDO, refer to Introduction to Service Data Objects.

SDO is based on data graphs, which are structured collections of data objects. In general, graphs generated from messages will have a tree structure. A mediation retrieves a data graph from a message, transforms the data graph, and the updates to the graph are reflected in the message.

In WebSphere Application Server, data access services connect mediations to data sources, allowing mediations to manipulate an abstract representation of the message, the SIMessage. The SIMessage API provides a method, getDataGraph(), that returns the SDO data graph containing the SIMessage content in a tree representation, or graph of data objects, each of which represents one or more fields in the message, or points to other objects.

When a data graph is requested from a message, the appropriate data access service is identified by a format property in the SIMessage. The format string controls which data access service is used to process the message, and may also contain additional control information for that data access service. In turn, the

structure of the message is controlled by the data access service. For more information about the data access services available in WebSphere Application Server, see "SDO data graph information" on page 612

You use the SIMessageContext API for access to the SIMessage and its rich set of message manipulation methods, and to the SIMediationSession, for Service Integration technologies functionality. For further information, see "SIMessageContext" on page 610

A data object holds a set of named properties, each of which contains either a primitive-type value or a reference to another data object. The Data Object API provides a dynamic data API for manipulating these properties, with the following interfaces that relate to instance data:

- The **DataObject** interface provides a set of methods to retrieve and update the contents of a data object. It also provides methods to access the container of the data object and the data graph to which the data object belongs, to create a new instance of a contained data object, and to delete a data object from its container. In addition, the DataObject interface provides the ability to get the type of the data object.

- A **DataGraph** is a graph of data objects. The graph consists of a single root data object along with all the data objects that can be reached by recursively traversing the containment references of the root data object.

SDO also contains a metadata API for examining the model of a DataGraph:

- A **Type** has a set of Property objects. SDO Types can be compared with type definitions in other type systems. For example, the SDO view of a Java Class would be a Type, and each field in the Class would be represented by a Property. For XML Schema, a ComplexType would be represented by a Type, with a Property for each element or attribute.

- **Property:** a data object is composed of properties. Each property can be accessed by specifying the Property object, the name of the property, or the index of the property.

Coding considerations for mediations

This topic contains programming hints for successful mediations programming.

- Take care to avoid looping in the Forward Routing Path. For example, if you set a destination in the path that is the same as the current destination, the message will endlessly circle, with the routing path being reset to the current destination each time. The mediation framework does not check for loops in routing paths.

- Avoid the use of static fields where possible. A single mediation may be deployed to process multiple messages concurrently.

- Do not cache values computed from the message context or message contents. Such values may change from message to message. The exception is caching values derived solely from the mediation handler properties for performance purposes.

- Mediation programming is subject to the same restrictions as programming an EJB. For more information about restrictions, see Section 18.1.2 of the EJB 1.1 specification.

- Choose the appropriate level of transactional control for your mediation: for example, a mediation that operates on fields within a message is unlikely to have implications for transactional control. At the other extreme, if your mediation updates database fields, it requires transactional control, and you should alert your administrator to set the UseGlobalTransaction flag in the mediation definition. This flag defaults to a value of *False*.

- Considerations that apply specifically to message format:

  – It is good practice to check that your message conforms to the expected format after your mediation function has operated on it. You should use the isWellFormed method in the SIMessage interface to check that all the values of the message properties can be serialized, and that the data graph of the message conforms to the format of the message.

  – Depending on how you want to process the message, you can specify a format that meets your needs rather than accept the natural format. For example, if you want to handle a SOAP message

simply as a byte string, use the getNewDataGraph method in the SIMessage interface and specify a format of `JMS/bytes`. getNewDataGraph returns a new SDO data graph containing a copy of the SIMessage payload content in the tree representation specified by the format field, in this example as a byte string.

    – It is good practice to check the message format in the mediation code because a mediation is unlikely to successfully process a message with an unexpected format. Use getFormat method on the SIMessage interface.

- Due to a restriction in the SDO user interface to the message, message access methods do not have a 'throws' clause. As a result, an exception thrown by an access method because of a parsing error is an unchecked exception. Your mediation can catch a parsing exception by checking for the exception class SIMessageParseException in the `com.ibm.websphere.sib.exception` package. Use code similar to the following example:

```
try {
  // Function involving SDO message access
} catch (SIMessageParseException e) {
  // Look at the real cause of the runtime exception, and act on it.
  // It is likely to indicate a parse failure...
  Throwable cause = e.getCause();
}
```

    Note: If a mediation does not catch the SIMessageParseException, the original version of the message is sent to the exception destination.

- When deploying your mediation, give the handler and the handler list memorable and descriptive names.
- Where you deploy a single mediation against a single destination, use exactly the same name for your mediation handler, the mediation handler list and the mediation object in the administrative console.
- For performance reasons, specify selector rules so that the mediation mediates required subsets only of the messages passing through a destination.

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.

Code examples for writing a mediation are provided at "Adding mediation function to handler code" on page 601.

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 the Application Server Toolkit (AST).

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 runtime.

Take the following steps to program a mediation:

1. Create a mediation handler. For more information, see "Writing a mediation handler."
2. Add mediation function code to your mediation handler. For more information, see "Adding mediation function to handler code" on page 601.
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 607.
4. Use the Application Server Toolkit (AST) deployment wizard 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 AST 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.

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, take 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.

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 start this task, you should have access to a Java programming environment, and the Eclipse development environment (part of the Application Server Toolkit, or AST, supplied with WebSphere Application Server.)

A mediation handler can be deployed. Each mediation handler executes some specific message processing at runtime, 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.

Your mediation handler class can be defined either in a Java project or an EJB project (which is needed for the deployment artefact.) Your programming and deployment artefacts 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 classpath correctly.

1. Create a new EJB project:

a. Switch to the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other >J2EE**.

b. From the File menu, select **New > Project**.

c. Expand the J2EE 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 6.0, as in the next step.

e. Enter a name for the project and target the project to WebSphere Application Server Version 6.0. (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 Application Server Toolkit (AST). 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. Prepare your mediation handler for installation into the application server. Follow the instructions in the topic entitled ″Deploying a mediation handler″ in the AST information center. Note that AST is also available in the Rational Software Development Platform.

Next, you are ready to install your mediation handler into the application server.

Adding mediation function to handler code

This topic describes how to add mediation function to a preexisting mediation handler.

You should have created the basic mediation handler in an EJB project (see "Writing a mediation handler" on page 600

There are four ways in which you can work with a mediation to add to, or change, its function. You can work with the mediation context properties, with the message properties, with the contents of the message (the message payload), or with the message header, for instance to change the routing of the message.

1. To work with the message context, see "Working with the message context"

2. To work with the message properties, see "Working with the message properties" on page 602

3. To work with the message header, see "Working with the message header" on page 603

4. To work with the message payload, see "Working with the message payload" on page 607

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 start this task, you should read about how information is carried in the mediation context in Mediation context information

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:

```
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 start this task, you should read about the properties that are supported by the SIMessage interface in Message properties support for mediations.

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 611.) 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");`

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 start this task, you should have created the basic mediation handler in an EJB project (see "Writing a mediation handler" on page 600. It will be useful to have understood the elements of the task "Working with the message payload" on page 607, because some of those elements are used in 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 606.

Setting routing addresses in a message header:

This topic describes how to add function to a preexisting mediation handler to set routing addresses in the message header.

Before you start this task, you should have created the basic mediation handler in an EJB project (see "Writing a mediation handler" on page 600.

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 605 and "SIDestinationAddressFactory" on page 605.

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. For example, `....` `SIMediationSession mediationSession = mediationContext.getSession();`

   b. Create a forward routing path to set on the cloned object. Use, for example the Vector class to create a extendable array of objects.

   c. Get the SIDestinationAddressFactory which is to be used for creating SIDestinationAddress instances. For instance, `SIDestinationAddressFactory destFactory = SIDestinationAddressFactory.getInstance();`

   d. Create a new SIDestinationAddress, representing an SIBus Destination. For instance, `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 instance, 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.

The complete mediation function code to changer the forward routing path may look similar to 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 clone's frp
   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.

In addition to the routing fields (see "Setting routing addresses in a message header" on page 604), 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 reference 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 reference 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 reference 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	• getPriority • setPriority • getReplyPriority • setReplyPriority
Reliability (ReplyReliability)	Specifies the reliability of message delivery. See Message reliability levels for a description of the possible values.	JMSDeliveryMode supports two levels of reliability: PERSISTENT and NON_PERSISTENT	• getReliability • setReliability • getReplyReliebility • setReplyReliability
TimeToLive (ReplyTimeToLive, RemainingTimeToLive)	Specifies the time in milliseconds a message can remain on the queue before it expires	JMSExpiration is the time of expiry calculated as current time plus time-to-live.	• getTimeToLive • getReplyTimeToLive • getRemainingTimeToLive • setTimeToLive • setReplyTimeToLive • setRemainingTimeToLive
Discriminator (ReplyDiscriminator)	String containing a topic that is tested by a selector rule to determine if message should be mediated.	No corresponding JMS field	• getDiscriminator • setDiscriminator • getReplyDiscriminator • setReplyDiscriminator
RedeliveredCount	Read-only field containing the count of each time a message has been redelivered	JMSRedelivered is an indicator it is likely, but not guaranteed, that this message was delivered but unacknowledged in the past.	getRedeliveredCount
ApiMessageId	A value that uniquely identifies each message sent.	JMSMessageId	• getApiMessageId • setApiMessageId
CorrelationId	Used to link one message with another - typically a response message with its request.	JMSCorrelationId	• getCorrelationId • setCorrelationId
UserId	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.

This task requires a mediation handler in an EJB project. For more information, see "Writing a mediation handler" on page 600. You should also read the tips for successfully programming mediations in the topic "Coding considerations for mediations" on page 598.

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" on page 597
- "MediationHandler" on page 610
- "SIMessageContext" on page 610

To work with specific fields within a message, use SDO data graphs. For more information, see "SDO data graphs" on page 597. For more information about the format of supported message types, and examples of how to work with them, see "SDO data graph information" on page 612.

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 (see "SDO data graphs" on page 597) from the message. For example, `DataGraph dataGraph = message.getDataGraph();`

3. **Optional:** Locate data objects within the payload as follows:

   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 instance, 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" on page 609.

5. **Optional:** Transcode the payload into another format as follows:

   a. Review the topic "Transcoding between different message formats" on page 629 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 instance, `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 as follows:

   a. Review the topics "Data graph to bytes conversion" on page 628 and "Bytes to data graph conversion" on page 629 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. For an example of how to work with a message as a stream of bytes, 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 that 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 may look similar to 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.

public interface MediationHandler. This interface defines the method which will be invoked by the mediation runtime.

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, 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 2 Platform, Enterprise Edition (J2EE) **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. Read more about the SIMediationSession API in "SIMediationSession" on page 612.

For more information about SIMessageContext, see the SIMessageContext generated API information.

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" on page 597.
- 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 different message formats" on page 629.
- 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 "Data graph to bytes conversion" on page 628 and "Bytes to data graph conversion" on page 629.
- 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 602.
- 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 604.
- 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, seeSIMessage.

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 .

SDO data graph information:

This topic describes working with SDO data graphs to access message data.

A message published in one format (for instance, a Web services SOAP message) may be routed to a consumer that requires another format, such as enterprise beans, using the Java API for XML-based RPC (JAX-RPC). Equally, the routing could be in the other direction. If the message is operated on by a mediation as it passes through the bus, in either direction, the mediation must be able to operate on the message regardless of the underlying format.

This is achieved by using a common message model for the data mediators. The model is called SDO DataGraph (see "SDO data graphs" on page 597) and it gives an abstract view of the message, allowing

you to concentrate on the information being conveyed (such as the parameters of the request, the data of the response) without having to worry about the packaging of that information.

you can read the WebSphere Application Server information on data access with SDO in SDO view. The topic also contains a link to an introductory white paper, http://www-106.ibm.com/developerworks/java/library/j-sdo/.

For further information about the data graph format of:
- Web services messages, see "Web services overview"
- JMS messages, see "JMS formats" on page 625

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 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" on page 614 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 617.
- 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 621.

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" on page 615
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" on page 615
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" on page 615
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 615
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 616
body	`commonj.sdo.DataObject`	A nested data object, which represents the body of the SOAP Envelope. See "Message body layout" on page 617

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" on page 616.

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's 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 621 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 621 for more information.

Mapping XML schema definitions to the SDO type system:

This topic contains 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 runtime.

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

Notes:

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 614 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 614.) 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 614) 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 615. 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 616. 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 617. 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" on page 626
JMS Stream message	JMS:stream	See "JMS formats -- Stream" on page 626
JMS Object message	JMS:object	See "JMS Formats -- object" on page 626

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();
  }
}
```

Message conversions:

This topic describes the information for converting or transcribing messages.

You can convert a message payload into a byte array, for example for the purpose of logging a message, and you can reconstruct the message from the byte array. Additionally, you can re-express the message payload in a different format using the transcoding operation. For more information, refer to the following topics:

- "XML Schema definition for stream messages." This describes the XML schema for stream messages, and describes some of the conversions between byte streams and message types.
- "Data graph to bytes conversion" on page 628. This describes converting messages into byte arrays, and the rules associated with each message format.
- "Bytes to data graph conversion" on page 629. This describes converting byte arrays into messages, and the rules associated with each message format.
- "Transcoding between different message formats" on page 629. This describes transcoding different message formats and the rules associated with each format pairing.

Note that JMS:map is not supported.

XML Schema definition for stream messages

The conversions defined in the following two tables use an XML Schema definition with target namespace http://www.ibm.com/xmlns/prod/websphere/messaging/jms/ for expressing JMS stream messages in XML.

```
<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>
```

Data graph to bytes conversion

You can convert a message payload into a byte array, for example for the purpose of logging a message. The following table summarizes the rules associated with each message format when converting a message payload into a byte array:

Table 21. Conversion of SIMessage data graph to bytes.

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.
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.

Table 21. Conversion of SIMessage data graph to bytes. (continued)

Datagraph format	Pre-conditions	Outcome	Character set encoding
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

Bytes to data graph conversion

You can reconstruct the message payload from a byte array, for example after a mediation has logged a message. The following table summarizes the rules associated with converting the byte array into the message payload:

Table 22. Conversion of bytes to 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.

Transcoding between different message formats

You can re-express the message payload in a different format using the transcoding operation. The following table summarizes the message formats that you can transcode, and the rules associated with each format pairing. Each cell in the table shows the result of attempting to transcode from the format given in the row title to the format given by the column title.

Note: The abbreviation DG is used for data graph.

Table 23. Allowed and disallowed transcodings between message formats

	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.

Notes:
1. A message with format JMS: does not have a payload so the DataGraph returned is always null. It is not valid to have a JMS:text data graph with the data set to null. A mediation that has a JMS: message and requires a JMS:text data graph should call createDataGraph() rather than attempt to turn a null value into another value.
2. Since a JMS: message does not have a payload, all operations which specify a to format of JMS: will return a null value for the new data graph.
3. You can call getNewDataGraph() with the same format as the original data graph. It returns a copy of the data graph that you can edit, 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 ":".

Programming for interoperation with WebSphere MQ

This topic covers programming considerations for messaging applications that interoperate with WebSphere MQ applications.

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 interoperability with WebSphere MQ."

2. Read about design considerations for programs that interoperate with WebSphere MQ in "Designing for interoperation with WebSphere MQ" on page 634.

Learning about programming for interoperability with WebSphere MQ

This topic describes what you need to know to write programs that will interoperate with a WebSphere MQ network.

The WebSphere Application Server can be connected to other messaging systems based upon WebSphere MQ, including

- WebSphere Application Server Version 5.0 systems that include the MQ-based JMS provider.
- WebSphere MQ queue managers.
- WebSphere Application Server 5.0 J2EE application clients.

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. Read more about how JNDI simplifies the programming task in "Using a JNDI namespace to connect to different JMS provider environments."

If your application has to interoperate with queue managers on WebSphere MQ systems, there are a few significant differences that programmers must account for in their messaging applications.

Application messages received from another WebSphere MQ-based messaging system are converted into JMS messages. When messages are sent to WebSphere MQ, the conversion is performed in the opposite direction. A configuration setting on the destination definitions determines whether JMS messages are forwarded to WebSphere MQ as MQ JMS messages (which include an MQRFH2 header) or as non-JMS MQ messages.

You can read more about how the two sets of formats are mapped to each other in "Mapping of messages flowing through the WebSphere MQ link" on page 636. You can read more 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 635.

There are three main differences between the WebSphere Application Server service integration and WebSphere MQ messages. You will have to take these differences into account when you design your WebSphere messaging application (for more information about designing, see "Designing for interoperation with WebSphere MQ" on page 634). The differences are:

- "Addressing destinations across the WebSphere MQ link" on page 632.
- "Reply-to queues for use with WebSphere MQ" on page 633.
- "Reply-to topics for use with WebSphere MQ" on page 633.

For information about ways of interoperating with WebSphere MQ, including using a WebSphere MQ server, or using WebSphere MQ as an external JMS provider, see Learning about WebSphere MQ server.

Using a JNDI namespace to connect 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 it's a WebSphere Application Server 6.0 default messaging ConnectionFactory, then a connection will be made to the same service integration bus.

Addressing 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 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.

Reply-to queues for use with WebSphere MQ:

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 "Addressing destinations across the WebSphere MQ link" on page 632.

Deciding to use reply-to queues is part of application design (see "Designing for interoperation with WebSphere MQ" on page 634). 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.

Reply-to topics for use with WebSphere MQ:

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 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 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 for interoperation with WebSphere MQ

This topic outlines the steps you should consider when designing an application that will interoperate with queue managers in a WebSphere MQ network.

You should identify the WebSphere MQ queues that your applications will interoperate with. 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 environment and the service integration bus. There are three types of reference material:
   a. Read about mapping unique to service integration bus messaging in "Mapping of additional MQRFH2 header fields in service integration."
   b. Read about mapping between WebSphere Application Server service integration bus messaging and WebSphere MQ in "Mapping between a WebSphere service integration bus and WebSphere MQ" on page 635.
   c. Read about the differences between the WebSphere MQ functions and the service integration bus in "WebSphere MQ functions not supported by service integration" on page 640

2. Design your JMS client based on the typical J2EE pattern:
   a. Use JNDI to find a ConnectionFactory object.
   b. Use JNDI to find one or more Destination objects.
   c. Use the ConnectionFactory to create a JMS Connection.
   d. Use the Connection to create one or more JMS Sessions.
   e. Use a Session and the Destinations to create the MessageProducers and MessageConsumers needed.
   f. Start delivery of messages by starting the 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. See "Addressing destinations across the WebSphere MQ link" on page 632 for more information.

4. Identify any reply destinations that are used by your application and check them for name-handling incompatibilities.

5. If your application publishes messages that you wish 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 "Reply-to topics for use with WebSphere MQ" on page 633 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
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 WebSphere service integration bus and WebSphere MQ:

This topic directs you to different sets of mapping information to help you program applications that interoperate with WebSphere MQ.

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. The WebSphere MQ link maps values between the two, selecting appropriate values 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 "Mapping of messages flowing through the WebSphere MQ link" on page 636.
- Inbound message fields and properties conversion to JMS. See "Mapping of WebSphere MQ message fields and properties to JMS" on page 637.
- Inbound MQRFH2 header fields to JMS files and properties conversion. See "Mapping of MQRFH2 header fields to JMS" on page 638.
- 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 639.
- Data conversion. See "Conversion of data to and from WebSphere MQ" on page 640.

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
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

Mapping of messages flowing through the WebSphere MQ link:

This topic describes how messages sent from WebSphere MQ, through the WebSphere MQ link, are mapped to JMS messages, and messages sent to WebSphere MQ, through the WebSphere MQ link, are mapped to JMS messages or non-JMS messages.

Application messages received from a WebSphere MQ application are converted into WebSphere Application Server JMS messages as shown in the table below.

WebSphere MQ or MA88	WebSphere Application Server service integration
WebSphere MQ JMS text message	WebSphere Application Server JMS text message
WebSphere MQ JMS bytes message	WebSphere Application Server JMS bytes message
WebSphere MQ JMS stream message	WebSphere Application Server JMS stream message
WebSphere MQ JMS map message	WebSphere Application Server JMS map message
WebSphere MQ JMS object message	WebSphere Application Server JMS object message
WebSphere MQ text message	WebSphere Application Server JMS text message
Other WebSphere MQ message format	WebSphere Application Server JMS bytes message
WebSphere MQ broker control message	WebSphere Application Server broker control message
WebSphere MQ broker response message	WebSphere Application Server broker response message

When messages are sent by an application through the WebSphere MQ link to a WebSphere MQ network they are converted as shown in the table below. A destination setting on the WebSphere MQ link determines whether JMS messages are forwarded to WebSphere MQ as WebSphere MQ JMS messages which include an MQRFH2 header, or as non-JMS messages. For more information, see Point-to-point messaging with a WebSphere MQ network.

WebSphere Application Server service integration	WebSphere MQ or MA88 (choice depends on destination setting)
WebSphere Application Server JMS text message	WebSphere MQ JMS text message or WebSphere MQ text message (MQFMT_STRING)
WebSphere Application Server JMS bytes message	WebSphere MQ JMS bytes message or WebSphere MQ message (MQFMT_NONE)
WebSphere Application Server JMS stream message	WebSphere MQ JMS stream message or WebSphere MQ message (MQFMT_NONE)
WebSphere Application Server JMS map message	WebSphere MQ JMS map message or WebSphere MQ message (MQFMT_NONE)
WebSphere Application Server JMS object message	WebSphere MQ JMS object message or WebSphere MQ message (MQFMT_NONE)
WebSphere Application Server broker control message	WebSphere MQ broker control message
WebSphere Application Server broker response message	WebSphere MQ broker response message

Mapping of WebSphere MQ message fields and properties to JMS:

This topic describes how the WebSphere MQ message fields and properties map to JMS and are converted by the WebSphere MQ link.

The JMS API defines the set of fields and properties available on a JMS message. The message originating from WebSphere MQ is translated into a form used by the service integration bus. This content is accessed using the methods defined by the JMS *javax.jms.Message* class and subclasses. For example, the text body of a WebSphere MQ message received by the WebSphere MQ link engine is accessed using the *getText* method of the resulting *javax.jms.TextMessage* object. Additionally, a set of JMS provider-specific properties are present that reflect the message's underlying WebSphere MQ representation. These properties are relevant only to applications designed to be WebSphere MQ-aware.

The tables below show the mapping of MQMD V1 and V2 fields.

Table 24. Mapping between message descriptor field V1 and JMS fields and properties.

WebSphere MQ MQMD V1 field in original message	WebSphere Application Server JMS message field or property	Type
StrucId V1	-	
Version	-	
Report	JMS_IBM_Report_COA JMS_IBM_Report_COD JMS_IBM_Report_Expiration JMS_IBM_Report_Exception JMS_IBM_Report_PAN JMS_IBM_Report_NAN JMS_IBM_Report_Pass_Msg_ID JMS_IBM_Report_Pass_Correl_ID JMS_IBM_Report_Discard_Msg	s s s s s s s s s
MsgType	JMS_IBM_MsgType	s
Expiry	JMSExpiration	h

Table 24. Mapping between message descriptor field V1 and JMS fields and properties. (continued)

WebSphere MQ MQMD V1 field in original message	WebSphere Application Server JMS message field or property	Type
Feedback	JMS_IBM_Feedback	s
Encoding	JMS_IBM_Encoding	s
CodedCharSetId	JMS_IBM_Character_Set	s
Format	JMS_IBM_Format	s
Priority	JMSPriority	h
Persistence	JMSDeliveryMode	h
MsgId	JMSMessageID	h
CorrelId	JMSCorrelationID	h
BackoutCount	JMSXDeliveryCount	p
ReplyToQ	JMSReplyTo	h
ReplyToQMgr	JMSReplyTo	h
UserIdentifier	JMSXUserID	p
AccountingToken	-	
ApplIdentityData	-	
PutApplType	JMS_IBM_PutApplType	s
PutApplName	JMSXAppID	p
PutDate	JMSTimestamp JMS_IBM_PutDate	h s
PutTime	JMSTimestamp JMS_IBM_PutTime	h s
ApplOriginData	-	
Key: h=message header field p=message property s=provider-specific property		

Table 25. Mapping between message descriptor field V2 and JMS fields and properties.

WebSphere MQ MQMD V2 field in original message	WebSphere Application Server JMS message field or property	Type
GroupId	JMSXGroupID	p
MsgSeqNumber	JMSXGroupSeq	p
Offset	-	
MsgFlags	JMS_IBM_Last_Msg_In_Group	s
OriginalLength	-	
Key: p=message property s=provider-specific property		

Mapping of MQRFH2 header fields to JMS:

This topic describes how WebSphere MQ MQRFH2 header fields are mapped to WebSphere Application Server JMS fields and properties.

Where items appear in the MQRFH2 header as well as in the message descriptor, the MQRFH2 value takes precedence if present. See "Mapping of WebSphere MQ message fields and properties to JMS" on page 637.

MQRFH2 folder and field (jms folder)	JMS message field or property	Type
Dst	JMSDestination	h
Dlv	JMSDeliveryMode	h
Exp	JMSExpiration	h
Tms	JMSTimestamp	h
Cid	JMSCorrelationID	h
Rto	JMSReplyTo	h
Gid	JMSXGroupId	p
Seq	JMSXGroupSeq	p
Key: h=message header field p=message property		

Mapping of MQMD Report fields to JMS provider-specific properties:

This topic describes how a WebSphere MQ MQMD Report field can map to a number of JMS properties rather than a single property. The possible values for each are shown in the table below.

Table 26. MQMD flag values mapped to JMS properties.

WebSphere MQ MQMD Report option	JMS field or property name
MQRO_NONE MQRO_COA MQRO_COA_WITH_DATA MQRO_COA_WITH_FULL_DATA	JMS_IBM_Report_COA
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 all dynamic queue names with '_Q' and suffixes them with 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 WebSphere MQ JMS clients, the inbound nonpersistent reliability and the inbound persistent reliability fields of the MQ client link can be set to control the message persistence.

Designing for interoperation with WebSphere MQ using a WebSphere MQ server

This topic outlines the reference information you should consider when designing an application that will interoperate with WebSphere MQ using a WebSphere MQ server.

You should identify the WebSphere MQ queues that your applications will interoperate with.

Familiarize yourself with important reference information for the two interoperating environments, WebSphere MQ environment and the service integration bus. There are four types of reference material:
1. Read about "Mapping of additional MQRFH2 header fields in service integration when using a WebSphere MQ Server"
2. Read about "Mapping the JMS Destination property between service integration and WebSphere MQ when using a WebSphere MQ server" on page 643
3. Read about "Mapping the Message Reliability property between service integration and WebSphere MQ when using a WebSphere MQ server" on page 643
4. Read about "Reply to queue constraints when using a WebSphere MQ server" on page 644

Mapping of additional MQRFH2 header fields in service integration when using a WebSphere MQ Server

This topic describes additional fields that are added to the MQRFH2 header when interoperating with WebSphere MQ using a WebSphere MQ server. These additional fields allow for properties that can be used by service integration, but that are unavailable in WebSphere MQ, to be preserved.

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 folder and are used to preserve key service integration message attributes when messages are sent to, then retrieved from, WebSphere MQ.

Table 27.

MQRFH2 header field	Equivalent SIBusMessage field or property	Description
JsApiUserId	Application user identifier (JMSXUserId)	The service integration application user identifier.
JsDst	Message format	The service integration JMS destination to which the message was sent.
JsFmt	JMS destination	The service integration message format.
JsSysMsgId	System message identifier	The service integration system message identifier assigned to the message.

When a message is sent to WebSphere MQ using a WebSphere MQ server, a sib folder is included in the MQRFH2 header of the message if both of the following are true:

- TheWebSphere MQ server definition is configured to use RFH2 headers.
- Fields corresponding to the sib folder content are set in the service integration message.

Mapping the JMS Destination property between service integration and WebSphere MQ when using a WebSphere MQ server

Service integration and WebSphere MQ JMS destinations are fundamentally different making it is impossible to map between the two representations of JMS destination property. Instead of mapping between the two JMS destination property representations, when a message leaves service integration and enters WebSphere MQ, an additional RFH2 property is introduced into the RFH2 header to store the service integration JMS destination.

The Service Integration JMS destination is serialized, then formatted as a hexadecimal string before being stored using the JsDst property of the sib RFH2 folder.

In an example where a service integration destination (SIQ1), that is localised on a WebSphere MQ queue (MQQ1), residing on queue manager QM1 has been configured, the following actions will be taken when a service integration JMS application sends a message to SIQ1:

- A serialised representation of PMQ1 will be placed into the sib RFH2 folder, using the JsDst property.
- The message will be stored on MQQ1.
- The string "queue://QM1/MQQ1" will also be placed into the jms RFH2 folder of the message using the Dst property.

This follows the convention used by the WebSphere MQ JMS to encode JMS destinations. If the message is retrieved by a service integration JMS application, then the JMS destination may be recovered from the contents of the sib folder RFH2 header. If the message is retrieved by a WebSphere MQ JMS application, then the JMS destination may be recovered from the contents of the jms folder in the RFH2 header.

Note: If the WebSphere MQ Server definition is configured not to use RFH2 headers, the JMS destination will not be preserved when the message enters WebSphere MQ. In this situation, a service integration JMS application is still able to retrieve the JMS message, however, any attempt to examine the JMS destination property will result in a JMS exception being thrown.

Mapping the Message Reliability property between service integration and WebSphere MQ when using a WebSphere MQ server
This topic describes the order in which message reliability mappings are applied.

There are several places where the reliability for a message can be set, overridden or mapped:

1. The reliability can be set when the message is created.
2. The message reliability can be mapped to a different level of reliability when it is sent to a destination. The settings which determine this mapping are a property of the service integration JMS Destination administer object.
3. If the message is sent to a service integration destination which is assigned to a WebSphere MQ queue, then the reliability is subject to the reliability mapping performed on all messages sent to WebSphere MQ. This process maps reliable persistent, or higher levels of reliability, to WebSphere MQ persistent reliability. All other levels of service integration reliability are mapped to WebSphere MQ non-persistent.
4. If a message is received from WebSphere MQ, then its WebSphere MQ persistence is mapped to a service integration reliability using the mappings defined as part of the WebSphere MQ Server Bus member from which the message is being retrieved.

Depending on whether a message is being sent to, or received by, a service integration destination assigned to a WebSphere MQ Server Bus member, the message reliability mappings listed above are applied in a different order.

For messages being sent to a service integration destination that is localized on a WebSphere MQ bus member, steps 1 to 3 (inclusive) apply in the order given.

For messages being received by a service integration destinations that are localized on a WebSphere MQ bus member, step 4 only applies.

Reply to queue constraints when using a WebSphere MQ server

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 Integraton.

- 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 which is a valid WebSphere MQ queue manager name.
- The reply to destination must reside in the same bus as the bus which 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.

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

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.

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`.

Writing a routing mediation

Use this topic to create a mediation that chooses a particular forward route for a message.

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 2 platform, Enterprise Edition (J2EE) session bean development environment such as the Application Server Toolkit (AST) or Rational Application Developer.

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 2 platform, Enterprise Edition (J2EE) 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 information.

4. Export the routing mediation enterprise application.

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.

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 2 platform, Enterprise Edition (J2EE) session bean development environment such as the Application Server Toolkit (AST) or 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

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 2 platform, Enterprise Edition (J2EE) 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 information.

4. Export the mapping mediation enterprise application.

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

If you have migrated your gateway, and the configuration included a routing filter that used the routing bean that is supplied with IBM WebSphere Application Server Version 5, then your routing filter still works in this version. However, if you did not use the supplied routing bean, then you must write a routing mediation and configure it to select a target service and port.

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.

Writing a WS-Notification application that exposes a Web service endpoint

Write a J2EE application, containing a JSR109 Web service definition, that can be deployed to the application server and act as a NotificationProducer, NotificationConsumer or demand based publisher.

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.

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 run-time environment". This wizard guides you through the following steps:

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. (e) The methods you choose depend upon the type of endpoint that you are exposing (NotificationProducer, NotificationConsumer or demand based publisher).

5. Export the application.

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 J2EE 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.

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.

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. The following steps follow the method provided by Rational Software Architect:

1. Get the WSDL for the service that you wish to invoke. If the target service is the notification broker inbound service that was generated by WebSphere Application Server, use the administrative console to navigate to **Service integration** › **Web services** › **WS-Notification services** › **[Content Pane]** *service_name* › **[Related items] Notification broker inbound service settings** › **[Additional Properties] Publish WSDL files to ZIP file** or **Service integration** › **Buses** › **[Content Pane]** *bus_name* › **[Services] WS-Notification services** › **[Content Pane]** *service_name* › **[Related items] Notification broker inbound service settings** › **[Additional Properties] Publish WSDL files to ZIP file**, then use the publish WSDL files property to export the template WSDL for this inbound 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."

You are now ready to deploy the application for use in the J2EE application client container as described in Running application clients.

Developing applications that use WS-Notification

Example code for common tasks that your WS-Notification applications can perform.

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, and those that do not expose a Web service endpoint. 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 651.

- "Writing a WS-Notification application that does not expose a Web service endpoint" on page 652.

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;
```

A single application can be coded to perform several WS-Notification tasks. Use the following examples to help you code these tasks into your WS-Notification applications:

- "Example: Subscribing a WS-Notification consumer" on page 654.
- "Example: Pausing a WS-Notification subscription" on page 656.
- "Example: Publishing a WS-Notification message" on page 656.
- "Example: Creating a WS-Notification pull point" on page 657.
- "Example: Getting messages from a WS-Notification pull point" on page 658.
- "Example: Registering a WS-Notification publisher" on page 658.
- "Example: Notification consumer Web service skeleton" on page 659.

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 Use pattern for WS-Notification as an entry or exit point to the SIBus 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 660.

Writing a WS-Notification application that exposes a Web service endpoint

Write a J2EE application, containing a JSR109 Web service definition, that can be deployed to the application server and act as a NotificationProducer, NotificationConsumer or demand based publisher.

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.

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 run-time environment". This wizard guides you through the following steps:

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. (e) The methods you choose depend upon the type of endpoint that you are exposing (NotificationProducer, NotificationConsumer or demand based publisher).

5. Export the application.

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 J2EE 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.

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.

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. The following steps follow the method provided by Rational Software Architect:

1. Get the WSDL for the service that you wish to invoke. If the target service is the notification broker inbound service that was generated by WebSphere Application Server, use the administrative console to navigate to **Service integration** → **Web services** → **WS-Notification services** → **[Content Pane]** *service_name* → **[Related items] Notification broker inbound service settings** → **[Additional Properties] Publish WSDL files to ZIP file** or **Service integration** → **Buses** → **[Content Pane]** *bus_name* → **[Services] WS-Notification services** → **[Content Pane]** *service_name* → **[Related items] Notification broker inbound service settings** → **[Additional Properties] Publish WSDL files to ZIP file**, then use the publish WSDL files property to export the template WSDL for this inbound 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 652.

You are now ready to deploy the application for use in the J2EE application client container as described in Running application clients.

Example: Subscribing a WS-Notification consumer

Example code that describes a 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 call (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.

```
// 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
EndpointReference consumerEPR =
    EndpointReferenceManager.createEndpointReference(new URI("http://myserver.mycom.com:9080/Consumer"));

// 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 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();

/*
Optional
--------
The following line causes the request to be targeted at a pull point. You must
```

do this if you want to subscribe a consumer to use pull-based notifications. The pullPointEPR
is the EndpointReference returned by the broker in relation to an invocation of the
CreatePullPoint operation.

```
    ((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, pullPointEPR);
*/

// 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

Example code that describes a 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 call (JAX-RPC) APIs in
conjunction with code generated using the WSDL2Java tool (run against the Subscription Manager WSDL
generated as a result of creating your WS-Notification service point) and WebSphere Application Server
APIs and SPIs.

```
// 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

Example code that describes a 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 call (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.

```
// 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

Example code that describes a 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 call (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.

```
// 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

Example code that describes a 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 call (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.

```
// 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

Example code that describes a 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 call (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.

```
// 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: 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 Use pattern for WS-Notification as an entry or exit point to the SIBus. 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.

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. IBM WebSphere Application Server provides several options for accessing an information system's backend data store:
- Programming directly to the database through the JDBC 2.0 optional package API or the JDBC 3.0 API.
- Programming to the procedural backend transaction through various J2EE 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. Consult the IBM Web address: http://www.ibm.com/software/webservers/appserv/doc/latest/prereq.html .

1. Develop data access applications. Develop your application to access data using the various ways available through the WebSphere 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 adapter

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.

An application server vendor extends its system once to support the J2EE 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.

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. It provides connection pooling, transaction, and security support. WebSphere Application Server supports JCA versions 1.0 and 1.5.

Data access for container-managed persistence (CMP) beans is managed by the WebSphere Persistence Manager indirectly. 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.

J2EE Connector Architecture resource adapters:

A J2EE Connector Architecture (JCA) resource adapter is any resource adapter conforming to the JCA Specification.

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 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.

Restriction: 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** > **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

WebSphere Relational Resource Adapter:

The WebSphere Relational Resource Adapter (RRA) provides enterprise applications deployed on WebSphere Application Server access to relational databases.

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 J2EE Connection Architecture (JCA) connection factories. The RRA supports the configuration and use of data sources implemented as either JDBC data sources or J2EE 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, see "Resource adapter" on page 662
- For information about resource adapters and data access, see "Data access portability features"
- For RRA settings, see "WebSphere relational resource adapter settings" on page 662
- For information about CMP connection factories, see "Connection factory" on page 667
- For information about enterprise beans, see EJB applications

Data access portability features: The WebSphere Application Server relational resource adapter (RRA) provides a portability feature that enables applications to access data from different databases without changing the application. In addition, 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 data store specific functions that the relational resource adapter run time uses. WebSphere Application Server provides an implementation for each supported JDBC provider.

In addition, 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 examples demonstrate the difference between a call to the XMLType.createXML() method over a direct connection to Oracle, and a call to the same method within WebSphere Application Server.

1. Over a direct connection:

```
XMLType poXML = XMLType.createXML(conn, poString);
```

2. Within Application Server, using the jdbcPass() method:

```
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}));
```

There are two different jdbcPass() methods available, one for use in invoking static methods, another for use when invoking non-static methods. See the API documentation **WSCallHelper** topic.

Note: Because of the possible problems that can occur by passing an underlying object to a method, 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, please contact WebSphere Application Server support with information on the method you require.

WARNING: Use of the jdbcPass() method causes the JDBC object to be used outside of WebSphere's protective mechanisms. 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.

Example: Developing your own DataStoreHelper class: The DataStoreHelper interface supports each data store platform plugging in its own private data store specific functions that are used by the Relational Resource Adapter run time.

```
package com.ibm.websphere.examples.adapter;

import java.sql.SQLException;
import javax.resource.ResourceException;

import com.ibm.websphere.appprofile.accessintent.AccessIntent;
import com.ibm.websphere.ce.cm.*;
import com.ibm.websphere.rsadapter.WSInteractionSpec;

/**
* Example DataStoreHelper class, demonstrating how to create a user-defined DataStoreHelper.
* Implementation for each method is provided only as an example.  More detail would likely be
* required for any custom DataStoreHelper created for use by a real application.
*/
public class ExampleDataStoreHelper extends com.ibm.websphere.rsadapter.GenericDataStoreHelper
{
    static final long serialVersionUID = 87889310901490908285L;
```

```
public ExampleDataStoreHelper(java.util.Properties props)
{
    super(props);

    // Update the DataStoreHelperMetaData values for this helper.
    getMetaData().setGetTypeMapSupport(false);

    // Update the exception mappings for this helper.
    java.util.Map xMap = new java.util.HashMap();

    // Add an Error Code mapping to StaleConnectionException.
    xMap.put(new Integer(2310),  StaleConnectionException.class);
    // Add an Error Code mapping to DuplicateKeyException.
    xMap.put(new Integer(1062),  DuplicateKeyException.class);
    // Add a SQL State mapping to the user-defined ColumnNotFoundException
    xMap.put("S0022",            ColumnNotFoundException.class);
    // Undo an inherited StaleConnection SQL State mapping.
    xMap.put("S1000",            Void.class);

    setUserDefinedMap(xMap);

    // If you are extending a helper class, it is
    // normally not necessary to issue 'getMetaData().setHelperType(...)'
    // because your custom helper will inherit the helper type from its
    //  parent class.


    }

public void doStatementCleanup(java.sql.PreparedStatement stmt) throws SQLException
{
    // Clean up the statement so it may be cached and reused.

    stmt.setCursorName("");
    stmt.setEscapeProcessing(true);
    stmt.setFetchDirection(java.sql.ResultSet.FETCH_FORWARD);
    stmt.setMaxFieldSize(0);
    stmt.setMaxRows(0);
    stmt.setQueryTimeout(0);
}


public int getIsolationLevel(AccessIntent intent) throws ResourceException
{
    // Determine an isolation level based on the AccessIntent.

    if (intent == null) return java.sql.Connection.TRANSACTION_SERIALIZABLE;

    return intent.getConcurrencyControl() == AccessIntent.CONCURRENCY_CONTROL_OPTIMISTIC ?
           java.sql.Connection.TRANSACTION_READ_COMMITTED :
           java.sql.Connection.TRANSACTION_REPEATABLE_READ;
}
    public int getLockType(AccessIntent intent) {
       if ( intent.getConcurrencyControl() == AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC) {
          if ( intent.getAccessType() == AccessIntent.ACCESS_TYPE_READ ) {
              return WSInteractionSpec.LOCKTYPE_SELECT;
          }
          else {
              return WSInteractionSpec.LOCKTYPE_SELECT_FOR_UPDATE;
          }
       }
       return WSInteractionSpec.LOCKTYPE_SELECT;
    }


public int getResultSetConcurrency(AccessIntent intent) throws ResourceException
```

```
    {
        // Determine a ResultSet concurrency based on the AccessIntent.

        return intent == null || intent.getAccessType() == AccessIntent.ACCESS_TYPE_READ ?
                java.sql.ResultSet.CONCUR_READ_ONLY :
                java.sql.ResultSet.CONCUR_UPDATABLE;
    }


    public int getResultSetType(AccessIntent intent) throws ResourceException
    {
        // Determine a ResultSet type based on the AccessIntent.

        if (intent == null) return java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE;

        return intent.getCollectionAccess() == AccessIntent.COLLECTION_ACCESS_SERIAL ?
                java.sql.ResultSet.TYPE_FORWARD_ONLY :
                java.sql.ResultSet.TYPE_SCROLL_SENSITIVE;
    }
}
```

ColumnNotFoundException

```
package com.ibm.websphere.examples.adapter;

import java.sql.SQLException;
import com.ibm.websphere.ce.cm.PortableSQLException;

/**
* Example PortableSQLException subclass, which demonstrates how to create a user-defined
* exception for exception mapping.
*/
public class ColumnNotFoundException extends PortableSQLException
{
    public ColumnNotFoundException(SQLException sqlX)
    {
        super(sqlX);
    }
}
```

Connection factory

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.

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, click **Resources** > **Resource Adapters** > **Resource Adapters** >*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 CMP bean to access any backend data store. This connection factory is only in ″read″ mode. It cannot be modified or deleted.

To view this administrative console page, click **Resources** >**Resource Adapters** > **Resource Adapters** > *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. This property is deprecated starting with version 6.0.

Data type   Drop-down list

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 J2EE 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.

See also Hardware and software requirements for more information. For detailed descriptions of the providers, including the supported data source classes and their required properties, refer to Vendor-specific data sources minimum required settings .

Data sources

Installed applications use a *data source* to obtain connections to a relational database. A data source is analogous to the J2EE 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.

    Note: In two cases, a connection pool data source does support two-phase commit transactions: when the JDBC provider is DB2 for z/OS Local JDBC provider (RRS), or when the data source is making use of *Last participant* support. Last participant support enables a single one-phase commit resource to participate in a global transaction with one or more two-phase commit resources. For more information, consult the article "Using one-phase and two-phase commit resources in the same transaction" on page 1116.

    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.
*   *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 WebSphere Application Server transaction manager provides transaction recovery.

In WebSphere Application Server releases prior to version 5.0, the function of data access was provided by a single connection manager (CM) architecture. This connection manager architecture remains available to support J2EE 1.2 applications, but another connection manager architecture is provided, based on the JCA architecture supporting the new J2EE 1.3 application style (also for J2EE 1.4 applications).

These two separate 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 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 J2EE 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 J2EE requirements levels. These differences are not related to the adoption of the JCA architecture.

If you have 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 (or later versions) applications cannot use the JDBC CM from WebSphere Application Server Version 4.0.

Connection pooling:

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, WebSphere 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.

WebSphere Application Server supports JDBC 3.0 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 WebSphere 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 WebSphere 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

How connections are pooled together

Whenever you configure a unique data source or connection factory, you are required to 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.

A separate instance of a given configured connection pool is created on each application server that uses that data source or connection factory. For example, if you run a three server cluster in which all of the servers use *myDataSource*, and *myDataSource* has a maximum connections setting of 10, then you can generate up to 30 connections (three servers times 10 connections). Be sure to consider this fact when determining how many connections to your backend resource you can support.

Other considerations for determining the maximum connections setting:
* Each entity bean transaction requires an additional database connection, dedicated to handling the transaction.

- On UNIX platforms, a separate DB2 process is created for each connection; these processes quickly affect performance on systems with low memory and cause errors.
- If clones are used, one data pool exists for each clone.

It is also important to note that when using *connection sharing*, it is only possible to share connections obtained from the same connection pool.

Avoiding a deadlock

Deadlock can occur if the application requires more than one concurrent connection per thread, and the database connection pool is not large enough for the number of threads. Suppose each of the application threads requires two concurrent database connections and the number of threads is equal to the maximum connection pool size. Deadlock can occur when both of the following are true:
- Each thread has its first database connection, and all are in use.
- Each thread is waiting for a second database connection, and none would become available since all threads are blocked.

To prevent the deadlock in this case, the maximum connections value for the database connection pool should be increased by at least one. Doing this allows for at least one of the waiting threads to obtain its second database connection and to avoid a deadlock.

To avoid deadlock, code the application to use, at most, one connection per thread. If the application is coded to require C concurrent database connections per thread, the connection pool must support at least the following number of connections, where T is the maximum number of threads.

```
T * (C - 1) + 1
```

The connection pool settings are directly related to the number of connections that the database server is configured to support.

If the maximum number of connections in the pool is raised, and the corresponding settings in the database are not raised, the application fails and SQL exception errors are displayed in the `stderr.log` file.

Deferred Enlistment: In the WebSphere Application Server environment, *deferred enlistment* is a term used to refer to the technique of waiting until a connection is first used to enlist it in its unit of work (UOW) scope.

In one example, the technique works like this: a component calls getConnection() from within a global transaction, and at some point later in time, the component uses the connection. The call that uses the connection is intercepted, and the XA resource for that connection is enlisted with the transaction service (which in turn calls XAResource.start()). Next, the actual call is sent to the resource manager.

In contrast, if a component gets a connection within a global transaction without deferred enlistment, then the connection is enlisted in the transaction and has all the overhead associated with that transaction. For XA connections, this includes the two phase commit (2PC) protocol to the resource manager. Deferred enlistment offers better performance in the case where a connection is obtained, but not used within the UOW scope. This saves all the overhead of participating in the UOW when it is not needed.

The WebSphere Application Server relational resource adapter automatically supports deferred enlistment without any additional configuration needed.

Lazy Transaction Enlistment Optimization: The J2EE Connector Architecture (JCA) Version 1.5 specification calls the deferred enlistment technique *lazy transaction enlistment optimization*. This support comes through a marker interface (LazyEnlistableManagedConnection) and a new method on the connection manager (LazyEnlistableConnectionManager()):

```
package javax.resource.spi;
import javax.resource.ResourceException;
import javax.transaction.xa.Xid;

interface LazyEnlistableConnectionManager { // application server
    void lazyEnlist(ManagedConnection) throws ResourceException;
}

interface LazyEnlistableManagedConnection { // resource adapter
}
```

A resource adapter is not required to support this functionality. Check with the resource adapter provider if you need to know if the resource adapter provides this functionality.

Connection and connection pool statistics: Performance Monitoring Infrastructure (PMI) method calls that are supported in the two existing Connection Managers (JDBC and J2C) are still 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:

In WebSphere Application Server Version 5.x, the JCAStats interface was implemented by the J2CResourceAdapter Mbean, and the JDBCStats interface was implemented by the JDBCProvider Mbean. The JCAConnectionStats and JDBCConnectionStats interfaces are not implemented because they collect statistics for nonpooled connections, which are not present in the JCA 1.0 Specification. JCAConnectionPoolStats, and JDBCConnectionPoolStats do not have a direct implementing Mbean; those 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:

WebSphere 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.

You can share a shareable connection with other components within the same transaction as long as each *getConnection()* request has the same connection properties. To enable connection sharing for data sources, the following connection 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

To enable connection sharing for resource adapters within the same transaction, the following connection properties must be the same:
- JNDI name. While not actually a connection property, this requirement simply means that you can only share connections from the same resource adapter in the same server.
- Resource authentication

In addition, the *ConnectionSpec* object used to get the connection must also be the same. For more information on sharing a connection with a CMP bean, see Sharing a connection with a CMP bean.

Java Message Service (JMS) connections cannot be shared with non-JMS connections.

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.

For WebSphere Application Server, all sharing of connections is relative to the current Unit of Work (UOW) boundary. Anyone within a specific transaction, when getting a connection from a specific connection pool, gets a handle to the same physical connection (if the sharing properties are the same).

Sharing a connection with a CMP bean

WebSphere Application Server allows you to share a physical connection between 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**

  There are two options to ensure that both CMP and BMP beans share the same physical connection:

  – 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 BMP is not 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 J2EE 1.3 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, WebSphere 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.

Connection sharing violations

There is a new exception, the `SharingViolation` 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 `SharingViolation` 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 `SharingViolation` 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 `SharingViolation` 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

In WebSphere Application Server Version 5.0, the Java 2 Platform, Enterprise Edition (J2EE) 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 J2EE 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 only share connections within a global transaction scope (assuming other sharing rules are met). However, 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 679 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*, then AutoCommit remains *TRUE* unless changed by the application.
- If you operate within an LTC and have its resolution-control set to *ContainerAtBoundary*, then the application should **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* since 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 1084 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 1084 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* that 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 you do multiple getConnection() calls using a resource reference that specifies res-sharing-scope=Unshareable, then 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** > **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 "Reference: Generated API documentation" on page 26 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 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 Specifications
- Container-managed relationships
- Resource adapters
- Miscellaneous articles from the Sun Developer Network and IBM developerWorks Web sites
- Rational Application Developer
- WebSphere Version 5.x Information Center
- IBM Cloudscape
- DB2 database software
- "IBM Informix" on page 689
- Supported hardware, software, and APIs

Programming Specifications
- Enterprise JavaBeans Technology (Source for download of the Enterprise Javabeans 2.1 specification)
- Java^{TM} 2 Platform, Enterprise Edition (J2EE^{TM})
- Java^{TM} Management Extensions (JMX)
- JDBC^{TM} 3.0 API Documentation
- J2EE Connector Architecture Version 1.5 specification
- What's New in the J2EE Connector Architecture 1.5
- What's New in the J2EE Connector Architecture 1.5 (Part 2)

Container-managed relationships

Though this article addresses the EJB 2.0 specification, you still might find parts of it pertinent to your environment.
- Enterprise JavaBeans^{TM} 2.0 Container-Managed Persistence Example

Resource adapters
- The J2EE Connector Architecture Resource Adapter

Miscellaneous articles from the Sun Developer Network and IBM developerWorks Web sites
- Developer Technical Articles & Tips -- Articles: Database Access (Sun Developer Network)
- Sharing connections in WebSphere Application Server V5 (This article is still pertinent to WebSphere Application Server Version 6.0. However, be aware that as of version 6.0, 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. However, be aware that the container-managed authentication type is now deprecated.)
- Understanding WebSphere Application Server EJB access intents

Rational Application Developer
- Rational Application Developer for WebSphere Software

WebSphere Version 5.x Information Center
- IBM WebSphere^{TM} Version 5.x Information Center

IBM Cloudscape
- IBM Cloudscape (product section in ibm.com)
- The IBM Cloudscape information center: http://publib.boulder.ibm.com/infocenter/cscv/v10r1/index.jsp.

DB2 database software
- DB2

IBM Informix
- http://www-306.ibm.com/software/data/informix/

Supported hardware, software, and APIs
- Supported hardware, software, and APIs

Developing data access applications

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 154 article for information on developing entity beans.

Tips:
- 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.
- Also consider using cursor holdability for potential performance gains; see the "Cursor holdability support for JDBC applications" on page 714 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, however; the EJB container handles this process for CMP beans.*

To run applications on WebSphere Application Server, your code must use resource references to logically name 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 J2EE 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, however; 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 J2EE 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.

Extensions to data access APIs

Applications can access the backend data through the standard J2EE 1.4 defined application programming interfaces (APIs). The standard APIs, however, do not always provide a complete solution for an application that runs in an application server. In some cases the JDBC programming model does not completely integrate with the J2EE Connector Architecture (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 J2EE 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: 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();
```

Example: Accessing data using IBM extended APIs for connections: 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 access an extended API in your JDBC application. Instead of using the DataSource interface, you use the WSDataSource interface. The following code segment illustrates how to get the connection.

```
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: Accessing data using IBM extended APIs to share connections between container-managed and bean-managed persistence beans: 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 access an extended API in your JDBC application. Instead of using the DataSource interface, you use the WSDataSource interface.

To ensure that both CMP and bean-managed persistence (BMP) beans are sharing the same physical connection, you can 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());
  }
 }
```

```java
/**
 * 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. You must run the ddl file on your database server to create the tables.

The following steps demonstrate the process for creating tables in DB2.
* Extract the Table.ddl file from your CMP enterprise bean JAR file to a working directory, such as *C:\temp*, on your DB2 machine.
* Run the command **C:\temp>db2cmd**, replacing *C:\temp* with your chosen directory. A DB2 command window is displayed, in which you enter the following commands (replacing *C:\temp* with your chosen directory):
   1. `C:\temp>db2 connect to` *your_db_name*
   2. `C:\temp>db2 -tf Table.ddl` //This command runs and creates tables for your CMP enterprise bean.
   3. `C:\temp>db2 disconnect all`

   Note: If you run DB2 on Unix, use these same commands. Simply run them from a user with permissions for DB2, rather than from a DB2 command window.
   Your tables are created.

Container-managed persistence features
The container-managed persistence (CMP) features include those defined by the EJB 2.1 Specification, as well as capabilities that are beyond the specification.

EJB Specification compliant capabilities

Container-Managed Relationships (CMR) is one of the most significant new features in recent versions of the EJB Specification. Like *Inheritance*, relationships are a key component of object-oriented software development and non-trivial object models can form complex networks with these relationships.

The container automatically manages the state of CMP entity beans. This management includes synchronizing the state of the bean with the underlying database when necessary and also managing any relationships (CMRs) with other entity beans. The bean developer is relieved of writing any database specific code and, instead, can focus on business logic.

Local interfaces are another feature introduced in recent versions of the EJB Specification. Local component interfaces allow co-located beans to interact without the overhead associated with remote access.

Value-add features

WebSphere Application Server provides enhancements to the function of CMP entity beans that supersede those capabilities defined by the specification. These include:

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 J2EE 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 the assembly tools" on page 188.

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 application assembly tooling (either Rational Application Developer or the Application Server Toolkit). See "Developing read-only entity beans" on page 155 for details.

Container-managed persistence restrictions and exceptions: The container-managed persistence (CMP) features have certain restrictions when used in specific ways.

Enterprise bean 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 (RDB).

- 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 container-managed persistence 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."

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 701.

Manipulating the synchronization of entity beans and datastores

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: You can set an EJB environment variable within the bean's deployment descriptor, or have the bean implementation class implement a marker interface. The 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.

1. **To use the EJB environment variable technique**, edit the EJB deployment descriptor using any standard Java 2, Enterprise Edition (J2EE) development tool. Use the following steps as a guide. (For information on your tool options, consult the "Assembly tools" on page 23 article.)

    a. Start the tool.

    b. Select the EJB deployment descriptor of the bean you want to work with.

    c. Create an EJB environment variable with the name **com/ibm/websphere/ejbcontainer/ DisableFlushBeforeFind**.

    d. Set the type of this variable to **java.lang.Boolean**.

    e. Set the value to True to prevent the pre-find synchronization, or False to enable the default behavior.

f. Save your changes.

2. **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 may 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 application-defined times. 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

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: You can set an EJB environment variable within the bean's deployment descriptor, or have the bean implementation class implement a marker interface. The 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.

1. **To use the EJB environment variable technique**, edit the EJB deployment descriptor using any standard Java 2, Enterprise Edition (J2EE) development tool. Use the following steps as a guide. (For information on your tool options, consult the "Assembly tools" on page 23 article.)

    a. Start the tool.

    b. Select the EJB deployment descriptor of the bean you want to work with.

    c. Create an EJB environment variable with the name **com/ibm/websphere/ejbcontainer/ disableEJBStoreForNonDirtyBeans**.

    d. Set the type of this variable to **java.lang.Boolean.**

    e. Set the value to True to avoid the ejbStore invocation, or False to enable the default behavior.

    f. Save your changes.

2. **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

Using a resource reference to access your data source or connection factory is required when running WebSphere Application Server. Reasons for this requirement include the following:

*   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 at the target application server.

*   You can set the default isolation level for the data source through resource references. With no resource reference you get the default for the JDBC driver you use.

Use a resource reference (resource-ref) for looking up a data source through the standard Java Naming and Directory Interface (JNDI) naming interface. The JNDI name defined in the resource reference is a logical name of the data source. Have your application use this JNDI name to look up a data source instead of using the JNDI name that is defined on the data source.

Later, you can substitute the real name, either by using an assembly tool or during installation of the application EAR file onto the server.

For example, assume that you use a data source *jdbc/Section* as illustrated in the following code:

```
javax.sql.DataSource specificDataSource  =
   (javax.sql.DataSource) (new InitialContext()).lookup("java:comp/env/jdbc/Section");
```

In the assembly tool, specify the name (*jdbc/Section*) as the resource reference. If you know the name of the data source, specify it in the resource references Bindings page.

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.

Isolation level requirements for different code specifications

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, versions 5.x and 6.x 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.*

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	Cloudscape	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)

Data source lookups for enterprise beans and Web modules:

During either application assembly or deployment, you must bind the resource reference to the actual name of the 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.

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.

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	Cloudscape	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.

New for MS SQL Server 2005: 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
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
Cloudscape	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 at the Web address http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/db2/rbafzmst02.htm .

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

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.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Process Definition** > **Control** (to define the property in the Control) or **Servant** (to define the property in the Servant) > **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(...)

Data access from J2EE Connector Architecture applications

To access data from a J2EE Connector Architecture (JCA) compliant application in WebSphere Application Server, you configure and use resource adapters and connection factories.

Example: Connection factory lookup:

```
import javax.resource.cci.*;
import javax.resource.ResourceException;

import javax.naming.*;

import java.util.*;

/**
 * This class is used to look up a connection factory.
 */
public class ConnectionFactoryLookup {

   String jndiName = "java:comp/env/eis/SampleConnection";
   boolean verbose = false;

   /**
    * main method
    */
   public static void main(String[] args) {
      ConnectionFactoryLookup cfl = new ConnectionFactoryLookup();
      cfl.checkParam(args);

      try {
         cfl.lookupConnectionFactory();
      }
      catch(javax.naming.NamingException ne) {
   System.out.println("Caught this " + ne);
   ne.printStackTrace(System.out);
      }
      catch(javax.resource.ResourceException re) {
   System.out.println("Caught this " + re);
   re.printStackTrace(System.out);
      }
   }

 /**
  * This method does a simple Connection Factory lookup.
  *
  * After the Connection Factory is looked up, a connection is got from
 * the Connection Factory.  Then the Connection MetaData is retrieved
  * to verfiy the connection is workable.
  */
 public void lookupConnectionFactory()
  throws javax.naming.NamingException, javax.resource.ResourceException {

  javax.resource.cci.ConnectionFactory factory = null;
  javax.resource.cci.Connection conn = null;
  javax.resource.cci.ConnectionMetaData metaData = null;
```

```java
    try {
      // lookup the connection factory
        if (verbose) System.out.println("Look up the connection factory...");

      InitialContext ic = new InitialContext();
      factory = (ConnectionFactory) ic.lookup(jndiName);

      // Get connection
        if (verbose) System.out.println("Get the connection...");
      conn = factory.getConnection();

      // Get ConnectionMetaData
      metaData = conn.getMetaData();

      // Print out the metadata Informatin.
      if (verbose) System.out.println("  ** EISProductName   :" + metaData.getEISProductName());
      if (verbose) System.out.println("     EISProductVersion:" + metaData.getEISProductName());
      if (verbose) System.out.println("     UserName          :" + metaData.getUserName());

        System.out.println("Connection factory "+jndiName+" is successfully looked up");
    }
    catch (javax.naming.NamingException ne) {
      // Connection factory cannot be looked up.
      throw ne;
    }
    catch (javax.resource.ResourceException re) {
      // Something wrong with connections.
      throw re;
    }
    finally {
      if (conn != null) {
        try {
          conn.close();
        }
        catch (javax.resource.ResourceException re) {
        }
      }
    }
}

  /**
   * Check and gather all the parameters.
   */
  private void checkParam(String args[]) {
  int i = 0, j;
  String arg;
  char flag;
      boolean help = false;

  // parse out the options
  while (i < args.length && args[i].startsWith("-")) {
    arg = args[i++];

    // get the database name
    if (arg.equalsIgnoreCase("-jndiName")) {
      if (i < args.length)
        jndiName = args[i++];
      else {
        System.err.println("-jndiName requires a J2C Connection Factory JNDI name");
        break;
      }
    }
    else { // check for verbose, cmp , bmp
      for (j = 1; j < arg.length(); j++) {
        flag = arg.charAt(j);
        switch (flag) {
          case 'v' :
```

```
    case 'V' :
     verbose = true;
     break;
                case 'h' :
                case 'H' :
                    help = true;
                    break;
    default :
     System.err.println("illegal option " + flag);
     break;
   }
  }
 }
}

 if ((i != args.length) || help) {
  System.err.println("Usage: java ConnectionFactoryLookup [-v] [-h]");
  System.err.println("    [-jndiName the J2C Connection Factory JNDI name]");
  System.err.println("-v=verbose");
  System.err.println("-h=this information");
  System.exit(1);
 }
  }

}
```

J2EE Connector Architecture migration tips:

Versions of WebSphere Application Server previous to Version 5.0 provided an initial implementation of the J2EE Connector Architecture (JCA) specification, Version 1.0. This implementation provided basic run time support based on the final JCA 1.0 Specification, but it was not a complete implementation.

Version 5.0 of the product provides a complete implementation of the JCA 1.0 Specification, which supports:
* Connection sharing (*res-sharing-scope*).
* Get/use/close programming model for connection handles.
* Get/use/cache programming model for connection handles.
* *XA*, *Local*, and *No Transaction* models of resource adapters, including XA recovery.
* Security options A and C per the specification.
* Applications with embedded .rar files

> **Additional feature:** Version 5.0 also provides connection pooling, for increasing the efficiency of connection usage.

As of Version 6.0, the product provides a complete implementation of the JCA 1.5 Specification, which supports:
* All the features of the JCA 1.0 Specification.
* Deferred enlistment transaction optimization.
* Lazy connection association optimization.
* Inbound communication from an enterprise information system (EIS) to a resource adapter.
* Inbound transactions from an EIS to a resource adapter.
* Work management, which enables a resource adapter to put work on separate threads and to pass execution context (such as inbound transactions) to the thread .
* Life cycle management, which enables a resource adapter to be stopped and started.

Moving from an early implementation of JCA

If you move from one of the earlier implementations of the J2EE Connector Architecture to the current implementation, be aware of the following corresponding changes in WebSphere Application Server:

- This version of the product supports the *res-sharing-scope* tag within the resource reference (resource-ref) element. This tag was not available in previous versions and defaulted to *shareable* connections. Beginning with Version 5.0, WebSphere Application Server supports **both** shareable and unshareable connections.
- The current product supports the Web container. Both enterprise bean and Web components can utilize the J2EE Connector Architecture.
- Both connection handle usage patterns (get/use/close and get/use/cache) are supported. The get/use/close pattern indicates that a connection is retrieved, used, and closed all within the same transaction or method boundary. The get/use/cache pattern indicates that you can cache a connection across transaction or method boundaries.
- The current version supports additional authentication mechanisms. The capability to support Options A and C per the JCA specification is provided, as well as support for *res-auth* settings of either *Application* or *Container*. In versions before Version 5.0, the *res-auth* setting was basically ignored, therefore it was treated as if *res-auth* was set to *Application*. If your existing applications have *res-auth* set to *Container*, they might behave differently if you install them into a current environment without any changes.
- As of Version 6.0, resource authentication for *res-auth* settings of *Container* is preferably specified on the resource-reference mapping during application deployment. Specification of container-managed authentication on a data source or connection factory is deprecated.
- As of Version 5.0, you can no longer specify pool and subpool names. The pool name is based on the data source or connection factory Java Naming and Directory Interface (JNDI) name. Subpools were eliminated to provide better performance.
- As of Version 6.0, configuration data formerly in the *j2c.properties* file is now supported through the wsadmin scripting tool and the administration console. A migration utility updates the *resources.xml* file (or files) based on the settings in a *j2c.properties* file. A template *j2c.properties* file is no longer placed in the installed directory tree, but run-time code remains in place to process the file and favor its settings over those from the real configuration.

Migration involving first-time use of connection pooling

If you are upgrading to WebSphere Application Server V6.x from a version prior to V5.0, be aware of the run-time behavior changes that your applications might incur because of the new connection pooling feature in the product.

Although not required by the J2EE architecture, connection pooling support is provided by the connection management component of WebSphere Application Server to help improve the performance of getting and using connections to a backend, such as a database or transaction resource manager (CICS or IMS, for example). The connection pooling support is provided, individually, to every data source and connection factory that you configure. The properties associated with each connection pool have default values that are sufficient for most application server environments. However, in some cases, the default values might not meet the needs of application connection requests, and result in problems such as ConnectionWaitTimeout exceptions.

Therefore you must consider the application requirements of each data source and connection factory that you configure, and set the corresponding connection pool properties appropriately. The Connection pool settings and Tuning parameters for data access resources topics provide good reference for setting the properties.

Migration issue for the combination of Web services and JCA connectors

For applications that use Web services and JCA connectors, be aware that those generated on WebSphere Studio Application Developer -- Integration Edition Version 4.1.1 can run unchanged on WebSphere Application Server Version 6.x only if they are regenerated using WebSphere Studio Application Developer -- Integration Edition Version 5.0 tools, or Rational Application Developer tools. This limitation is because of the *wsd14j.jar* file. As delivered in WebSphere Application Server Enterprise Version 4.1, the file is not fully compliant with JSR 110 (because JSR 110 was not final at the time that Version 4.1 shipped). The *wsd14j.jar* file shipped with WebSphere Application Server Version 6.0 and later,

of course, is compliant. However, because most of the classes have the same package names and interfaces, BUT NOT ALL, the two *wsd14j.jar* files cannot co-exist in the same WebSphere Application Server installation.

JDBC provider templates: important general migration tip

Always handle the jdbc-resource-provider-templates.xml file as read-only . When updating this file, special consideration should be taken. Before installing a PTF, you should save your updated jdbc-resource-provider-templates.xml file. After applying the PTF, you will need to verify that the new jdbc-resource-provider-templates.xml file has your correct entries. If the entries are not valid, you will have to merge your changes into this new jdbc-resource-provider-templates.xml file manually.

Accessing data using J2EE Connector Architecture connectors

As indicated in the J2EE 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.

For each EIS resource, do the following:
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. Configure, during deployment, each resource adapter and associated connection factory through the console. See Configuring J2C resource adapters and Configuring J2C connection factories 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.

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 */}
   }
}
```

Cursor holdability support for JDBC applications

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 is moot for 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

Data access beans are essentially a class library that makes it easier to access a database. The library contains a set of beans with methods that access the database through the JDBC API. 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 J2EE 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 preconfigured data source object.

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

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 in that it does not support the retrieval of objects, as is the case with Enterprise JavaBeans (EJB) persistence mechanisms. Results are retrieved as a structured graph of data (the DataGraph).
- 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 DataGraphs 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.

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 Connectivity Mediator Service" on page 719 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 "Enterprise JavaBeans Data Mediator Service" on page 733. 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. Only the DMS accesses data sources, SDO applications do not. SDO applications only work with DataObjects in DataGraphs.

Each of the following components corresponds to a Java interface in the SDO programming model.

DataObjects

DataObjects are the fundamental components of SDO. In fact, they are the *service DataObjects* found in the name of the specification itself. DataObjects are the SDO representation of structured data which can hold multiple different attributes of any serializable type (such as string or integer), including other DataObjects. Each DataObject also has a *type* (see "SDO data object types" on page 722 for more information). DataObjects are generic and provide a common view of structured data built by a DMS. While a JDBC DMS, for instance, needs to know about the persistence technology (for example, relational databases) and how to configure and access it, SDO clients need not know anything about it. DataObjects hold their data in *properties* . DataObjects provide convenience creation and deletion methods (createDataObject() with various signatures and delete()), and reflective methods to get their types (instance class, name, properties, and namespaces). DataObjects are linked together and contained in DataGraphs. DataObjects have

different ways to access linked data, the two most common being through XPath expressions and by a property index. DataObjects keep track of the original value of any attribute that is modified.

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

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 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, including a small sample SDO application, refer to 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 "Task overview: Using enterprise beans in applications" on page 145, and "Service Data Objects: Resources for learning" on page 24

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:
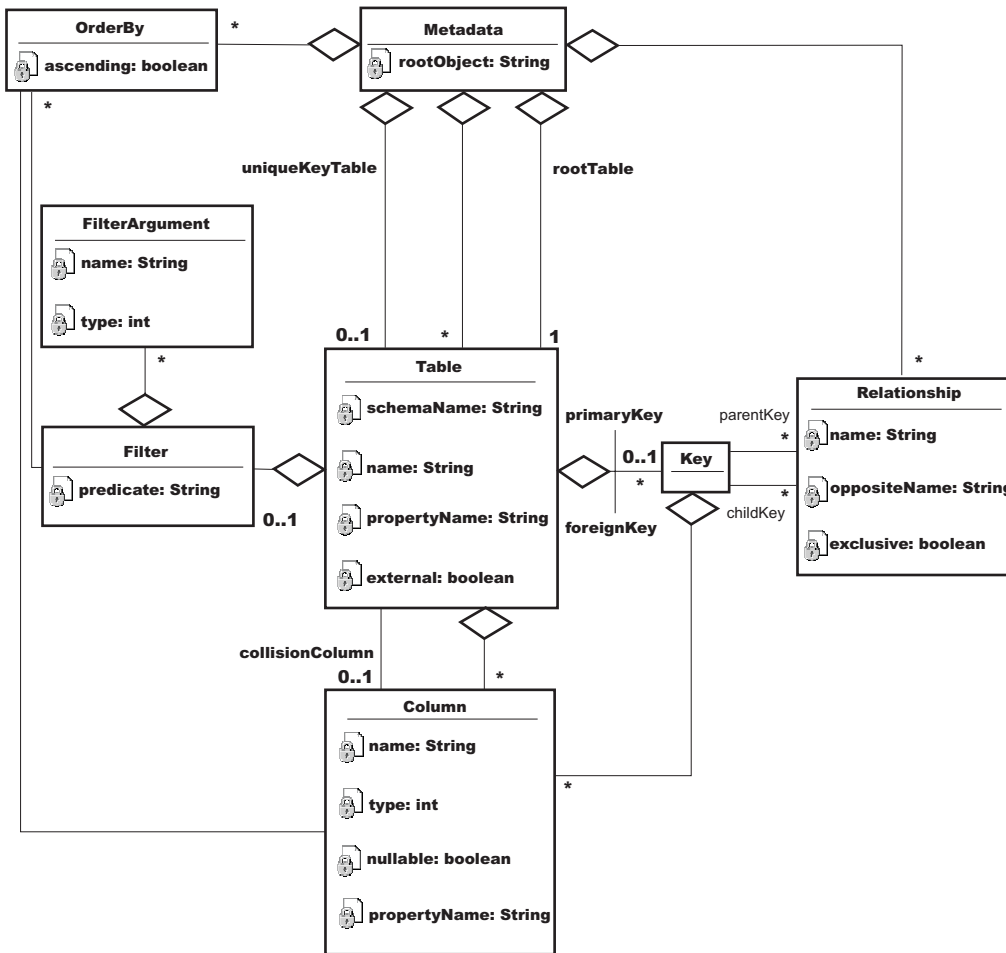
Table This represents a table within the target database and is composed of the following items:

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.

SDO data object types: DataObjects in the Service Data Object (SDO) can use either dynamic or static types. With dynamic types, the information that defines the shape of a DataGraph is constructed at runtime. Clients must use a DataGraph dynamic API to access data when using dynamic types. 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 tools to generate strongly typed interfaces that simplify DataGraph navigation, provide better compile-time checking for errors, and improve performance.

The tools 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.

When using typed DataObjects, the dynamic API is still available. 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: Although the JDBC Data Mediator Service (DMS) generates a SELECT statement from the metadata provided at the creation of an instance, the DMS also enables the client to provide a specific SELECT statement to be used instead of the generated one. The provided statement is a standard structured query language (SQL) SELECT string and can contain parameter markers. Using supplied queries gives you more control over the data used to populate a DataGraph. 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.

Limitation: 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);
```

Limitation: 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.

Limitation: 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:

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:
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.

The default action is to manage transactions.

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.

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:

```
                        <<signal>>
                     MediatorException
                     _____

                     _____
                     _____


                        <<signal>>
                   JDBCmediatorexception
                   _____

                   _____


        <<signal>>                          <<signal>>
       DBexception                     invalidmetadataexception
       _____                     _____

       _____                        🔒 errorcode: int


     <<signal>>
    OCCexception
    _____

  🔒 originalDO: dataobject

  🔒 changedDO: dataobject

  🔒 databaseDO: dataobject
```
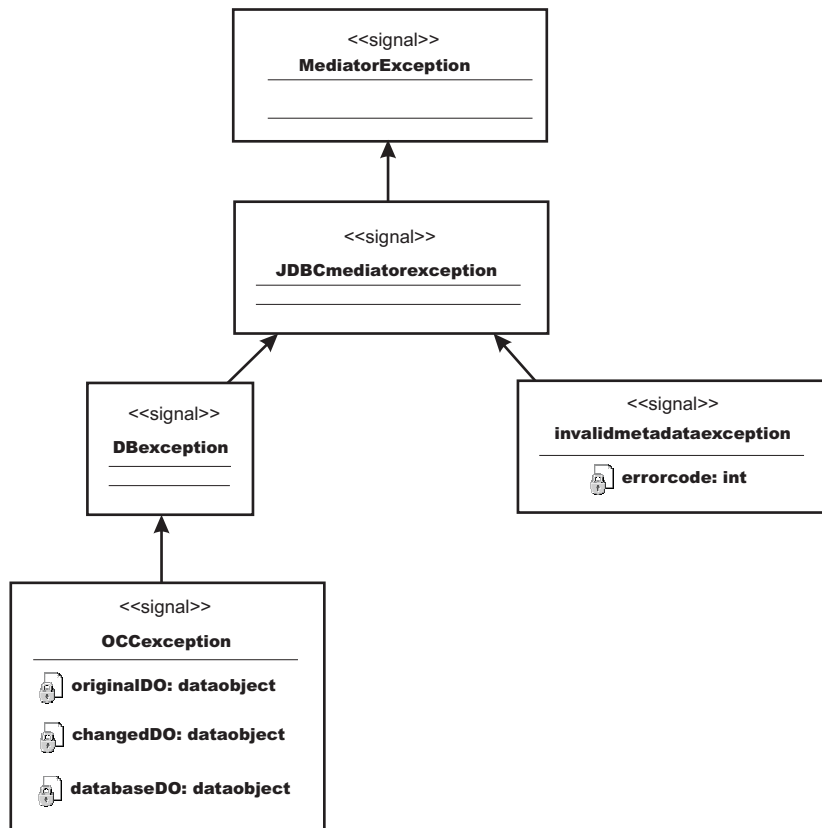
The `Mediator` exception is the root exception of all the data mediator services, and the `JDBCMediator` exception is the root exception for the JDBC DMS in particular.

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: OCC data collisions and JDBC mediator: 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"));
}
```

Monitoring optimistic concurrency control collisions:

To diagnose transaction problems that are caused by update collisions, implement an optimistic concurrency control (OCC) strategy for the JDBC DMS.

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 full-fledged code example that forces a collision to demonstrate the OCC exception, see the topic "Example: OCC data collisions and JDBC mediator" on page 727.

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). A general understanding of both of these technologies is

assumed for this section. 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 24.

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: JavaServer Faces and JDBC Mediator"

Example: JavaServer Faces and JDBC Mediator: This code would be located inside of a Faces JSP page. 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. 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 differs from a normal EJB finder or ejbSelect method, which also takes an EJB query but returns a collection of EJB objects (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 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 deals with creating, populating, and updating copy helper objects can be replaced with a DataGraph and a DMS.

Important: The EJB Data Mediator Service has support for EJB2.x container managed persistence (CMP) entity beans only.

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 as it avoids EJB activation. However, if business logic or EJB container function is required by the application, then writing back through EJB is the preferred approach. When writing back through EJB, you can specify a user defined MediatorAdapter to allow 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 EJB 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 update to insure that the current value is still equal to the old value of the field in the DataObject.

Runtime processing

An EJB mediator request is a compound EJB query, which consists of an ordered list of more or less 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 which 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 744.

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 WebSphere Application Server, consider the following items.

EJB programming model

Only a subset of the Enterprise JavaBeans 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 737 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 (create, getGraph, 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

- It is allowable to 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: EJB mediator query arguments" on page 738.

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 WebSphere Application Server, 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: EJB mediator query arguments:

The following examples show how you can fine-tune your 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 737). 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 will iterate 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 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 Enterprise JavaBeans (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:
- Specifications: 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:
* IBM Faces Component Catalog
* 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 use code samples to describe a simple instance of how to create the Java Database Connectivity (JDBC) data mediator service (DMS) metadata.

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 there is a method called *connect()* that does that.

10. Create the JDBC DMS object (DataGraph) using this metadata. 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."

11. Submit the changed information to the database.

Example: manipulating data in a DataGraph: Using the simple DataGraph that was created during the task "Using the Java Database Connectivity data mediator service for data access" on page 746, 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 Enterprise JavaBeans data mediator service for data access

The following steps use code samples to describe a simple instance of how to create the Enterprise JavaBeans 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		

Establishing custom finder SQL dynamic enhancement server-wide

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

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. 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 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

To establish this support for specific custom finders use the following steps.

1. Start a J2EE 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

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 J2EE 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.

Embedded Structured Query language in Java (SQLJ) support

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.

Advantages of developing applications with SQLJ include improved performance and a shorter, more efficient development cycle:
- You improve performance by using static SQL statements.
- You reduce the development cycle because:
  - You write less code with the simpler SQLJ syntax, which reduces the number of lines of code required to execute statements, and to set and retrieve parameters.
  - You detect programming errors earlier in the development phase with the onlinecheck 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.

You might consider using SQLJ in situations where dynamic SQL is not needed, and where applications use DB2 as the backend database.

Using embedded Structured Query Language in Java (SQLJ) support

WebSphere Application Server Version supports Structured Query Language in Java (SQLJ) with the DB2 universal JDBC driver and the DB2 Legacy CLI-based Type 2 JDBC driver. Perform the following steps to convert existing JDBC applications to SQLJ in WebSphere Application Server.

1. Acquire the required drivers to run SQLJ.
   - DB2 Legacy CLI-based JDBC driver

     You must have the `db2java.zip` and `db2jcc.jar` files in your class path. Define these files in the class path of the WebSphere Application Server DB2 Legacy CLI-based JDBC Driver Provider.
   - DB2 Universal JDBC driver

     You only need the `db2jcc.jar` file.
   - WebSphere Application Server driver
2. Review the appropriate task file for instructions on using SQLJ.
   - Enterprise Java Bean (EJB) Container Managed Persistence (CMP) beans, or with
   - EJB Bean Managed Persistence (BMP) entity beans, session beans, and servlets

Using Structured Query Language in Java (SQLJ) for Enterprise Java Bean (EJB) container-managed persistence (CMP) beans:

Structured Query Language in Java (SQLJ) for EJB Persistence requires DB2 Version 8 FixPak1 or later. You need DB2 UDB Application Development Client Version 8 FixPak1 (or later) on the machine where you generate the deployed code, and DB2 UDB Server (Version 8 FixPak1 or later for Unix and Windows, and Version 6 or Version 7 for DB2 on z/OS and OS/390) on the machine containing the database for running the scenario.

SQLJ support requires Version 2.x CMP Enterprise Java Beans. If you mix EJB 1.1 and 2.x beans in an EJB 2.x module, then only the EJB 2.x beans are deployed to use SQLJ. The EJB 1.1 beans continue using JDBC for data access.

The deployment command for Enterprise Java Beans (EJBDeploy) only provides SQLJ support for the Version 2.0 CMP beans. It is your responsibility to manually translate and customize the profile on the database for the Bean Managed Persistence (BMP) entity beans, session beans, and servlet SQLJ applications. See the Using SQLJ for Enterprise Java Bean (EJB) Bean Managed Persistence (BMP) entity beans, session beans, and Servlets article for more information.

WebSphere Application Server provides tools for using SQLJ as the persistence mechanism for CMP Enterprise Java Beans. You can deploy the CMP beans to use SQLJ either by using the EJB deploy tool in Rational Application Developer (RAD), or by using the command utility **ejbdeploy** with the **-sqlj** option.

You cannot deploy a CMP bean with the **sqlj** option from the administrative console or the assembly tool.

When you install an application comprised of CMP beans, you have the following choices:
- For data access with SQLJ, deploy the CMP beans to use SQLJ before installing the application in WebSphere Application Server. Perform deployment within the RAD environment, or by using the command utility **ejbdeploy** with the **-sqlj** option.

  Do not deploy this bean again in the administrative console when you install the application. If you pre-deploy this bean with the **-sqlj** option and then redeploy it from the administrative console when you install the application, this bean deploys with JDBC access.
- For JDBC access, use the **ejbdeploy** command or the administrative console to deploy the CMP bean.

  If the bean is not deployed, the installation of the application fails.

  1. Deploy the EAR file.

a. Verify that the *was_home*/`bin` directory is in your PATH statement.

b. Verify that the `ws_ant.bat` file is available on your system.

   You need the `ws_ant.bat` file to run the rest of the commands.

   Note: On Windows platforms, the Apache Ant tool is `ws_ant.bat` On Unix systems, the Ant tool is `ws_ant`.

c. Run the **ejbdeploy** command utility with the **-sqlj** option.

2. Perform the following SQLJ profile customization steps if your application is running in a clustered environment.

   You must supply the location of the SQLJ translator `sqlj.zip` file with the **-cp** (class path) option:

   ```
   ejbdeploy d:\MyApplication.ear
             working d:\deployedMyApplication.ear
             -sqlj
             -dbvendor DB2UDB_V81
             -cp "C:\PROGRA~1\IBM\SQLLIB\java\sqlj.zip;C:\PROGRA~1\IBM\SQLLIB\java\sqlj.zip "
   ```

   The **ejbdeploy** command does not access sqlj.zip from your system class path. The **ejbdeploy** command will generate an EAR file with the name you specify as, for example, `deployedMyApplication.ear`, and an Ant script with the name *InputEarName*.`ear.xml`, or in this example, `deployedMyApplication.ear.xml`.

   If you create the EAR file using Rational Application Developer, you can associate SQLJ with a particular database backend in the mapping editor. If you use the mapping editor, you do not have to specify the **-sqlj** option in the **ejbdeploy** command.

   If your application is not running in a clustered environment, go to the Ant tool properties description.

   Perform customization once on a single host.

   a. Run the DB2 SQLJ customizer, **db2sqljcustomize**, against the serialized profiles.

      One serialized profile exists for each EJB `.jar` file. You can find the serialized profile in the EJB `.jar` file. One example of a serialized profile file name is `FS_TopDown1_SJProfile0.ser`.

      When you run the DB2 SQLJ customizer with the "**-automaticbind yes**" default option against the serialized profiles, you create static SQL in the database, which is used at runtime. The customization phase creates four database packages that contain static SQL, one for each isolation level.

   b. Include the customized profiles in the EJB `.jar` files installed on WebSphere Application Server.

      The customization step also updates the generated customized profiles. To make these updated customized profiles available to the generated code at runtime, include the profiles in the EJB `.jar` file, which is installed in WebSphere Application Server.

      If you omit the customization step, the EJB applications run, but do not use the static SQL stored in the database, and you lose all the benefits of SQLJ.

   c. Use the Ant script to make customization easier.

      When you run batch SQLJ `ejbDeploy` against an EAR file, it produces an Ant script. Use this script file to run the DB2 customizer program against every serialized profile in every EJB `.jar` file in the associated EAR file. The script updates each EJB `.jar` file with a serialized profile, and replaces the `.jar` files in the EAR file with the modified versions. The Ant script is specific to the corresponding EAR file.

      This script modifies the existing EAR file.

      The script also uses a set of default names for the packages created in the database. Change the names used by the script file to ensure the names for each customization profile do not conflict with existing package names in the database. Ant scripts generated for different EAR files use the same package names by default, and overwrite existing packages unless you change the names. Overwritten packages cause run-time errors.

   d. Change the values of the database URL, and the database user and password properties in the generated Ant script.

The package names, database URLs, users, and passwords are created in the script using Ant properties.

The Ant script defines the following global properties for the:
- Database URL - `db.url`
- User - `db.user`
- Password - `db.password`

e. Change the values used by the Ant script and the serialized profile by specifying new values in a property file.

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.

The script for a particular EAR file reads properties from two files:
- `ejbdeploy.sqlj.properties`

  The `ejbdeploy.sqlj.properties` file is common to all Ant scripts generated by the **EJBDeploy** command. Use the `ejbdeploy.sqlj.properties` file to specify global properties, such as the database user and password.
- *ear-name*`.properties`

  The *ear-name* value is the name of the EAR file. The *ear-name*`.properties` file is specific to the Ant script for the EAR file. If you want your Ant script to use a another file instead of *ear-name*`.properties`, specify the `script.property.file` property when you run the script.

f. Use the DB2 Control Center to identify the packages 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.

3. Run the Ant script, specifying the **properties** target:

```
ws_ant -buildfile deployedMyApplication.ear.xml properties
```

This script creates the properties file, `deployedMyApplication.ear.properties`. The `deployedMyApplication.ear.properties` file contains properties specifying default names for the packages corresponding to each serialized profile in the EAR file, as for example:

```
pkg.MyEJB1.jar.DB2UDBNT_V8_1=PKG1_
pkg.MyEJB2.jar.DB2UDBNT_V8_1=PKG2_
```

In this example, the EAR file contains two EJB `.jar` files: `MyEJB1.jar` and `MyEJB2.jar`.

4. Edit the generated properties file to change the package names.

```
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
```

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 customizer will create packages called:

```
TEST1, TEST2, TEST3, TEST4
```

You can also specify other properties in this file, such as the database URL, user, or password.

5. Use the following **db2sqljcustomize** options to temporarily circumvent profile customization problems. These options bypass errors during a profile customization and ensure a successful customization:

```
-onlinecheck NO and -bindoptions "VALIDATE RUN"
```

However, you must understand what the problems are and fix them at the appropriate time.

6. Run the Ant script.

   The DB2 `db2jcc.jar` file must be on the class path. This file should have been added to the class path environment variable when DB2 V8 FixPak1 was installed.

7. Specify a working directory for the Ant 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 the files and directories.

   Use the following command to specify a working directory:

   ```
   ws_ant -Dwork.dir=tmp
          -buildfile MyApplication.ear.xml
   ```

   The script creates and uses a directory called `tmp` as its working directory. If you want the script to use a different property file, set the `script.property.file` property when you run the script:

   ```
   ws_ant -Dwork.dir=tmp
          -Dscript.property.file=other.properties
          -buildfile MyApplication.ear.xml
   ```

   The Ant script updates the original EAR file with the modified serialized profiles.

8. If you rerun the **EJBDeploy** command, rerun the Ant script.

   Generate a new properties file if any of the following conditions are true:
   - You change the number of `.jar` files in the EAR file.
   - You change the names of the `.jar` files in the EAR file.
   - You change the database backend ids in any of the `.jar` files

9. Install the updated EAR file in WebSphere Application Server.

10. Create a JDBC provider and data source in WebSphere Application Server.

    Generate a new properties file if any of the following conditions are true:
    - You change the number of `.jar` files in the EAR file.
    - You change the names of the `.jar` files in the EAR file.
    - You change the database backend ids in any of the `.jar` files

    Generate a new properties file if any of the following conditions are true:
    - You change the number of `.jar` files in the EAR file.
    - You change the names of the `.jar` files in the EAR file.
    - You change the database backend IDs in any of the `.jar` files

11. Install your application through the administrative console but do not redeploy the EJB. If you check the **EJBDeploy** box, your application is redeployed to JDBC access.

12. Stop the server.

13. Start the server.

    You are now ready to run your application.

Using Structured Query Language in Java for bean-managed persistence entity beans, session beans, and servlets:

This article describes how JDBC applications, comprised of bean-managed persistence (BMP) entity beans, session beans, or servlets, are converted to SQLJ applications. It also describes how the SQLJ applications are then deployed in WebSphere Application Server.

Follow these steps precisely and in the right order to ensure a correct conversion:

1. 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. 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`.

Now when you run the **sqlj** tool in the next step, the `.java` file that it creates will have the same name as your old `.java` file, providing you with a seamless transition to the SQLJ technology.

3. Edit the `.sqlj` file to convert the JDBC syntax to SQLJ syntax. When using SQLJ in WebSphere Application Server, if you want WebSphere Application Server connection management 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();
   ```

4. Run the DB2 SQLJ translator. This tool creates a `.java` version of your `.sqlj` file, as well as a `.ser` profile that is used later in the processing. Refer to the DB2 documentation for more information on the SQLJ translator tool.

5. Package your EJB jar and deploy it in the usual manner. After deployment, one serialized profile exists for each EJB `.jar` file. (You can find the profile in the EJB .jar file; one example of a serialized profile file name is MyBMPBeanProfile.ser.)

6. Run the `db2sqljcustomize` tool to customize the `.ser` files that correspond to each EJB `.jar` file. When you run the DB2 SQLJ customizer against the serialized profiles, you create static SQL in the database, which is used at runtime. The customization phase creates four database packages that contain static SQL, one for each isolation level.

   Optionally, you can use the SQLJ customizer tool to enable **context caching** for your application's data source connections, thereby improving application performance.

   - DB2 V8.1 fix pack 6 provides the new caching option, called `db2optimize` , with the `db2sqljcustomize` tool. You can run this option if your application uses the explicit connection context instead of the default context.

   - The following example code demonstrates proper syntax for running the option:

   ```
   sqlj -db2optimize SQLJTransactionTest.sqlj
   db2sqljcustomize -url jdbc:db2://localhost:50000/jtest -user dbuser1 -password dbpwd1
   SQLJTransactionTest_SJProfile0.ser
   ```

   - If you want to enable context caching for an application or BMP bean that caches connections across transaction boundaries, you cannot use shareable connections. You must use the get/use/close pattern of connection usage when you invoke the `db2optimize` option. Otherwise, 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.

7. Configure your database.
8. Update your EJB jar with the .ser file.
9. Package your EJB jar and servlets, along with .ser files, into an .ear file.
10. Install the application in the usual manner.

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 ConnectionWaitTimeout exception indicates that the application timed out trying to get a connection. The StaleConnection exception indicates that the connection is no longer valid.

Connection wait timeout:

The ConnectionWaitTimeout 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.

Later version data sources issue an SQL exception of the com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException subclass.

Example: Handling data access exception - ConnectionWaitTimeoutException (for the JDBC API):   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 the JDBC API:
```
 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 (com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException cwte) {
  //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.
 }
 catch (java.sql.SQLException sqle) {
  // 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 J2EE Connector Architecture): 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 J2EE 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) {
         }
      }
   }
}
```

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. It 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
  `StaleConnection` 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 `StaleConnection` 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 `StaleConnection` 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 `StaleConnection` 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 `StaleConnection` exception for the database server in use.

Recovering from stale connections

Recovering from stale connections is a joint effort between the application server run time and the application developer. From an application server perspective, the connection pool is purged based on its *PurgePolicy* setting.

Explicitly catching a `StaleConnection` exception is not required in an application. Because applications are already required to catch the `java.sql.SQL` exception, and the `StaleConnection` exception extends an SQL exception, a `StaleConnection` exception can result from any method that is declared to create an SQL exception, and is caught automatically in the general catch-block. However, explicitly catching a `StaleConnection` exception makes it possible for an application to recover from bad connections. When application code catches a `StaleConnection` exception, it should take explicit steps to handle the exception.

Example: Handling data access exception - StaleConnectionException: When an application receives a stale connection exception on a database operation, it indicates that the connection currently held is no longer valid. While it is possible to get a stale connection exception on any database operation, the most common time to see a stale connection exception issued is the first time that a connection is used, just after it 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.

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(com.ibm.websphere.ce.cm.StaleConnectionException
          sce)
  {
    //if a StaleConnectionException is caught
    // rollback and retry the action
    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;
    }
  } catch (java.sql.SQLException sqle) {
    //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(com.ibm.websphere.ce.cm.StaleConnectionException
sce) {
getSessionContext().setRollbackOnly();
throw new RetryableConnectionException();
}
catch(java.sql.SQLException sqle) {
//handle other database problem
}
finally {
21
//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
EJBException)
{
java.rmi.EJBException re = EJBException;
Throwable t = null;
while (true) {
t = re.getCause();
try { re = (java.rmi.EJBException)t; }
catch(ClassCastException cce) {
return (t instanceof
com.ibm.websphere.ce.cm.StaleConnectionException);
}
}
}
}
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(com.ibm.websphere.ce.cm.StaleConnectionException
sce) {
//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
}
// rollback the tran when method returns
getEntityContext().setRollbackOnly();
throw new java.rmi.EJBException("Exception occurred in
ejbStore", sce);
}
catch(java.sql.SQLException sqle) {
```

```
// handle some other problem
       }
   }
}
```

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 *StaleConnectionException* 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 a non-Remote exception. If this final exception is a stale connection exception, you find it and *true* is returned; otherwise, there is no stale connection exception in the list (because a stale connection exception can never be cast to a remote exception), 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.

StaleConnectionException on Linux systems: Linux systems have a semaphore problem causing the DB2 stale connection exception **SQL1224** error. This is related to the extension shared memory attachment.

To work around the problem, set the loopback for your database. For example, if your database is *WAS*, host name is *LHOST*, and database service port number is *50000*, issue the following commands from the DB2 command line window:

```
 db2 catalog TCPIP node RHOST remote LHOST server 50000
 db2 catalog WAS as WASAlias
 db2 uncatalog db WAS
 db2 catalog WASAlias as WAS at node RHOST
```

Verify this by issuing the following commands from the DB2 command line window:

```
db2 connect to WAS user xxx
passwd: xxx
```

Example: Developing servlet with user transaction:

```
//===================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002
//   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.  Note the lack
// of an IBM Extensions package import.  This is no longer required.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.transaction.*;

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
// StaleConnectionException
        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 (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

// 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 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 (com.ibm.websphere.ce.cm.StaleConnectionException sc) {

// This exception indicates that the connection to the database is no longer valid.
//Rollback 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("Stale Connection Exception during get connection or process SQL: " + 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.");
      }
    }
    catch (SQLException sq) {
      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 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;
   }
   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: Developing session bean with container managed transaction:

```
//==================START_PROLOG======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002
//   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.*;

/*****************************************************************************
* 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 com.ibm.ejs.cm.pool.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 (com.ibm.websphere.ce.cm.StaleConnectionException 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("Stale Connection Exception during get connection or process SQL: " + se.getMessage());

System.out.println("Rolling back transaction and throwing  RetryableConnectionException");

      mySessionCtx.setRollbackOnly();
      throw new RetryableConnectionException(se.toString());
      }
      catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

// 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: " +
      cw.getMessage());
 throw cw;
      }
      catch (SQLException sq) {

//Throwing a remote exception will automatically roll back the container managed //transaction

      System.out.println("SQL Exception during get connection or process SQL: " +
    sq.getMessage());
   throw sq;
      }
      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 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");
 }
       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
//   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
//    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,
 com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException,
    WebSphereSamples.ConnPool.RetryableConnectionException;
}
//==================START_PROLOG======================================
//
//    5630-A23, 5630-A22,
//    (C) COPYRIGHT International Business Machines Corp. 2002
//    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: Developing session bean with bean managed transaction:

```
//===================START_PROLOG======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002
//   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.*;

/*******************************************************************************
 * 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 StaleConnectionException 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
    //StaleConnectionException
    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) + ", " +
                  }
//Set error to false, as all database operations are successfully completed
                  error = false;
                }
                catch (com.ibm.websphere.ce.cm.StaleConnectionException 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("Stale Connection Exception during get connection or process SQL: " +
            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: " + se.toString());
        }
        else {
            System.out.println("Retrying transaction, retryCount = " +
                    retryCount);
        }
    }
    catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

// 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: " +
        cw.getMessage());
        userTran.rollback();
        throw new EJBException("Transaction failure: " + cw.getMessage());
    }
    catch (SQLException sq) {
// This catch handles all other SQL Exceptions
    System.out.println("SQL Exception during get connection or process SQL: " +
     sq.getMessage());
        userTran.rollback();
        throw new EJBException("Transaction failure: " + sq.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
//   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
//   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: Developing entity bean with bean managed persistence (container managed transaction):

```
//==================START_PROLOG======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2005
//   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.*;

/**
 * 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 (com.ibm.websphere.ce.cm.StaleConnectionException 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("Stale Connection Exception during get connection or
process SQL: " +  se.getMessage());
      throw new CreateException(se.getMessage());
   }
   catch (SQLException sq) {
      System.out.println("SQL Exception during get connection or process SQL: " +
                       sq.getMessage());
   throw new CreateException(sq.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 (com.ibm.websphere.ce.cm.StaleConnectionException 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("Stale Connection Exception during get connection or
process SQL: " + se.getMessage());
      throw new EJBException(se.getMessage());
   }
   catch (SQLException sq) {
      System.out.println("SQL Exception during get connection or process SQL: " +
                         sq.getMessage());
      throw new EJBException(sq.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());
            }
        }
    }
}
 try {
        con = ds.getConnection();
        ps = con.prepareStatement(sql);
        ps.setString(1, empNo);
        if (ps.executeUpdate() != 1){
            throw new EJBException("Cannot remove employee: " + empNo);
        }
    }
    catch (com.ibm.websphere.ce.cm.StaleConnectionException 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("Stale Connection Exception during get connection or
process SQL: " + se.getMessage());
        throw new EJBException(se.getMessage());
    }
    catch (SQLException sq) {
        System.out.println("SQL Exception during get connection or process SQL: " +
                            sq.getMessage());
        throw new EJBException(sq.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());
            }
        }
    }
}
}
    catch (com.ibm.websphere.ce.cm.StaleConnectionException 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
```

```java
    // can retry the transaction if desired.

System.out.println("Stale Connection Exception during get connection or
process SQL: " + se.getMessage());
        throw new EJBException(se.getMessage());
    }
    catch (SQLException sq) {

        System.out.println("SQL Exception during get connection or process SQL: " +
                          sq.getMessage());
        throw new EJBException(sq.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 (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 (com.ibm.websphere.ce.cm.StaleConnectionException 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("Stale Connection Exception during get connection or
process SQL: " + se.getMessage());
          throw new FinderException(se.getMessage());
```

```
            }
        catch (SQLException sq) {
System.out.println("SQL Exception during get connection or process SQL: " +
                                sq.getMessage());
            throw new FinderException(sq.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
//   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 Entity Bean
 */
public interface EmployeeBMPHome extends javax.ejb.EJBHome {

/**
 *
 * @return WebSphereSamples.ConnPool.EmployeeBMP
 * @param empNo java.lang.String
 * @param firstName java.lang.String
 * @param lastName java.lang.String
 * @param middleInit java.lang.String
 * @param edLevel int
 */
WebSphereSamples.ConnPool.EmployeeBMP create(java.lang.String empNo, java.lang.String firstName,
 java.lang.String lastName, java.lang.String middleInit, int edLevel) throws
 javax.ejb.CreateException, java.rmi.RemoteException;
/**
 * findByPrimaryKey method comment
 * @return WebSphereSamples.ConnPool.EmployeeBMP
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception java.rmi.RemoteException
 * @exception javax.ejb.FinderException
 */
WebSphereSamples.ConnPool.EmployeeBMP
  findByPrimaryKey(WebSphereSamples.ConnPool.EmployeeBMPKey key)
    throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

```
//==================START_PROLOG======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002
//   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
//   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 - error mapping in DataStoreHelper: Error mapping is necessary because various database vendors can provide differing SQL errors and codes that might mean the same things. 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.

To provide portability for applications, WebSphere Application Server provides a *DataStoreHelper* interface to enable mapping of these codes to the WebSphere Application Server exceptions. 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);
    ...
  }
}
```

Database deadlock and foreign key conflicts:

This article describes troubleshooting issues with database deadlock and foreign key conflicts.

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.

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.

This topic assumes that you have created an enterprise application containing an EJB module that must transact with a database.

A data access application uses resources, such as data sources or connection factories, to connect with a database. During application assembly you perform activities that enable the application to use these resources. The process typically requires an assembly tool such as the Application Server Toolkit (AST) or Rational Application Developer.

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 "The benefits of using resource references" on page 701 topic.

2. Start an assembly tool.

3. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that **J2EE** 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 703 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
      WebSphere Application Server 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. For an overview of the mapping options in the Application Server Toolkit, consult Approaches for mapping enterprise beans to database tables.

Files for the updated application are shown in the Project Explorer view.

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 run-time environment. This capability eliminates the necessity to alter application code when you change the resource run-time configurations.

This topic guides you through updating the resource references of an enterprise application that you assembled previously. The topic Chapter 7, "Assembling applications," on page 1345 details the assembly procedure.

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.

This topic describes how to update the resource references of an enterprise application using an assembly tool.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE 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 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.

Files for the updated module are shown in the Project Explorer view.

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 201.

Resource adapter archive file

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 WebSphere Application Server.

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*.

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).

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.

In the Application Server Toolkit (AST) and Rational Application Developer assembly tools, RAR files are called *connectors* and assembled resource adapters are called *connector modules*.

A *connector* is a J2EE component that provides access to Enterprise Information Systems (EIS), and must comply with the J2EE 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. This topic points you to AST documentation. The Application Server Toolkit information center accompanies this WebSphere Application Server information center.
1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that **J2EE** 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.

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**.

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.

For more information, see the online help for the assembly tool. Similar information is in the **Application Server Toolkit** information center available with this information center. Click **Application Server Toolkit > J2EE applications > Working with projects > Creating a connector project**.

After assembling the connector project, deploy the module or its application onto a server. When deploying the RAR file, WebSphere Application Server looks first for the connector module manifest (`manifest.mf`) in the _connectorModule.jar file and loads the manifest from the _connectorModule.jar file. If the class path entry is in the manifest from the _connectorModule.jar file, then the RAR uses that class path. 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 J2EE 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.

To use the connection management infrastructure in WebSphere Application Server Version 6.x, you must package your application as a J2EE 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.

In WebSphere Application Server Version 6.x, data sources are intended for use within J2EE applications and designed to operate within the EJB and Web containers.

Converting a 2.2 Web module to a 2.3 Web module:
Use the following steps to migrate each of your Web modules.

1. Open an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.
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.

Converting a 1.1 EJB module to a 2.1 EJB module (or later):

Use the following steps to migrate each of your EJB modules.

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 Migrating enterprise bean code to the supported specification, the documentation for Rational Application Developer, or the documentation for WebSphere Studio Application Developer Integration Edition.

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.

Installing the Application on WebSphere Application Server:

1. Install the application following the instructions in the Installing a new application article in the information center, 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 Creating a JDBC provider and data source article in the information center.

Connection considerations when migrating servlets, JavaServer Pages, or enterprise session beans: 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.

Explanation of the underlying problem

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 1.4 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.

Examples of the connection behavior change

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.

Alternatives to the connection behavior change

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.

You can only deploy application code that is assembled into the appropriate modules. The topic "Assembling data access applications" on page 790 provides guidelines for this process.

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 "J2EE Connector Architecture resource adapters" on page 662 and "Accessing data using J2EE Connector Architecture connectors" on page 713.

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 the Application Server Toolkit (AST), consult the that product information center about the mapping wizard. To learn about all of your assembly tool options, consult the "Assembly tools" on page 23 article in this information center.

c. Check Minimum required properties for vendor-specific data sources 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 **Application Server Toolkit** information center.

3. Install your application onto the application server. Consult "Installing application files" on page 1364. 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 703 topic.

   Tip: After deployment, you can use the WebSphere Application Server administrative console to alter resource bindings. Click **Applications** > **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 **1.x CMP bean data sources** or **2.x CMP bean data sources**. 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.

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** > **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.

1.x CMP bean data sources

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** > **Enterprise Applications** > *application_name* > **1.x CMP bean data sources**.

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 J2EE Connector (J2C) authentication data list, you must input them into the list after saving your settings on this page. Read the "Managing J2EE Connector Architecture authentication data entries" on page 936 information center topic for instruction.
  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.

Table column heading descriptions:

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.

1.x entity bean data sources

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** > **Enterprise Applications** > *application_name* > **1.x entity bean data sources**.

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 J2EE Connector (J2C) authentication data list, you must input them into the list after saving your settings on this page. Read the "Managing J2EE Connector Architecture authentication data entries" on page 936 information center topic for instruction.
  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.

Table column heading descriptions:

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.

2.x CMP bean data sources

Use this page to designate how the container-managed persistence (CMP) 2.x beans of an application map to data sources that are available to the application.

To view this administrative console page, click **Applications** > **Enterprise Applications** > *application_name* > **2.x CMP bean data sources**.

Guidelines for using this administrative console page:
- The page displays a table that depicts the 2.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. 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 2.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 define data source security:
  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. 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. Click **Apply**.

  3. To modify the authorization method of a data source with container-managed authorization, you have three options: None, Default, or Custom login configuration. The reconfiguring process differs slightly for each option:

     – If you select **None**:
       a. Determine which data source configurations to designate with no authentication method.
       b. Select the appropriate table rows.
       c. Select **None** from the list of authentication method options that precede the table.
       d. Click **Apply**.

     – If you select **Default**:
       a. Determine which data source configurations to designate with the WebSphere Application Server DefaultPrincipalMapping login configuration. You must apply this option to each data source individually if you want to designate different authentication data aliases. See the ″J2EE Connector security″ information center topic for more information on the default mapping configuration.
       b. Select the appropriate table rows.
       c. Select **Use default method** from the list of authentication method options that precede the table.
       d. Select an authentication data entry or alias from the list.
       e. Click **Apply**.

     – If you select **Custom login configuration**:
       a. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the ″J2EE Connector security″ information center topic for more information on custom JAAS login configurations.
       b. Select the appropriate table row.
       c. Select **Use custom login configuration** from the list of authentication method options that precede the table.
       d. Select an application login configuration from the list.
       e. Click **Apply**.
       f. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

- Repeat all of the previous steps as necessary.

- Click **OK** to save your work. 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.

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

Resource authorization:

The authorization type and the authentication method for securing the data source.

2.x entity bean data sources

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 page, click **Applications** > **Enterprise Applications** > *application_name* > **2.x entity bean data sources**.

Guidelines for using this administrative console page:
- The page 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.
- 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 2.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 define data source security:
  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. 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. Click **Apply**.

3. To modify the authorization method of a data source with container-managed authorization, you have three options: None, Default, or Custom login configuration. The reconfiguring process differs slightly for each option:

   – If you select **None**:

     a. Determine which data source configurations to designate with no authentication method.

     b. Select the appropriate table rows.

     c. Select **None** from the list of authentication method options that precede the table.

     d. Click **Apply**.

   – If you select **Default**:

     a. Determine which data source configurations to designate with the WebSphere Application Server DefaultPrincipalMapping login configuration. You must apply this option to each data source individually if you want to designate different authentication data aliases. See the ″J2EE Connector security″ information center topic for more information on the default mapping configuration.

     b. Select the appropriate table rows.

     c. Select **Use default method** from the list of authentication method options that precede the table.

     d. Select an authentication data entry or alias from the list.

     e. Click **Apply**.

   – If you select **Custom login configuration**:

     a. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the ″J2EE Connector security″ information center topic for more information on custom JAAS login configurations.

     b. Select the appropriate table row.

     c. Select **Use custom login configuration** from the list of authentication method options that precede the table.

     d. Select an application login configuration from the list.

     e. Click **Apply**.

     f. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

- Repeat all of the previous steps as necessary.

- Click **OK** to save your work. You now return to the general configuration page for your enterprise application.

Table column heading descriptions:

Select:

Select the check boxes of the rows you want to edit.

EJB Module:

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:

The Java Naming and Directory Interface (JNDI) name of the default data source for the EJB module.

Data type   String

Resource authorization:

The authorization type and the authentication method for securing the data source.

Messaging resources

Using asynchronous messaging

These topics describe how enterprise applications can use asynchronous messaging as a method of communication based on the Java Message Service (JMS). With the support provided by WebSphere Application Server, applications can make use of JMS resources and message-driven beans.

WebSphere Application Server support for JMS is provided by one or more *JMS providers*, and associated services and resources, that you configure for use by enterprise applications. You can deploy EJB 2.1 applications that use the JMS 1.1 interfaces and EJB 2.0 applications that use the JMS 1.0.2 interfaces.

You can use the WebSphere administrative console to administer the WebSphere Application Server support for asynchronous messaging. For example, you can configure messaging providers and their resources, and can control the activity of messaging services.

For more information about implementing WebSphere enterprise applications that use asynchronous messaging, see the following topics:
* Learning about messaging with WebSphere
* Installing a messaging provider
* Using the default messaging provider
* Maintaining Version 5 default messaging resources
* Using the JMS resources provided by WebSphere MQ
* Using JMS resources of a generic provider
* Administering support for message-driven beans
* "Programming to use asynchronous messaging" on page 823
* Troubleshooting WebSphere messaging

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.

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) and Java Connector Architecture (JCA) programming interfaces. These interfaces provide a common way for Java programs (clients and J2EE applications) to create, send, receive, and read asynchronous requests, as messages. For additional information about messaging resources, see: Messaging resources.
* JMS providers
* Styles of messaging in applications
* JMS interfaces - explicit polling for messages
* Message-driven beans - automatic message retrieval
* Configuring JMS resources for the WebSphere MQ messaging provider
* Components of message-driven bean support

- WebSphere Application Server cloning and WebSphere MQ clustering
- Security considerations for asynchronous messaging

JMS providers

This topic provides an overview of the support for JMS providers by WebSphere Application Server.

Overview

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.

WebSphere Application Server supports JMS messaging using the following:
- Service integration default messaging provider
- WebSphere MQ JMS provider
- Version 5 default provider
- Generic JMS provider

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.

Service integration default provider

The service integration technologies of 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 MQ JMS provider

WebSphere Application Server also includes support for the WebSphere MQ JMS provider. This is provided for use with supported versions of WebSphere MQ.

Version 5 default provider

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.

The V5 default messaging provider is the version 5 embedded WebSphere MQ provider. It is designed for use with applications that still use version 5 resources to communicate with version 5 nodes in mixed version cells that use embedded messaging.

Generic JMS provider

WebSphere Application Server also includes support for the generic JMS provider. This is provided for use with any third party messaging system. If you want to use message-driven beans, the messaging system must support ASF.

For additional information about connecting to WebSphere MQ, see Ways of interoperating with WebSphere MQ

For more information about connecting to WebSphere MQ, see Learning about WebSphere MQ server

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 / 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:
* An overview of asynchronous messaging with JMS
* An overview of asynchronous messaging with message-driven beans

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 J2EE 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.
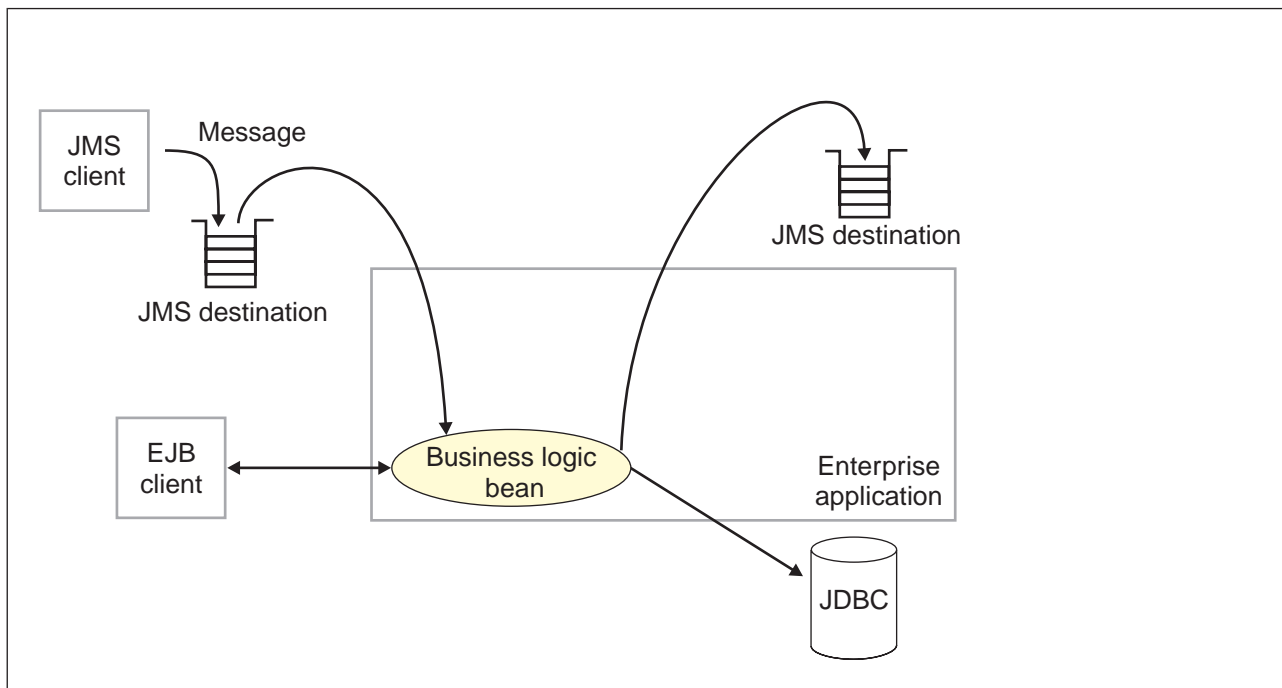
Figure 7. Asynchronous messaging using JMS. This figure shows an enterprise application polling a JMS destination to retrieve an incoming message, which it processes with a business logic bean. The business logic bean uses standard JMS calls to process the message; for example, to extract data or to 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. This level of functionality for asynchronous messaging is called *bean-managed messaging*, and gives an enterprise bean complete control over the messaging infrastructure; for example, for connection and session pool management. The application server has no role in bean-managed messaging.

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 Messaging with message-driven beans.

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 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.

A message driven bean can be registered with the EJB timer service for time-based event notifications if it implements the javax.ejb.TimedObject interface in addition to the message listener interface.

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 EJB component security.

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 1.5-compliant 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).
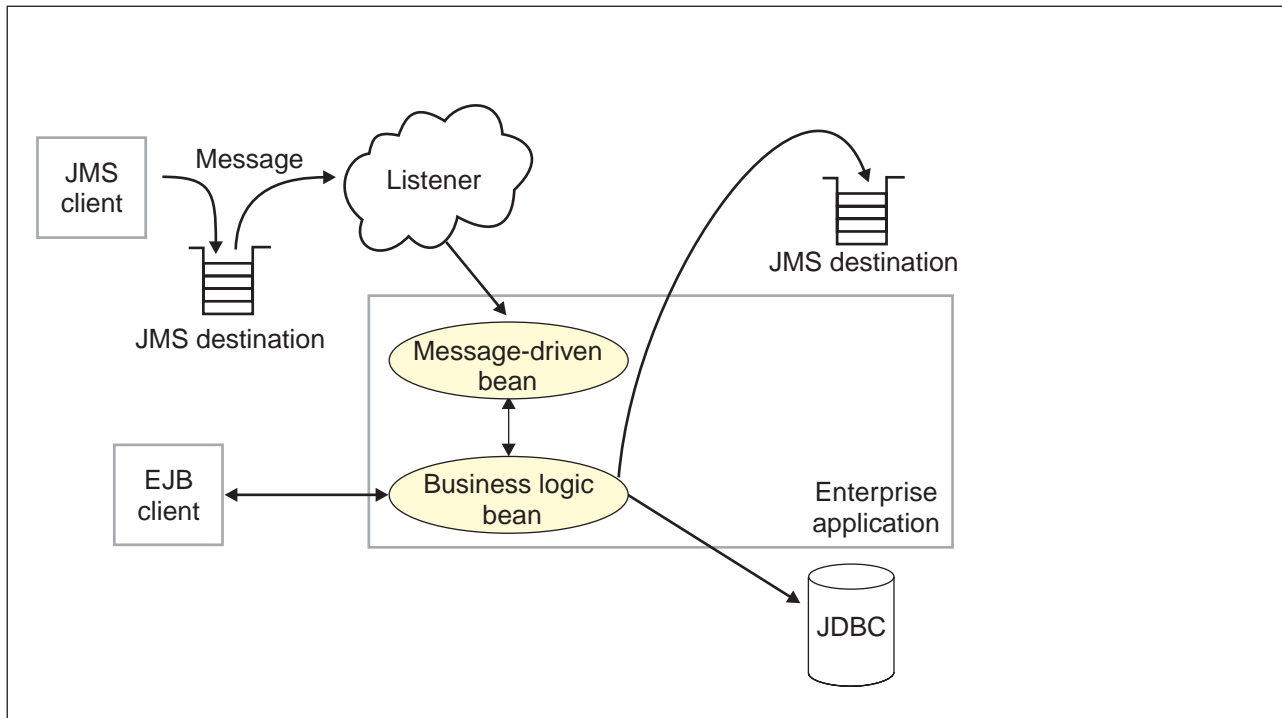
Figure 8. Messaging with message-driven beans. This figure shows an incoming message being passed automatically to the onMessage() method of a message-driven bean that is deployed as a listener for the destination. The message-driven bean processes the message, in this case passing the message on to 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 as listeners on a Java Connector Architecture (JCA) 1.5 resource adapter.

Components for a JCA resource adapter

To handle non-JMS requests inbound to WebSphere Application Server from enterprise information systems, message-driven beans use a Java Connector Architecture (JCA) 1.5 *resource adapter* written by a third party for that purpose.

With a Java Connector Architecture (JCA) 1.5 resource adapter, a message-driven bean acts as a listener on a specific endpoint. In the JCA 1.5 specification, such message-driven beans are commonly called *message endpoints* or simply *endpoints*.

Each application configuring one or more message-driven beans must specify the resource adapter that sends messages to the endpoint. To specify the resource adapter, you configure the message-driven bean to use an activation specification that has been configured by the administrator for the resource adapter.
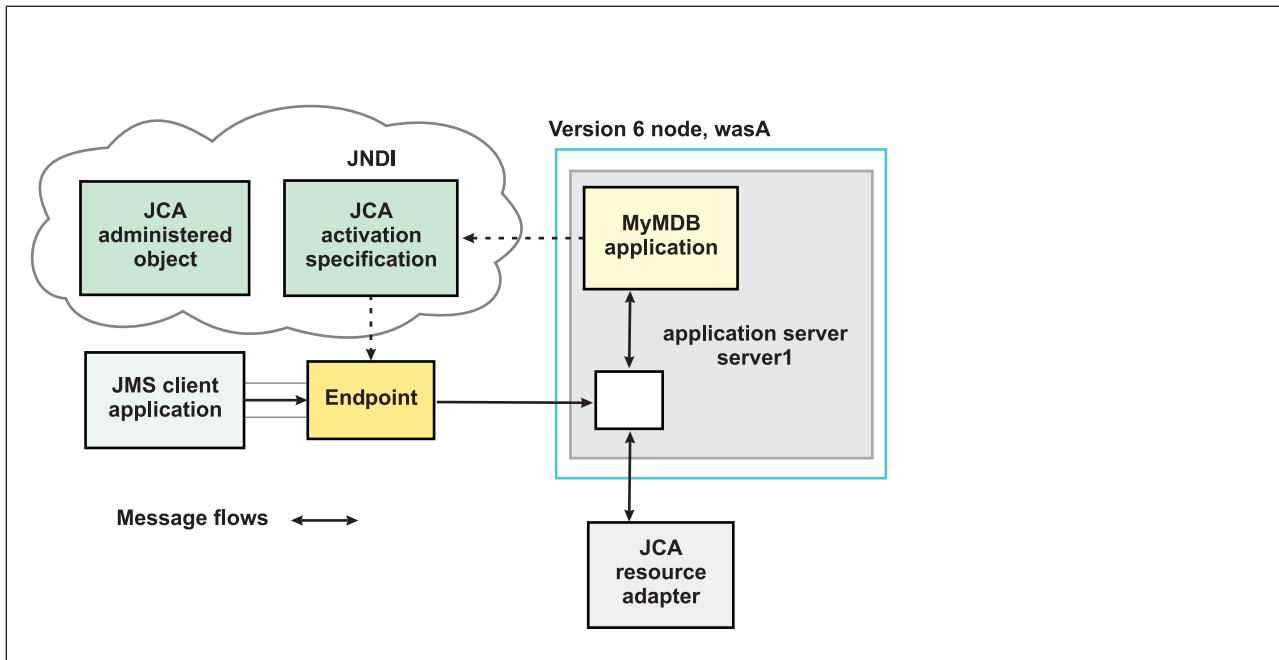
Figure 9. Message-driven bean components for a JCA resource adapter. This figure shows the main components of WebSphere support for message-driven beans for use with an external JCA resource adapter.

The administrator creates a *J2C activation specification* for the appropriate resource adapter to provide information to the deployer about the configuration properties of an endpoint instance (message-driven bean) related to the processing of the inbound messages. Properties specified on an activation specification can be overridden by appropriately named activation-configuration properties in the deployment descriptor of an associated EJB 2.1 message-driven bean.

When a deployed message-driven bean is installed, it is associated with an activation specification for an endpoint. When a message arrives on the endpoint, the message is passed to a new instance of a message-driven bean for processing.

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 programming interfaces specific to a messaging style). Administered objects can also be used to perform transformations on an asynchronously-received message in a way that is specific to a message provider. Administered objects can be accessed by a component by using either a message destination reference (preferred) or a resource environment reference.

JMS components used with a JCA messaging provider

Message-driven beans that implement the javax.jms.MessageListener interface can be used for JMS messaging. For JMS messaging, message-driven beans can use a JCA-based messaging provider such as the SIB JMS Resource Adapter (which is the default messaging provider listed under JMS providers) that is part of WebSphere Application Server and configure message-driven beans to use a JCA activation specification.
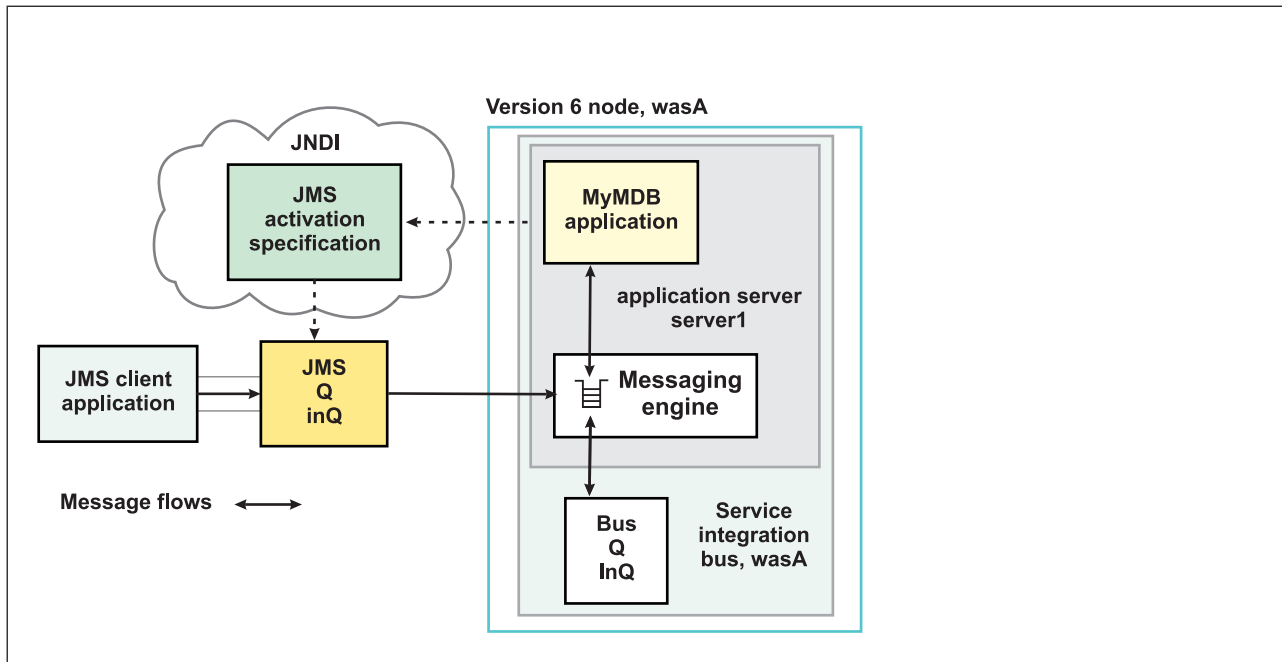
Figure 10. Message-driven bean components for the default messaging provider. This figure shows the main components of WebSphere support for message-driven beans for use with the default messaging provider.

With the SIB JMS Resource Adapter, a message-driven bean acts as a listener on a specific *JMS destination*.

The administrator creates a *JMS activation specification* (which, the WebSphere administrative console shows on the panel **Resources** → **JMS providers** → **Default messaging** → **JMS activation specifications**) to provide information to the deployer about the configuration properties of a message-driven bean related to the processing of the inbound messages. WebSphere provides additional support for JCA activation specifications that are JMS-based and shows the JMS-specific panel rather than the generic JCA activation specification panel. For example, a JMS activation specification specifies the name of the service integration bus to connect to, and includes 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. Properties specified on an activation specification can be overridden by appropriately named activation-configuration properties in the deployment descriptor of an associated EJB 2.1 message-driven bean.

The administrator also creates other administered objects that configure the JMS destination and the associated resources of a service integration bus that are used to implement messaging with that JMS destination.

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 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.

Destination transaction handling

If transaction handling is specified for a destination, the message-driven bean starts a global transaction *before* it reads any incoming message from that destination. When the message-driven bean processing has finished, it commits or rolls back the transaction (using JTA transaction control).

All messages retrieved from a specific destination have the same transactional behavior.

If messages are queued to be sent within a global transaction they are sent when the transaction is committed. If the processing of a message causes the transaction to be rolled back, then the message that caused the bean instance to be invoked is left on the JMS destination.

Inbound resource adapter transaction handling

A message-driven bean can be set up to either have Bean or Container transaction handling. The resource adapter owner must tell the message-driven bean developer how to set up the message-driven bean 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 WebSphere Application Server horizontal cloning with WebSphere MQ clustered queues.

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 an WebSphere MQ server. If a host is only used to distribute messages, it only contains an 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 "WebSphere Application Server cloning and WebSphere MQ clustering."
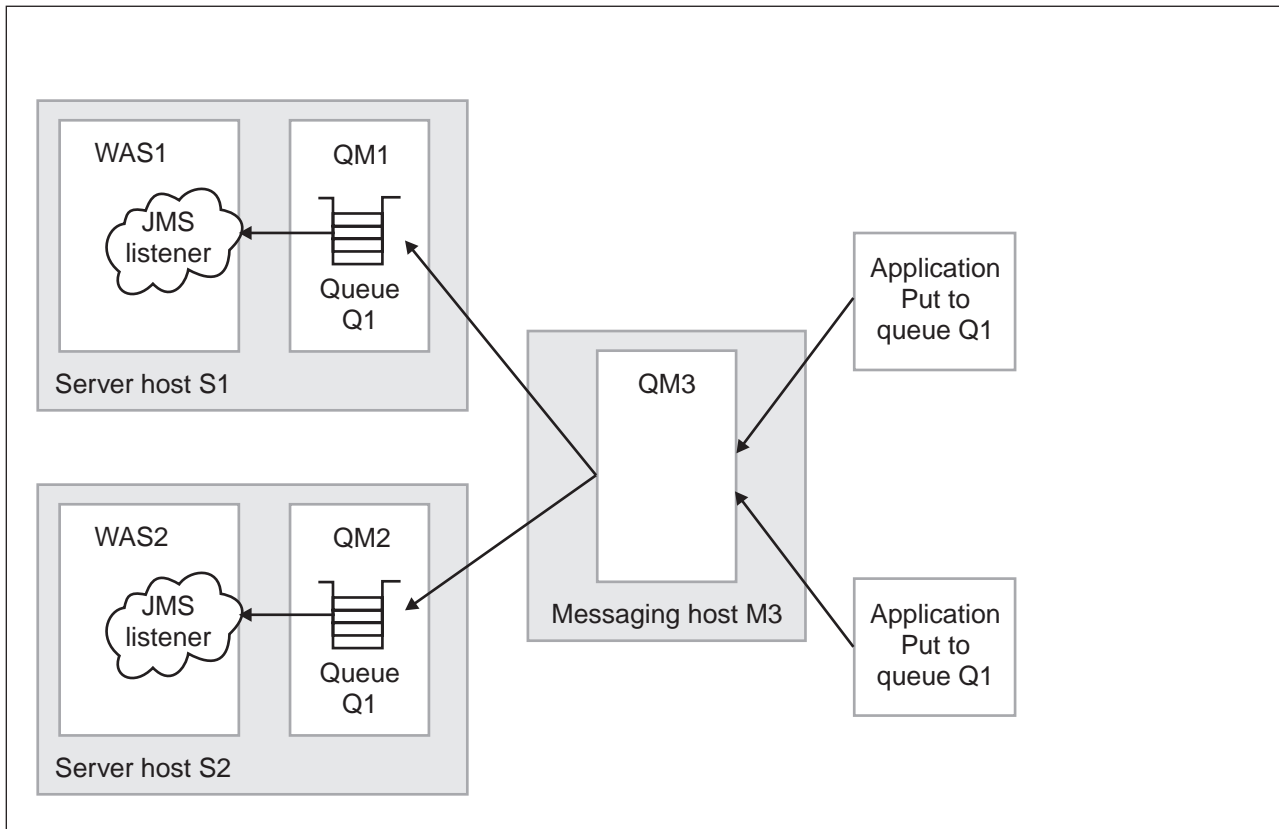
Figure 11. WebSphere Application Server horizontal cloning with WebSphere MQ clustered queues. This figure shows two WebSphere Application Server hosts, with horizontal clustering, and a messaging host used to distribute messages for WebSphere MQ server clustering. For more information, see the text that accompanies this figure.

The scenario shown in "WebSphere Application Server cloning and WebSphere MQ clustering" on page 813 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 security is enabled.

The user ID and password do not need to be provided by the application. If authentication is successful, then the JMS connection is created; if the authentication fails then the connection request is ended.

When WebSphere Application Server security is enabled, JMS connections made to the JMS provider are authenticated, and access to JMS resources owned by the JMS provider are controlled by access authorizations. Also, all 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. 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.

Restriction:

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.

2. If you want to use Bindings transport mode for JMS connections to WebSphere MQ, you set the property **Transport type**=BINDINGS on the WebSphere MQ Queue Connection Factory. You must also choose one of the following options:

- To use security credentials, ensure that the user specified is the currently logged on user for the WebSphere Application Server process. If the user specified is not the current logged on user for the WebSphere Application Server process, then the WebSphere MQ JMS Bindings authentication throws the error `MQJMS2013 invalid security authentication supplied for MQQueueManager` error.

- Do not specify security credentials. On the WebSphere MQ Connection Factory, ensure that both the **Component-managed Authentication Alias** and the **Container-managed Authentication Alias** properties are not set.

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 permissions.

Messaging: Resources for learning

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

- Sun's Java Message Service (JMS) specification documentation.

  Provides details about the Java Message Service (JMS).

- Sun's J2EE Connector Architecture specification (http://java.sun.com/j2ee/connector/download.html).

  Provides details about inbound messaging processing using the J2EE Connector architecture.

- J2EE specification

  Provides details about the J2EE specification, including messaging considerations.

- WebSphere MQ Using Java.

  Provides information about using JMS with WebSphere MQ as a messaging provider.

- http://www-3.ibm.com/software/ts/mqseries/library/manualsa/manuals/platspecific.html

  Provides WebSphere MQ messaging platform-specific books.

- WebSphere MQ Event Broker Web site at http://www-4.ibm.com/software/ts/mqseries/platforms/#eventb

  Provides books about WebSphere MQ Event Broker as a publish/subscribe messaging broker.

- WebSphere MQ Integrator Web site at http://www-4.ibm.com/software/ts/mqseries/platforms/#integrator

  Provides books about WebSphere MQ Integrator as a publish/subscribe messaging broker.

- IBM Publications Center

  This Web site provides a wide range of IBM publications, including publications about messaging products.

Installing and configuring a JMS provider

This topic describes the different ways that you can use JMS providers with WebSphere Application Server. A JMS provider enables use of the Java Message Service (JMS) and other message resources in 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. If you want to use message-driven beans, the messaging system must support ASF.

For more information about the support for JMS providers, see "JMS providers" on page 805.

For more information about installing and using JMS providers, see the following topics:
- Installing the default messaging provider
- Using WebSphere MQ as a JMS provider. Installing WebSphere MQ as a JMS provider.

  Note:
  - You can install WebSphere MQ in addition to the default messaging provider. The preferred solution for publish/subscribe messaging with WebSphere MQ as a JMS provider is a full message broker such as WebSphere MQ Event Broker.
  - If you install WebSphere MQ as a JMS provider, you can use the WebSphere administrative console to administer the JMS resources provided by WebSphere MQ, such as queue connection factories. However, you cannot administer MQ security, which is administered through WebSphere MQ.

  For more information about scenarios and considerations for using WebSphere MQ with IBM WebSphere Application Server, see the White Papers and Red books provided by WebSphere MQ; for example, through the WebSphere MQ library Web page at http://www-3.ibm.com/software/ts/mqseries/library/
- Installing another JMS provider, which must conform to the JMS specification and, to use message-driven beans, support the ASF function. If you want to use a JMS provider other than the default messaging provider or WebSphere MQ, you should complete the following steps:
  1. Installing and configuring the JMS provider and its resources by using the tools and information provided with the product.
  2. Defining the JMS provider to WebSphere Application Server as a generic messaging provider.

  Note: You can use the WebSphere administrative console to administer JMS connection factories and destinations (within WebSphere Application Server) for a generic provider, but cannot administer the JMS provider or its resources outside of WebSphere Application Server.

Installing the default messaging provider

Use this task to install the default messaging provider of IBM WebSphere Application Server.

The default messaging provider is installed as a fully-integrated component of WebSphere Application Server, and needs no separate installation steps. However, ensure that there is enough space in the file systems where you want to store messaging data.

You can use the WebSphere administrative console to for the default messaging provider.

For more information about the default messaging provider, see Using the default messaging provider.

JMS providers collection

A JMS provider enables messaging based on the Java Message Service (JMS). It provides J2EE connection factories to create connections for JMS destinations.

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 The name of the JMS provider.

Description

     A description of the JMS provider.

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

Select the provider with which to create the {0}. The following providers support the selected resource type and are available at the selected scope. 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.

Activation specification collection

A JMS activation specification is associated with one or more message-driven beans and provides configuration necessary for them to receive messages.

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 Java Connector Architecture (JCA) 1.5 resource, to act as a listener on the default messaging provider.

General properties

Name   The name of the activation specification.

Provider
> This JMS resource is for the *Default messaging provider*, provided by service integration technologies as part of WebSphere Application Server.

Description
> A description of the activation specification.

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

Use this page to create connections to the associated JMS provider for JMS destinations.

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   The name of the connection factory.

Provider
> Specifies a JMS provider, which enables asynchronous messaging based on the Java Message Service (JMS). It provides J2EE connection factories to create connections for specific JMS queue or topic destinations. JMS provider administrative objects are used to manage JMS resources for the associated JMS provider.

Description
> A description of the connection factory.

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

A queue connection factory is used to create connections to the associated JMS provider of the JMS queue destinations, for point-to-point messaging.

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 The name of the queue connection factory.

Provider
> The type of JMS provider that provides the JMS resource. For example, for *Default messaging provider* resources are provided by service integration technologies as part of WebSphere Application Server; for *WebSphere MQ messaging provider* resources are provided by a separate WebSphere MQ installation.

Description
> A description of the queue connection factory.

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

A JMS queue is used as a destination for point-to-point messaging.

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 The name of the queue.

Provider
> Specifies a JMS provider, which enables asynchronous messaging based on the Java Message Service (JMS). It provides J2EE connection factories to create connections for specific JMS queue or topic destinations. JMS provider administrative objects are used to manage JMS resources for the associated JMS provider.

Description
>   A description of the queue.

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

A topic connection factory is used to create connections to the associated JMS provider of JMS topic destinations, for publish and subscribe messaging.

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   The name of the topic connection factory.

Provider
>   The type of JMS provider that provides the JMS resource. For example, for *Default messaging provider* resources are provided by service integration technologies as part of WebSphere Application Server; for *WebSphere MQ messaging provider* resources are provided by a separate WebSphere MQ installation.

Description
>   A description of the topic connection factory.

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

A JMS topic is used as a destination for publish/subscribe messaging.

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   The name of the topic.

Provider

> The type of JMS provider that provides the JMS resource. For example, for *Default messaging provider* resources are provided by service integration technologies as part of WebSphere Application Server; for *WebSphere MQ messaging provider* resources are provided by a separate WebSphere MQ installation.

Description

> A description of the topic.

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.

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.

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 837 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.

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.

This topic describes things 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

JMS Common Interfaces	Point-to-Point	Publish/Subscribe
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 more information about the properties of these JMS resources, see Configuring JMS provider resources.

- 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.

- Applications can cache JMS connections, sessions, and producers or consumers. Due to the pooling mentioned above this may not give as much of a performance improvement as you might expect.

  You *must not* cache session handles in stateless session beans that operate in transactions started by a client of the bean. Caching handles in this way causes the bean to be returned to the pool while the session is still involved in the transaction. Also, you should not cache non-durable subscribers due to the restriction mentioned above.

- 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 829.

- 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.

- 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.

The effect of transaction context 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

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
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
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 a J2EE application to use JMS:

Use this task to develop a J2EE application to use the JMS API directly for asynchronous messaging.

This topic gives an overview of the steps needed to develop a J2EE 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 J2EE application programming, which you should already be familiar with. For detailed information about these steps, and for examples of developing a J2EE application to use JMS, see the Java Message Service Documentation

Details of JMS resources that are used by J2EE applications are defined to WebSphere Application Server and bound into the JNDI namespace by the WebSphere administrative support.

To use JMS, any method of a J2EE application completes the following general steps:

1. Import JMS packages. A J2EE 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);
    }
}
```

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.

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 use JMS resources administered by the application server or administered by the client container regardless of whether the client application is running on the same machine as the server or remotely.

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:

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);
      }
    }
    ```

For information about running a client against a specific remote server: "Running application clients" on page 332.

Deploying a J2EE application to use JMS:

This topic describes how to deploy a J2EE application to use JMS.

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 a J2EE 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 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.

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.

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. **Optional:** If you want to handle messages at a scheduled date and time, or after a specified interval has elapsed, identify the schedule values and the business logic that you want to react to time-based messages. A message-driven bean can be registered with the EJB timer service for time-based event notifications. When a message arrives on the destination, a message-driven bean timer is initiated. When the timer expires, a message-driven bean is selected to process the ejbTimeout() method, which implements the business logic that is to process the message.

3. 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.

4. 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.

5. 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.

6. Topic durability considerations. 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.

7. Discarding of best-effort non-persistent messages with the default messaging provider. If you configure a JMS destination (queue or topic) to use the default messaging provider, you can configure the maximum reliability of messages on the bus destination to which the JMS destination is assigned.For non-transactional JMS message-driven beans and MessageListeners that use a JMS destination configured on the default messaging provider, best-effort nonpersistent messages are not recoverable. In this case, if a message is unlocked because the message-driven bean or MessageListener threw an exception, then the message is not redelivered or sent to the exception destination because it was deleted from the message store when it was passed to the listener. If you want better message reliability for non-transactional JMS message-driven beans and MessageListeners, you should configure a different option for the Maximum reliability property of the bus destination.

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.

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 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. This means the application server is given one attempt to process the message.

    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:

```
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 12. 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.

The result of this step is a message-driven bean that can be assembled into an EAR file for 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. All messages retrieved from a specific destination have the same transactional behavior. To enable the transactional behavior that you want, you must configure the JMS destination with the same transactional behavior as you configure for the message 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.

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 843.

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.

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.

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.

Transaction type

Whether the message-driven bean manages its own transactions or the container manages transactions on behalf of the bean. All messages retrieved from a specific destination have the same transactional behavior. To enable the transactional behavior that you want, you must configure the JMS destination with the same transactional behavior as you configure for the message-driven 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.

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 JCA 1.5-compliant resource adapter.

Message-driven beans can be configured as listeners on a Java Connector Architecture (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 such 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.

2. For each message-driven bean in the application, configure the J2C deployment attributes, as described in Configuring deployment attributes.

3. Use the WebSphere 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 JCA 1.5-compliant enterprise bean, to override the deployment properties defined within the application EAR file.

You can configure the deployment attributes of an application by using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This topic describes the use of the Application Server Toolkit (AST) 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 849.

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. See "Starting WebSphere Application Server Toolkit" in the Application Server Toolkit documentation for more 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 J2EE 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 843.

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 from the Application Server Toolkit (AST) or Rational Web Developer to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

Important

Important: Use **Run On Server** for unit testing only. The Application Server Toolkit (AST) or Rational Web Developer 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 on production servers.

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.

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 Providers** → **Default messaging** → **[Activation Specifications] JMS activation specification**
  2. 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.
  3. Set the **Authentication alias** property.
  4. Click **OK**
  5. Save your changes to the master configuration.
- For a message-driven bean listening on a destination (or endpoint) of another 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 of 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 J2EE Connector Architecture (JCA) Version 1.5 inbound resource adapters.

For installations that use resource adapters that implement the J2EE 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**.

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.

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 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.

To deploy an enterprise application to use EJB 2.0 message-driven beans with listener ports, complete the following steps:

1. Use the WebSphere 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 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.

You can configure the deployment attributes of an application by using an assembly tool such as the Application Server Toolkit (AST) or Rational Application Developer.

This topic describes the use of the Application Server Toolkit (AST) 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 J2EE 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. All messages retrieved from a specific destination have the same transactional behavior. To enable the transactional behavior that you want, you must configure the JMS destination with the same transactional behavior as you configure for the message 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 843.

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. See the Application Server Toolkit documentation for more information.

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 that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Application Server Toolkit (AST) or Rational Application Developer to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important: Use **Run On Server** for unit testing only. The Application Server Toolkit (AST) or Rational Application Developer 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 on production servers.

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 and domain-specific interfaces as provided for JMS 1.0.2 in WebSphere Application Server.

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 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

Common interfaces	point-point interfaces	publish/subscribe interfaces
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 MQCIH for CICS Bridge, or MQWIH for WebSphere MQ Workflow applications. For more information about how the JMS message structure is mapped onto an 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.

Mail, URLs, and other J2EE resources

Using mail

Using the JavaMail API, a code segment can be embedded in any Java 2 Enterprise Edition (J2EE) 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 J2EE 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});
```

J2EE 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 J2EE 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 J2EE 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 J2EE 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 a *soft link* 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 shipped with WebSphere 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>
   ```

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, WebSphere 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. Enable debugger for a mail session.

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 platform and protocol-independent framework for building Java-based mail client applications.

WebSphere Application Server supports the JavaMail API, Version 1.3, and the JavaBeans Activation Framework (JAF) Version 1.0. In WebSphere Application Server, the JavaMail API is supported in all Web application components, namely:

*   Servlets
*   JavaServer Pages (JSP) files
*   Enterprise beans
*   Application clients

The JavaMail APIs are generic for sending and reading mail. They require service providers, known in WebSphere 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. JavaMail applications can connect to an SMTP server and send mail through it by using this SMTP protocol provider.

In addition to service providers, the JavaMail API requires the Java 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 WebSphere Application Server. The API and related specifications are repackaged from Sun-licensed materials into the following file groupings:

*   `j2ee.jar` - Contains the JavaMail API and the JAF
*   `mail-impl.jar` - Contains the implementation of the JavaMail API
*   `activation-impl.jar` - Contains the implementation of the JAF

Mail providers and mail sessions

A JavaMail service provider is a driver that supports JavaMail interaction with mail servers using a particular mail protocol. WebSphere Application Server includes service providers, also known as *protocol providers*, for mail protocols including Simple Mail Transfer Protocol (SMTP), Internet Message Access Protocol (IMAP), and Post Office Protocol 3 (POP3).

A mail provider encapsulates a collection of protocol providers. For example, WebSphere Application Server has a built-in mail provider that encompasses the three protocol providers: SMTP, IMAP and POP3. These protocol providers are installed as the default and suffice for most applications.

If you have a particular application that requires custom protocol providers, you must first follow the steps outlined in the ″JavaMail API Design Specification, V1.2, Chapter 5 - The Mail Session″ to install your own protocol providers. This document outlines the process for the JavaMail 1.3 API as well as JavaMail 1.2. See the article, *Mail: Resources for learning*, for a link to the specification.

Mail sessions are represented by the `javax.mail.Session` class. A mail `Session` object authenticates users, and controls users' access to messaging systems.

To create platform-independent applications, use a resource factory reference to create a JavaMail session. A resource factory is an object that provides access to resources in the deployed environment of a program using the naming conventions defined by the Java Naming and Directory Interface (JNDI).

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 such configuration files can exist are `<user.home>` and `<java.home>/lib`. For example, if the JavaMail API needs to access a file named `mailcap` when sending a message, it first tries to access `<user.home>/.mailcap`. If that attempt fails, either due to lack of security permission or a nonexistent file, the API continues to try`<java.home> /lib/mailcap`. If that attempts also fails, it tries META-INF/mailcap in the class path, which actually leads to the configuration files contained in the `mail-impl.jar` and `activation-impl.jar` files. WebSphere Application Server uses the general-purpose JavaMail configuration files contained in the `mail-impl.jar` and `activation-impl.jar` files and does not put any mail configuration files in `<user.home>`and `<java.home>/lib`. File read permission for both the `mail-impl.jar` and `activation-impl.jar` files is granted to all applications to ensure proper functioning of the JavaMail API, as shown in the following segment of the `app.policy` file:

```
grant codeBase "file:${application}" {
  // The following are required by Java mail
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-impl.jar", "read";
};
```

JavaMail code attempts to access configuration files at `<user.home>`and `<java.home>/lib` causing `AccessControlExceptions` to be thrown, since there is no file read permission granted 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 JavaMail-related security exceptions reported in the system log, which might swamp harmful errors that you are looking for. If this situation is a problem, consider adding two more permission lines to the permission block above. This should eliminate most, if not all, JavaMail-related harmless security exceptions from the log file. The application permission block in the `app.policy` file now looks like:

```
grant codeBase "file:${application}" {
  // The following are required by Java mail
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-impl.jar", "read";
java.io.FilePermission "${java.home}${/}lib${/}.mailcap", "read";
  permission java.io.FilePermission "${user.home}${/}lib${/}.mailcap", "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.

Enabling debugger for a mail session

When you need to debug a JavaMail application, you can use the JavaMail debugging feature. Enabling the debugger triggers the JavaMail component of WebSphere Application Server to print the following data to the `stdout` 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.

The mail debugger functions on a per session basis. To enable the JavaMail debugging feature:

1. Open the administrative console.

2. Click **Resources**>**Mail Providers**>*mail_session*>**Mail Session**>*mail session*.

3. Click **Debug**. Debug is enabled for just that session.

4. Click **Apply** or **OK**.

The following example shows sample JavaMail debugging output:

```
DEBUG: not loading system providers in <java.home>/lib
DEBUG  not loading optional custom providers file: /META-INF/javamail.providers
DEBUG: successfully loaded default providers

DEBUG  Tables of loaded providers
DEBUG: Providers listed by Class Name:
{com.sun.mail.smtp.SMTPTransport=javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,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]}
DEBUG:  Providers Listed By Protocol:
{imap=javax.mail.Provider[STORE,imap,com.sun.mail.imap.IMAPStore,Sun Microsystems,
Inc], pop3=javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Sun
```

```
Microsystems, Inc], smtp=javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun
Microsystems, Inc]}
DEBUG: not loading optional address map file: /META-INF/javamail.address.map
*** In SessionFactory.getObjectInstance,
  The default SessionAuthenticator is based on:
   store_user = john_smith
   store_pw = abcdef
*** In SessionFactory.getObjectInstance, parameters in the new session:
    mail.store.protocol="imap"
    mail.transport.protocol="smtp"
    mail.imap.user="john_smith"
    mail.smtp.host="smtp.coldmail.com"
    mail.debug="true"
    ws.store.password="abcdef"
    mail.from="john_smith@coldmail.com"
    mail.smtp.class="com.sun.mail.smtp.SMTPTransport"
    mail.imap.class="com.sun.mail.imap.IMAPStore"
    mail.imap.host="coldmail.com"
DEBUG: mail.smtp.class property exists and points to com.sun.mail.smtp.SMTPTransport
DEBUG SMTP: useEhlo true, useAuth false
DEBUG: SMTPTransport trying to connect to host "smtp.coldmail.com", port 25

javax.mail.SendFailedException: Sending failed;
     nested exception is:
     javax.mail.MessagingException: Unknown SMTP host: smtp.coldmail.com;
      nested exception is
      java.net.UnknownHostException: smtp.coldmail.com
      at javax.mail.Transport.send0(Transport.java:219)
      at javax.mail.Transport.send(Transport.java:81)
      at ws.mailfvt.SendSaveTestCore.runAll(SendSaveTestCore.java:48)
      at testers.AnyTester.main(AnyTester.java:130)
```

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 JavaMail run-time, while the two lines headed by *** are printed by the WebSphere environment run-time.
- The first two lines say that some configuration files are skipped. At run-time the JavaMail component attempts to load a number of configuration files from different locations. All those files are not required. If a required file cannot be accessed, however, the JavaMail 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 three providers that are listed are the default protocol providers that come under the WebSphere built-in mail provider. They are the protocols SMTP, IMAP, and POP3, respectively. If you install special protocol providers (or, in JavaMail terminology, service providers) and these providers are used in the current mail session, you see them listed here with the default providers.
- The two lines headed by *** and the few lines below them are printed by WebSphere 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 property *mail.store.protocol* corresponds to the Protocol Name property in the Store Access section of the mail session configuration page.

  Note: Review the listed properties and values to verify that they correspond.
- The few lines above the exception stack show the JavaMail activities when sending a message. First, the JavaMail API recognizes that the transport protocol is set to SMTP and that the provider `com.sun.mail.smtp.SMTPTransport` exists. Next, the parameters used by SMTP, `useEhlo` and `useAuth`, are shown. Finally, the log shows the SMTP provider trying to connect to the mail server `smtp.coldmail.com`.

- Next is the exception stack. This data indicates that the specified mail server either does not exist or is not functioning.

Using URL resources within an application

Java 2 Enterprise Edition (J2EE) 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.

As in the case of other J2EE 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 J2EE 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-4.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.

To create a new URL configuration: 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.

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

Resource environment entries

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 864 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 866 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

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** > **Enterprise Applications** > *application_name* > **Resource environment references**.

Guidelines for using this administrative console page:

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

Security

Task overview: Securing resources

WebSphere Application Server supports the Java 2 Platform, Enterprise Edition (J2EE) model for creating, assembling, securing, and deploying applications. Applications are often created, assembled, and deployed in different phases and by different teams.

You can secure resources in a J2EE environment by following the required high-level steps. Consult the J2EE 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.
- 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."
- 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.

Your applications and production environment are secured.

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.

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.

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. Use the methods that are defined in the 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 Secure administration, applications, and infrastructure panel.

In WebSphere Application Server Version 6.1, 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, use DriverManager to connect to the database.

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. However, it is recommended that you use the new interface to implement your custom registry.

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.

   Attention: 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_rootBase*/plugins/ com.ibm.ws.runtime_6.1.0.jar and the *app_server_install_rootBase*/plugins/ com.ibm.ws.security.crypto_6.1.0/cryptosf.jar files in your class path. For example:

   ```
   %install_root%/java/bin/javac -classpath
   %install_root%app_server_install_rootBase/plugins/com.ibm.ws.runtime_6.1.0.jar;
   %install_root%app_server_install_rootBase/plugins/com.ibm.ws.security.crypto_6.1.0
                   /cryptosf.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.

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;
}
```

Implementing custom password encryption

WebSphere Application Server supports the use of custom password encryption.

An installation can implement any password encryption algorithm it chooses.

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`.

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.

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 2 Platform, Enterprise Edition (J2EE) 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 901.

2. Develop servlet filters for form login processing. For more information, see "Developing servlet filters for form login processing" on page 912.

3. Develop form login pages. For more information, see "Customizing Web application login" on page 909.

4. Develop enterprise bean component applications. For more information, see "Developing with programmatic APIs for EJB applications" on page 905.

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" on page 917.

6. Develop your own Java 2 security mapping module. For more information, see "Configuring programmatic logins for Java Authentication and Authorization Service" on page 921.

7. Develop custom user registries. For more information, see "Developing standalone custom registries" on page 868.

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.

Java 2 security is orthogonal to Java 2 Platform, Enterprise Edition (J2EE) 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 J2EE 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 881 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 > Secure administration, applications, and infrastructure**. The Secure administration, applications, and infrastructure 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.

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).

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.

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.

1. Start the **PolicyTool**.

   ▶ Windows ◀ For example, you can enter the following command at a Windows command prompt:

   `%{was.install.root}/java/jre/bin/policytool`

   The **PolicyTool** window opens. The tool looks for the `java.policy` file in your home directory. If it does not exist, an error message displays.

   Click **OK**.
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.

Important: 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.

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:

Use can configure Java 2 security policy files so that the required permission is granted for the specified WebSphere Application Server enterprise application.

Java 2 security uses several policy files to determine the permissions for each Java programs.

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. Six dynamic policy files are provided:

Policy file name	Description
app.policy	Contains default permissions for all of the enterprise applications in the cell.
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";
};
```

Important: 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 895.

   • 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 890.

   • 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 792. Otherwise, update the was.policy file. Refer to "Configuring the was.policy file" on page 888 and "Adding the was.policy file to applications" on page 893.

- If the permission is required by shared libraries, refer to "library.policy file permissions" on page 891.
- If the permission is required by SPI libraries, refer to "spi.policy file permissions" on page 890.

Tip: 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 886.

2. Restart the WebSphere Application Server enterprise application.

The required permission is granted for the specified WebSphere Application Server enterprise application.

If an WebSphere Application Server enterprise application in a cell requires permissions, some of the dynamic policy files need updating. The symptom of the missing permission is the java.security.AccessControlException exception. The missing permission is listed in the exception data.

```
java.security.AccessControlException: access denied
(java.io.FilePermission C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar read)
```

The previous two lines were split onto two lines because of the width of the page. Enter the permission on one line.

When a Java program receives this exception and adding this permission is justified, add a permission to an adequate dynamic policy file.

```
grant codeBase "file:user_client_installed_location" {
permission java.io.FilePermission
"C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar", "read";
};
```

The previous permission information lines were split onto more than one line because of their length. Enter the permission on one line.

To decide whether to add a permission, refer to the Access control exception topic.

app.policy file permissions:

Java 2 security uses several policy files to determine the granted permissions for each Java program.

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";
};
```

Important: 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.

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 profiles/*profile_name*/cells/*cell_name*/node/ *node_name*/app.policy c:/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 880. 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 profiles/*profile_name*/cells/*cell_name*/nodes/ *node_name*/app.policy c:/temp/test/was.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
${current.cell.name}	Current cell name
${current.node.name}	Current node name
${current.server.name}	Current server name

Tip: You cannot use the `${was.module.path}` in the `${application}` entry.

The `app.policy` file that is supplied by WebSphere Application Server resides at *app_server_root*/`profiles`/*profile_name*/`config/cells`/*cell_name*/`nodes`/*node_name*/`app.policy`, which contains the following default permissions:

Attention: 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:${application}" {
  // The following are required by Java mail
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-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. 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)`.

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 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 878. 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 Secure administration, applications, and infrastructure 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 c:/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 c:/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.

Important: 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 enable the **Enforce Java 2 Security** option on the Global security settings page 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).

Important: 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 enable the **Use Java 2 security to restrict application access to local resources** option on the Secure administration, applications, and infrastructure 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.

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:${application}	file:${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.

Important: 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,

```
"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

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.

Tip: 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 880.

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 893.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.

a. Modify the `was.policy` file with the Policy Tool. For more information, see "Using PolicyTool to edit policy files" on page 880.

The updated `was.policy` file is applied to the application after the application restarts.

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:

```
java.security.AccessControlException: access denied (java.io.FilePermission
app_server_root/lib/mail-impl.jar read)
```

The previous example was split onto several lines for illustrative purposes only.

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 "app_server_root/lib/mail-impl.jar", "read"; };
```

The previous example was split onto several lines for illustrative purposes only.

To determine whether to add a permission, see Access control exception.

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.

Important: 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 880.

To check in the policy file, enter the following command using information from your environment:

$AdminConfig checkin profiles/*profile_name*/cells/*cell_name*/nodes/*node_name*/spi.policy c:/temp/test/spi.policy $obj

The updated `spi.policy` is applied to the Service Provider Interface (SPI) libraries after the Java process is restarted.

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.

Important: 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:

wsadmin> set obj [$AdminConfig extract cells/*cell_name*/nodes/
node_name/library.policy c:/temp/test/library.policy]

The previous two lines were split onto two lines for illustrative purposes only.

Edit the extracted `library.policy` file with the Policy Tool. For more information, see "Using PolicyTool to edit policy files" on page 880.

To check in the policy file, use a command prompt to enter the following command using the appropriate variable values for your environment:

wsadmin> $AdminConfig checkin cells/*cell_name*/nodes/*node_name*/library.policy
c:/temp/test/library.policy $obj

An updated `library.policy` is applied to shared libraries after the servers restart.

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.

Windows For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
app_server_root/lib/mail-impl.jar read)
```

The previous lines are split into two lines for illustrative purposes only. The *app_server_root* variable represents your installation directory.

When a Java program receives this exception and adding this permission is justified, add a permission to the `library.policy` file.

▶ Windows   For example:

```
grant { permission java.io.FilePermission "app_server_root/lib/mail-impl.jar", "read"; };
```

The previous lines are split into two lines for illustrative purposes only. The *app_server_root* variable represents your installation directory.

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.

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 is located in the *profile_root*/config/cells/*cell_name*/nodes/*node_name* directory. The contents of the `app.policy` file are presented in the following example:

Attention: In the following code sample, the two permissions that are required by JavaMail are split onto two lines for illustrative purposes only.

```
// 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${/}mail-impl.jar","read";
   permission java.io.FilePermission "
       ${was.install.root}${/}lib${/}activation-impl.jar","read"; };
   // The following permissions apply to all utility .jar files  (other
   //  than enterprise beans JAR files) in the application.
grant codeBase "file:${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.

The following example illustrates adding extra permissions for an application in the `was.policy` file:

Attention: In the following code sample, the permission for the EJB module was split onto two lines for illustrative purposes only.

```
// 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 880.

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.

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**.

An application EAR file is now ready to run when Java 2 security is enabled.

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
app_server_root/lib/mail-impl.jar read)
```

The previous two lines are one continuous line for illustration purposes only.

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:${application}" {
  permission java.io.FilePermission
 "app_server_root/lib/mail-impl.jar", "read";
  };
```

Lines are split in this example for illustration purposes only.

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.

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 881.
   - If the permission is required only by applets and client containers, update the client.policy file. Refer to "client.policy file permissions" on page 899.
   - 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 897.
   - 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" on page 896.

2. Stop and restart WebSphere Application Server.

The required permission is granted for all of the Java programs that run with the restarted JVM.

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. 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, modify the `java.policy` file with the policy tool. For more information, see "Using PolicyTool to edit policy files" on page 880. 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. Edit these policy files carefully. 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.

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.

See Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server.

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 880 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.

When you do need to modify the `server.policy` file, locate this file at: *install_root*/properties/ `server.policy`. This file contains these default permissions:

```
// Allow to use ibm jdk extensions
grant codeBase "file:${was.install.root}/java/ext/-" {
  permission java.security.AllPermission;
};

// Allow to use ibm tools
grant codeBase "file:${was.install.root}/java/tools/ibmtools.jar" {
  permission java.security.AllPermission;
};

// Allow to use sun tools
grant codeBase "file:/QIBM/ProdData/Java400/jdk14/lib/tools.jar" {
  permission java.security.AllPermission;
};

// Allow to use sun tools (V5R2M0 codebase)
grant codeBase "file:/QIBM/ProdData/OS400/Java400/jdk/lib/tools.jar" {
  permission java.security.AllPermission;
};

// WebSphere system classes
grant codeBase "file:${was.install.root}/plugins/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/lib/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/classes/-" {
  permission java.security.AllPermission;
};

// Allow the WebSphere deploy tool all permissions
grant codeBase "file:${was.install.root}/deploytool/-" {
  permission java.security.AllPermission;
};

// Allow the WebSphere deploy tool all permissions
grant codeBase "file:${was.install.root}/optionalLibraries/-" {
  permission java.security.AllPermission;
};

// Allow Channel Framework classes all permission
grant codeBase "file:${was.install.root}/installedChannels/-" {
  permission java.security.AllPermission;
};

grant codeBase "file:${user.install.root}/lib/-" {
  permission java.security.AllPermission;
};
```

```
grant codeBase "file:${user.install.root}/classes/-" {
  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 880, 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.

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 Authentication mechanisms.

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. 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.

A programmatically secured servlet application.

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
    ....

}
```

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 82.

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.

Attention: You can disable security at the application server level by overriding the administrative security setting. For more information, see Securing specific application servers.

Example: Web application code:

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. Using the declarative security model whenever possible is strongly recommended.

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 > Secure administration and applications**.
2. Under Authentication, expand **Web 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 > 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 28. 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.

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.

Important: 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.

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.

Performing the previous steps result in a programmatically secured EJB application.

Hard coding security policies in applications is strongly discouraged. The Java 2 Platform, Enterprise Edition (J2EE) security model capabilities of declaratively specifying security policies is encouraged wherever possible. Use these APIs to develop security-aware EJB applications.

Using J2EE 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

        ....
        }

        ....
}
```

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 that 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;
```

```java
/**
 * 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.

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."
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.

See the "Example: Form login" 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 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. For more information on how to access the form login sample, see "Accessing the Samples (Samples Gallery)" on page 11.

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 security code logs the user out.
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.

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.

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 of servlet filters:

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) {
            }

        }

    }

}
```

Important: 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>
```

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 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. For more information on how to access the form login sample, see "Accessing the Samples (Samples Gallery)" on page 11.

Customizing application login with Java Authentication and Authorization Service

The following topics are covered in this section:

- Developing programmatic logins with the Java Authentication and Authorization Service (JAAS)
- Configuring programmatic logins for JAAS
- Configuring a server-side Java Authentication and Authorization Service authentication and login configuration

Developing programmatic logins with the Java Authentication and Authorization Service:

Use this topic to develop programmatic logins with the Java Authentication and Authorization Service.

Java Authentication and Authorization Service (JAAS) represents the strategic application programming interfaces (API) for authentication.

JAAS replaces the Common Object Request Broker Architecture (CORBA) programmatic login application programming interfaces (APIs).

WebSphere Application Server provides some extension to JAAS:
- Refer to the "Developing applications that use CosNaming (CORBA Naming interface)" on page 1047 article for details on how to set up the environment for thin client applications to access remote resources on a server.
- If the application uses a custom JAAS login configuration, verify that the JAAS login configuration is properly defined. See "Configuring programmatic logins for Java Authentication and Authorization Service" on page 921 for details.
- Some of the JAAS APIs are protected by Java 2 security permissions. If these APIs are used by application code, verify that these permissions are added to the application `was.policy` file.

  For details, see the following articles:
  – "Adding the was.policy file to applications" on page 893
  – "Using PolicyTool to edit policy files" on page 880
  – "Configuring the was.policy file" on page 888

  For more details on which APIs are protected by Java 2 security permissions, check the IBM Developer Kit, Java Technology Edition; JAAS and WebSphere Application Server public APIs documentation in Security: Resources for learning.

  Some of the APIs that are used in the sample code in this documentation and the Java 2 security permissions that are required by these APIs are in the following list:
  – javax.security.auth.login.LoginContext constructors are protected by the javax.security.auth.AuthPermission ″createLoginContext″ object.
  – javax.security.auth.Subject.doAs and com.ibm.websphere.security.auth.WSSubject.doAs methods are protected by the javax.security.auth.AuthPermission ″doAs″ object.
  – javax.security.auth.Subject.doAsPrivileged and com.ibm.websphere.security.auth.WSSubject.doAsPrivileged methods are protected by the javax.security.auth.AuthPermission ″doAsPrivileged″ object.
- **Enhanced model to Java 2 Platform, Enterprise Edition (J2EE) resources for authorization checks**.

  Due to a design oversight in JAAS Version 1.0, the javax.security.auth.Subject.getSubject method does not return the Subject that is associated with the running thread inside a java.security.AccessController.doPrivileged code block. This oversight can present inconsistent behavior, which might have unwanted effects. The com.ibm.websphere.security.auth.WSSubject class provides a workaround to associate a Subject to a running thread. The com.ibm.websphere.security.auth.WSSubject class extends the JAAS model to Java 2 Platform, Enterprise Edition (J2EE) resources for authorization checks. If the Subject associates with the running thread within the com.ibm.websphere.security.auth.WSSubject.doAs method or if the com.ibm.websphere.security.auth.WSSubject.doAsPrivileged code block contains product credentials, the Subject is used for J2EE resource authorization checks.
- **User interface support for defining new JAAS login configuration**.

  You can configure a JAAS login configuration in the administrative console and store the JAAS login configuration in a configuration repository. Applications can define a new JAAS login configuration in the administrative console and the data is persisted in the configuration repository. However, WebSphere Application Server still supports the default JAAS login configuration format (plain text file) that is provided by the JAAS default implementation. If duplicate login configurations are defined in both the

configuration repository and the plain text file format, the one in the repository takes precedence. Advantages to defining the login configuration in the configuration repository includes:

– Administrative console support in defining JAAS login configuration
– Central management of the JAAS login configuration
– Distribution of the JAAS login configuration

- **Application support for programmatic authentication**.

  WebSphere Application Server provides JAAS login configurations for applications to perform programmatic authentication to the WebSphere security runtime. These configurations perform authentication to the WebSphere Application Server-configured authentication mechanism (Simple WebSphere Authentication Mechanism (SWAM) or Lightweight Third Party Authentication (LTPA)) and user registry (Local OS, Lightweight Directory Access Protocol (LDAP), custom registries, or federated repositories) based on the authentication data that is supplied. The authenticated Subject from these JAAS login configurations contains the required principal and credentials that the WebSphere security runtime can use to perform authorization checks on J2EE role-based protected resources.

  Note: SWAM is deprecated in WebSphere Application Server Version 6.1 and will be removed in a future release.

  Here are the JAAS login configurations that are provided by WebSphere Application Server:

  – **WSLogin JAAS login configuration**. A generic JAAS login configuration can use Java clients, client container applications, servlets, JavaServer Pages (JSP) files, and Enterprise JavaBeans (EJB) components to perform authentication based on a user ID and password, or a token to the security runtime for WebSphere Application Server. However, this configuration does not honor the CallbackHandler handler that is specified in the client container deployment descriptor.
  – **ClientContainer JAAS login configuration**. This JAAS login configuration honors the CallbackHandler handler that is specified in the client container deployment descriptor. The login module of this login configuration uses the CallbackHandler handler in the client container deployment descriptor if one is specified, even if the application code specified one callback handler in the login context. This is for a client container application.

    A Subject authenticated with the previously mentioned JAAS login configurations contains a com.ibm.websphere.security.auth.WSPrincipal principal and a com.ibm.websphere.security.cred.WSCredential credential. If the authenticated Subject is passed in the com.ibm.websphere.security.auth.WSSubject.doAs or the other doAs methods, the product security runtime can perform authorization checks on J2EE resources based on the com.ibm.websphere.security.cred.WSCredential Subject.

- **Customer-defined JAAS login configurations**.

  You can define other JAAS login configurations to perform programmatic authentication to your authentication mechanism. See the "Configuring programmatic logins for Java Authentication and Authorization Service" on page 921 for details. For the product security runtime to perform authorization checks, the subjects from these customer-defined JAAS login configurations must contain the required principal and credentials.

- **Naming requirements for programmatic login on a pure Java client**.

  When programmatic login occurs on a pure Java client and the property com.ibm.CORBA.validateBasicAuth equals `true`, it is necessary for the security code to know where the SecurityServer resides. Typically, the default InitialContext is sufficient when a java.naming.provider.url property is set as a system property or when the property is set in the `jndi.properties` file. In other cases it is not desirable to have the same java.naming.provider.url properties set in a system-wide scope. In this case, there is a need to specify security specific bootstrap information in the `sas.client.props` file. The following steps present the order of precedence for determining how to find the SecurityServer in a pure Java client:

1. Use the `sas.client.props` file and look for the following properties:

   ```
   com.ibm.CORBA.securityServerHost=myhost.mydomain
   com.ibm.CORBA.securityServerPort=mybootstrap port
   ```

If you specify these properties, you are guaranteed that security looks here for the SecurityServer. The host and port specified can represent any valid WebSphere host and bootstrap port. The SecurityServer resides on all server processes and therefore it is not important which host or port you choose. If specified, the security infrastructure within the client process look up the SecurityServer based on the information in the `sas.client.props` file.

2. Place the following code in your client application to get a new InitialContext():

```
...
    import java.util.Hashtable;
    import javax.naming.Context;
    import javax.naming.InitialContext;
    ...


// Perform an InitialContext and default lookup prior to logging
// in so that target realm and bootstrap host/port can be
// determined for SecurityServer lookup.

        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);
        Object obj = initialContext.lookup("");

    // programmatic login code goes here.
```

Complete this step prior to running any programmatic login. It is in this code that you specify a URL provider for your naming context, but it must point to a valid WebSphere Application Server within the cell to which you are authenticating. Pointing to one cell allows thread specific programmatic logins going to different cells to have a single system-wide SecurityServer location.

3. Use the new default InitialContext() method relying on the naming precedence rules. These rules are defined in the article, "Example: Getting the default initial context" on page 1033.

See the "Example: Programmatic logins" article.

Example: Programmatic logins:

This example illustrates how application programs can perform a programmatic login using Java Authentication and Authorization Service (JAAS).

```
LoginContext lc = null;

 try {
      lc = new LoginContext("WSLogin",
                new WSCallbackHandlerImpl("userName", "password"));
} catch (LoginException le) {
        System.out.println("Cannot create LoginContext. " + le.getMessage());
        // Insert the error processing code
} catch(SecurityException se) {
        System.out.println("Cannot create LoginContext." + se.getMessage());
        // Insert the error processing code
 }

 try {
        lc.login();
} catch(LoginException le) {
        System.out.println("Fails to create Subject. " + le.getMessage());
         // Insert the error processing code
```

As shown in the example, the new login context is initialized with the WSLogin login configuration and the WSCallbackHandlerImpl callback handler. Use the WSCallbackHandlerImpl instance on a server-side application where you do not want prompting. A WSCallbackHandlerImpl instance is initialized by the specified user ID, password, and realm information. The present WSLoginModuleImpl class implementation that is specified by the WSLogin login configuration can only retrieve authentication information from the specified callback handler. You can construct a login context with a Subject object, but the Subject is disregarded by the present WSLoginModuleImpl implementation. For product client-container applications, replace WSLogin login configuration by ClientContainer login configuration, which specifies the WSClientLoginModuleImpl implementation that is tailored for client container requirements.

For a pure Java application client, the product provides two other callback handler implementations: WSStdinCallbackHandlerImpl and WSGUICallbackHandlerImpl, which prompt for user ID, password, and realm information on the command line and pop-up panel, respectively. You can choose either of these product callback handler implementations, depending on the particular application environment. You can develop a new callback handler if neither of these implementations fit your particular application requirement.

You also can develop your own login module if the default WSLoginModuleImpl implementation fails to meet all your requirements. This product provides utility functions that the custom login module can use, which are described in the next section.

In cases where no java.naming.provider.url property is set as a system property or in the `jndi.properties` file, a default InitialContext context does not function if the product server is not at the `localhost:2809` location. In this situation, construct a new InitialContext context programmatically ahead of the JAAS login. JAAS needs to know where the security server resides to verify that the entered user ID or password is correct, prior to performing a commit method. By constructing a new InitialContext context in the way specified below, the security code has the information that is needed to find the security server location and the target realm.

Attention: The first line starting with `env.put` was split into two lines for illustration purposes only.

```
import java.util.Hashtable;
   import javax.naming.Context;
   import javax.naming.InitialContext;
   ...

// Perform an InitialContext and default lookup prior to logging in so that target realm
// and bootstrap host/port can be determined for SecurityServer lookup.

   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);
   Object obj = initialContext.lookup("");

    LoginContext lc = null;
    try {
       lc = new LoginContext("WSLogin",
                 new WSCallbackHandlerImpl("userName", "realm", "password"));
    } catch (LoginException le) {
       System.out.println("Cannot create LoginContext. " + le.getMessage());
       // insert error processing code
    } catch(SecurityException se) {
       System.out.printlin("Cannot create LoginContext." + se.getMessage());
       // Insert error processing
    }

    try {
        lc.login();
```

```
    } catch(LoginException le) {
        System.out.println("Fails to create Subject. " + le.getMessage());
        // Insert error processing code
    }
```

Configuring programmatic logins for Java Authentication and Authorization Service:

A new JAAS login configuration can be added and modified using the administrative console. The changes are saved in the cell-level security document and are available to all managed application servers.

Java Authentication and Authorization Service (JAAS) is a feature in WebSphere Application Server. JAAS is a collection of WebSphere Application Server strategic authentication APIs and replaces the Common Object Request Broker Architecture (CORBA) programmatic login APIs.

WebSphere Application Server provides some extensions to JAAS:
* **com.ibm.websphere.security.auth.WSSubject**. The com.ibm.websphere.security.auth.WSSubject API extends the JAAS authorization model to Java 2 Platform, Enterprise Edition (J2EE) resources.
* You can configure the JAAS login in the administrative console and store this login configuration in the Application Server configuration. However, WebSphere Application Server still supports the default JAAS login configuration format (plain text file) that is provided by the JAAS default implementation. If duplicate login configurations are defined in both the WebSphere Application Server configuration API and the plain text file format, the one in the WebSphere Application Server configuration API takes precedence. Advantages to defining the login configuration in the WebSphere configuration API include:
  – User interface support in defining JAAS login configuration
  – Central management of the JAAS login configuration
  – Distribution of the JAAS login configuration during installation

  Due to a design oversight in JAAS Version 1.0, the javax.security.auth.Subject.getSubject method does not return the subject that is associated with the running thread inside a java.security.AccessController.doPrivileged code block. This problem presents an inconsistent behavior that might cause unfavorable results. The com.ibm.websphere.security.auth.WSSubject API provides a workaround to associate the subject to a running thread.
* **Proxy LoginModule**. The Proxy LoginModule loads the actual LoginModule module. The default JAAS implementation does not use the thread context class loader to load classes. The LoginModule module cannot load if the LoginModule class file is not in the application class loader or the Java extension class loader class path. Due to this class loader visibility problem, WebSphere Application Server provides a proxy LoginModule module to load the JAAS LoginModule using the thread context class loader. You do not need to place the LoginModule implementation on the application class loader or the class path for the Java extension class loader with this proxy LoginModule module.

  If you do not want to use the Proxy LoginModule module, you can place the LoginModule module in the *app_server_root*/lib/ext/ directory. However, this action is not recommended due to the security risks.

JAAS login configurations are defined in the WebSphere Application Server configuration application programming interface (API) security document. Click **Security > Secure administration, applications, and infrastructure**. Under Java Authentication and Authorization Service, click **Application logins**. The following JAAS login configurations are available:

ClientContainer
> Defines a login configuration and a LoginModule implementation that is similar to that of the WSLogin configuration, but enforces the requirements of the WebSphere Application Server client container. For more information, see "Configuration entry settings for Java Authentication and Authorization Service" on page 925.

DefaultPrincipalMapping,
> Defines a special LoginModule module that is typically used by J2EE connectors to map an authenticated WebSphere Application Server user identity to a set of user authentication data

(user ID and password) for the specified back-end enterprise information system (EIS). For more information about J2EE Connector and the DefaultMappingModule module, refer to the J2EE security section.

WSLogin

Defines a login configuration and a LoginModule implementation that applications can use in general.

A new JAAS login configuration can be added and modified using the administrative console. The changes are saved in the cell-level security document and are available to all managed application servers. An application server restart is required for the changes to take effect at run time.

Attention: Do not remove or delete the predefined JAAS login configurations (such as, ClientContainer, WSLogin, and DefaultPrincipalMapping). Deleting or removing them can cause other enterprise applications to fail.

1. Delete a JAAS login configuration.

   a. Click **Security > Secure administration, applications, and infrastructure**.

   b. Under Java Authentication and Authorization Service, click **Application logins**. The Application Login Configuration panel is displayed.

   c. Select the check box for the login configurations to delete and click **Delete**.

2. Create a new JAAS login configuration.

   a. Click **Security > Secure administration, applications, and infrastructure**.

   b. Under Java Authentication and Authorization Service, click **Application logins**.

   c. Click **New**. The Application Login Configuration panel is displayed.

   d. Specify the alias name of the new JAAS login configuration and click **Apply**. This value is the name of the login configuration that you pass in the javax.security.auth.login.LoginContext implementation for creating a new LoginContext context.

      Click **Apply** to save changes and to add the extra node name that precedes the original alias name. Clicking **OK** does not save the new changes in the `security.xml` file.

   e. Under Additional properties, click **JAAS Login Modules**.

   f. Click **New**.

   g. Specify the Module class name. Specify the WebSphere Application Server proxy LoginModule module because of the limitation of the class loader visibility.

   h. Specify the LoginModule implementation as the delegate property of the Proxy LoginModule module. The WebSphere Application Server proxy LoginModule class name is com.ibm.ws.security.common.auth.module.proxy.WSLoginModuleProxy.

   i. Select **Authentication strategy** from the list and click **Apply**.

   j. Under Additional properties, click **Custom properties**. The Custom properties panel is displayed for the selected LoginModule.

   k. Create a new property with the name `delegate` and the value of the real LoginModule implementation. You can specify other properties like debug with the `true` value. These properties are passed to the LoginModule class as options to the initialize method of the LoginModule instance.

   l. Click **Save**.

      For a Network Deployment installation, make sure that a file synchronization operation is performed to propagate the changes to other nodes.

Several locations are within the WebSphere Application Server directory structure where you can place a JAAS login module. The following list provides locations for the JAAS login module in order of recommendation:

- Within an enterprise archive (EAR) file for a specific Java 2 Platform, Enterprise Edition (J2EE) application.

  If you place the login module within the EAR file, the login module is accessible by the specific application only.

- In the WebSphere Application Server-shared library.

  If you place the login module in the shared library, you must specify which applications can access the module. For more information on shared libraries, see Managing shared libraries.

- In the Java extensions directory.

  If you place the JAAS login module in the Java extensions directory, the login module is available to all applications.

  Although the Java extensions directory provides the greatest availability for the login module, place the login module in an application EAR file. If other applications need to access the same login module, consider using shared libraries.

3. Change the plain text file.

   WebSphere Application Server supports the default JAAS login configuration format, which is a plain text file, that is provided by the JAAS default implementation. However, a tool is not provided that edits plain text files in this format. You can define the JAAS login configuration in the plain text file, which is located in the *app_server_root*/properties/wsjaas.conf file. Any syntax errors can cause the incorrect parsing of the plain JAAS login configuration text file. This problem can cause other applications to fail.

   Java client programs that use the Java Authentication and Authorization Service (JAAS) for authentication must invoke with the JAAS configuration file specified. This configuration file is set in the *app_server_root*/bin/launchClient.bat file as:

   ```
   set JAAS_LOGIN_CONFIG=-Djava.security.auth.login.config=%install_root%\properties\wsjaas_client.conf
   ```

   If the **launchClient.bat** file is not used to invoke the Java client program, verify that the appropriate JAAS configuration file is passed to the Java virtual machine with the **-Djava.security.auth.login.config** flag.

A new JAAS login configuration is created or an old JAAS login configuration is removed. An enterprise application can use a newly created JAAS login configuration without restarting the application server process.

However, new JAAS login configurations that are defined in the *app_server_root*/properties/wsjaas.conf file, do not refresh automatically. Restart the application servers to validate changes. These JAAS login configurations are specific to a particular node and are not available for other application servers running on other nodes.

Create new JAAS login configurations that are used by enterprise applications to perform custom authentication. Use these newly defined JAAS login configurations to perform programmatic login.

Login configuration for Java Authentication and Authorization Service:

Java Authentication and Authorization Service (JAAS) is a new feature in WebSphere Application Server. JAAS is WebSphere Application Server strategic application programming interface (API) for authentication that replaces the Common Object Request Broker Architecture (CORBA) programmatic login API.

WebSphere Application Server provides some extensions to JAAS:
- **com.ibm.websphere.security.auth.WSSubject**: The com.ibm.websphere.security.auth.WSSubject API extends the JAAS authorization model to Java 2 Platform, Enterprise Edition (J2EE) resources. You can configure JAAS login in the administrative console or by using the scripting functions and store this configuration in the WebSphere Application Server configuration API. However, WebSphere Application Server still supports the default JAAS login configuration format, a plain text file, which is provided by the JAAS default implementation. If duplicate login configurations are defined in both the WebSphere

Application Server configuration API and the plain text file format, the one in the WebSphere Application Server configuration API takes precedence. Advantages to defining the login configuration in the WebSphere configuration API include:

- User interface support in defining JAAS login configuration
- Central management of the JAAS login configuration
- Distribution of the JAAS login configuration in a Network Deployment product installation

Due to a design oversight in JAAS 1.0, the javax.security.auth.Subject.getSubject method does not return the Subject that is associated with the running thread inside a java.security.AccessController.doPrivileged code block. This action can present an inconsistent behavior that is problematic. The com.ibm.websphere.security.auth.WSSubject extension provides a workaround to associate the Subject to the running thread. The com.ibm.websphere.security.auth.WSSubject extension expands the JAAS authorization model to J2EE resources.

Why WebSphere Application Server has its own subject class: You can retrieve the subjects in a Subject.doAs block with the Subject.getSubject call. However, this procedure does not work if an AccessController.doPrivileged call is contained within the Subject.doAs block. In the following example, s1 is equal to s, but s2 is null:

```
* AccessController.doPrivileged() not only truncates the Subject propagation,
* but also reduces the permissions. It does not include the JAAS security
* policy defined for the principals in the Subject.
Subject.doAs(s, new PrivilegedAction() {
  public Object run() {
    System.out.println("Within Subject.doAsPrivileged()");
    Subject s1 = Subject.getSubject(AccessController.getContext());
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
      Subject s2 = Subject.getSubject(AccessController.getContext());
      return null;
    }
  });
  return null;
}
});
```

- JAAS Login Configuration can be configured in either the administrative console or by using the scripting functions and stored in the WebSphere Application Server configuration repository. An application can define a new JAAS login configuration in the administrative console and persist the data in the configuration repository that is stored in the WebSphere Application Server configuration API. However, WebSphere Application Server still supports the default JAAS login configuration format that is provided by the JAAS default implementation. If duplicate login configurations are defined in both the WebSphere Application Server configuration API and the plan text file format, the one in the WebSphere Application Server configuration API takes precedence. The advantages to defining the login configuration in the WebSphere Application Server configuration API include:
  - UI support in defining JAAS login configuration.
  - The JAAS configuration login configuration can be managed centrally.
  - The JAAS configuration login configuration is distributed in a Network Deployment installation.
- **Proxy LoginModule**: The Proxy.LoginModule is a proxy to the configured user or the system-defined module that the context class loader uses to load the module instead of the system class loader. The default JAAS implementation does not use the thread context class loader to load classes. The LoginModule module cannot be loaded if the LoginModule class file is not in the application class loader or the class loader class path for the Java extension. WebSphere Application Server provides a proxy LoginModule module to load the JAAS LoginModule using the thread context class loader. You do not need to place the LoginModule implementation on the application class loader or the class loader class path for the Java extension with this proxy LoginModule module.

Tip: Do not remove or delete the predefined JAAS login configurations (ClientContainer, WSLogin and DefaultPrincipalMapping). Deleting or removing them can cause other enterprise applications to fail.

A system administrator determines the authentication technologies, or login modules, to use for each application and configures them in a login configuration. The source of the configuration information, for example, a file or a database, is up to the current javax.security.auth.login.Configuration implementation.

The WebSphere Application Server implementation permits the definition of the login configuration in both the WebSphere Application Server configuration API security document and in a JAAS configuration file, where the former takes precedence.

JAAS login configurations are defined in the API security document for WebSphere Application Server configuration for applications to use. To access the configurations, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Java Authentication and Authorization Service, click **Application logins**.

The WSLogin module defines a login configuration and the LoginModule implementation that can be used by applications in general.

The ClientContainer module defines a login configuration and the LoginModule implementation that is similar to the WSLogin module, but enforces the requirements of the WebSphere Application Server client container.

The DefaultPrincipalMapping module defines a special LoginModule that is typically used by Java 2 Connector to map an authenticated WebSphere Application Server user identity to a set of user authentication data (user ID and password) for the specified back-end enterprise information system (EIS). For more information about Java 2 Connector and the DefaultMappingModule, see the Java 2 Security section.

A new JAAS login configuration can be added and modified using the administrative console. The changes are saved in the cell-level security document and are available to all managed application servers. An application server restart is required for the changes to take effect at runtime and for the client container login configuration to be made available.

WebSphere Application Server also reads JAAS configuration information from the `wsjaas.conf` file under the properties subdirectory of the root directory under which WebSphere Application Server is installed. Changes made to the `wsjaas.conf` file are used only by the local application server and take effect after the application server restarts. The JAAS configuration in the WebSphere Application Server configuration API security document takes precedence over that defined in the `wsjaas.conf` file. A configuration entry in the `wsjaas.conf` is overridden by an entry of the same alias name in the WebSphere Application Server configuration API security document.

The Java Authentication and Authorization Service (JAAS) login configuration entries in the administrative console are propagated to the server runtime when they are created, not when the configuration is saved. However, the deleted JAAS login configuration entries are not removed from the server runtime. To remove the entries, save the new configuration, then stop and restart the server.

The Samples Gallery provides a JAAS login sample that demonstrates how to use JAAS with WebSphere Application Server. The sample uses a server-side login with JAAS to authenticate a user with the security runtime for WebSphere Application Server. The sample demonstrates the following technology:

- Java 2 Platform, Enterprise Edition (J2EE) Java Authentication and Authorization Service (JAAS)
- JAAS for WebSphere Application Server
- WebSphere Application Server security

The form login sample is a component of the technology samples. For more information on how to access the form login sample, see "Accessing the Samples (Samples Gallery)" on page 11.

Configuration entry settings for Java Authentication and Authorization Service:

Use this page to specify a list of Java Authentication and Authorization Service (JAAS) login configurations for the application code to use, including Java 2 Platform, Enterprise Edition (J2EE) components such as enterprise beans, JavaServer Pages (JSP) files, servlets, resource adapters, and message-driven beans (MDBs).

To view this administrative console page, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, click **Java Authentication and Authorization Service > Application logins**.

Read the JAAS specifications before you begin defining additional login modules for authenticating to the application server security run time. You can define additional login configurations for your applications. However, if the application server LoginModule com.ibm.ws.security.common.auth.module.WSLoginModuleImpl module is not used or the LoginModule module does not produce a credential that is recognized by the application server. The application server security run time cannot use the authenticated subject from these login configurations for an authorization check for resource access.

You must invoke Java client programs that use Java Authentication and Authorization Service (JAAS) for authentication with a JAAS configuration file that is specified. The application server supplies the wsjaas_client.conf default JAAS configuration file under the *app_server_root*/properties directory. This configuration file is set in the *app_server_root*/bin/launchClient.bat file as:

```
set JAAS_LOGIN_CONFIG=-Djava.security.auth.login.config=%WAS_HOME%\properties\wsjaas_client.conf
```

ClientContainer:

Specifies the login configuration used by the client container application, which uses the CallbackHandler API that is defined in the client container deployment descriptor.

The ClientContainer configuration is the default login configuration for the application server. Do not remove this default, as other applications that use it fail.

Default:	ClientContainer

DefaultPrincipalMapping:

Specifies the login configuration that is used by Java 2 Connectors to map users to principals that are defined in the J2C authentication data entries.

The ClientContainer configuration is the default login configuration for the application server. Do not remove this default, as other applications that use it fail.

Default:	ClientContainer

WSLogin:

Indicates whether all of the applications can use the WSLogin configuration to perform authentication for the application server security run time.

This login configuration does not honor the CallbackHandler handler that is defined in the client container deployment descriptor. To use this functionality, use the ClientContainer login configuration.

The WSLogin configuration is the default login configuration for the application server. Do not remove this default because other administrative applications that use it fail. This login configuration authenticates users for the application server security run time. Use the credentials from the authenticated subject that

are returned from this login configuration as an authorization check for access to application server resources.

Default: ClientContainer

System login configuration entry settings for Java Authentication and Authorization Service:

Use this page to specify a list of Java Authentication and Authorization Service (JAAS) system login configurations.

To view this administrative console page, complete the following steps:
1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, click **Java Authentication and Authorization Service > System logins**.

Read the Java Authentication and Authorization Service documentation before you begin defining additional login modules for authenticating to the application server security runtime. Do not remove the following system login modules:
- RMI_INBOUND
- WEB_INBOUND
- DEFAULT
- RMI_OUTBOUND
- SWAM
- wssecurity.IDAssertion
- wssecurity.signature
- wssecurity.PKCS7
- wssecurity.PkiPath
- wssecurity.UsernameToken
- wssecurity.X509BST
- LTPA
- LTPA_WEB

RMI_INBOUND, WEB_INBOUND, DEFAULT:

Processes inbound login requests for Remote Method Invocation (RMI), Web applications, and most of the other login protocols.

RMI_INBOUND
> This login configuration handles logins for inbound RMI requests. Typically, these logins are requests for authenticated access to Enterprise JavaBeans (EJB) files. When using the RMI connector, these logins might be requests for Java Management Extensions (JMX).

WEB_INBOUND
> This login configuration handles logins for Web application requests, which include servlets and JavaServer Pages (JSP) files. This login configuration can interact with the output that is generated from a trust association interceptor (TAI), if configured. The Subject that is passed into the WEB_INBOUND login configuration might contain objects that are generated by the TAI.

DEFAULT
> This login configuration handles the logins for inbound requests that are made by most of the other protocols and internal authentications.

These three login configurations will pass in the following callback information, which is handled by the login modules within these configurations. These callbacks are not passed in at the same time. However, the combination of these callbacks determines how the application server authenticates the user.

Callback

```
callbacks[0] = new javax.security.auth.callback.
NameCallback("Username: ");
```

Responsibility

Collects the user name that is provided during a login. This information can be the user name for the following types of logins:

- User name and password login, which is known as *basic authentication*.
- User name only for identity assertion.

Callback

```
callbacks[1] = new javax.security.auth.callback.
PasswordCallback("Password: ", false);
```

Responsibility

Collects the password that is provided during a login.

Callback

```
callbacks[2] = new com.ibm.websphere.security.auth.callback.
WSCredTokenCallbackImpl("Credential Token: ");
```

Responsibility

Collects the Lightweight Third Party Authentication (LTPA) token or other token type during a login. Typically, this information is present when a user name and a password are not present.

Callback

```
callbacks[3] = new com.ibm.wsspi.security.auth.callback.
WSTokenHolderCallback("Authz Token List: ");
```

Responsibility

Collects the ArrayList list of the TokenHolder objects that are returned from the call to the WSOpaqueTokenHelper. The callback uses the createTokenHolderListFromOpaqueToken method with the Common Secure Interoperability version 2 (CSIv2) authorization token as input.

Restriction: This callback is present only when the **Security Attribute Propagation** option is enabled and this login is a propagation login. In a propagation login, sufficient security attributes are propagated with the request to prevent having to access the user registry for additional attributes. You must enable security attribute propagation for both the outbound and inbound authentication.

You can enable the **Security attribute propagation** option for CSIv2 outbound authentication by completing the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, expand RMI/IIOP security and click **CSIv2 outbound authentication**.
3. Enable the **Security attribute propagation** option.

You can enable the **Security attribute propagation** option for CSIv2 inbound authentication by completing the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, expand RMI/IIOP security and click **CSIv2 inbound authentication**.
3. Enable the **Security attribute propagation** option.

In system login configurations, the application server authenticates the user based upon the information that is collected by the callbacks. However, a custom login module does not need to act upon any of these callbacks. The following list explains the typical combinations of these callbacks:

- The `callbacks[0] = new javax.security.auth.callback.NameCallback("Username: ");` callback only

  This callback occurs for CSIv2 identity assertion; Web and CSIv2 X509 certificate logins; old-style trust association interceptor logins, and so on. In Web and CSIv2 X509 certificate logins, the application server maps the certificate to a user name. This callback is used by any login type that establishes trust with the user name only.

- Both the `callbacks[0] = new javax.security.auth.callback.NameCallback("Username: ");` callback and the `callbacks[1] = new javax.security.auth.callback.PasswordCallback("Password: ", false);` callbacks.

  This combination of callbacks is typical for basic authentication logins. Most user authentications occur using these two callbacks.

- The `callbacks[2] = new com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl("Credential Token: ");` only

  This callback is used to validate a Lightweight Third Party Authentication (LTPA) token. This validation typically occurs during a single sign-on (SSO) or downstream login. Any time a request originates from the application server, instead of a pure client, the LTPA token flows to the target server. For single sign-on (SSO), the LTPA token is received in the cookie and the token is used for login. If a custom login module needs the user name from an LTPA token, the module can use the following method to retrieve the unique ID from the token:

  ```
  com.ibm.wsspi.security.token.WSSecurityPropagationHelper.
  validateLTPAToken(byte[])
  ```

  After retrieving the unique ID, use the following method to get the user name:

  ```
  com.ibm.wsspi.security.token.WSSecurityPropagationHelper.
  getUserFromUniqueID(uniqueID)
  ```

  Important: Any time a custom login module is plugged in ahead of the application server login modules and it changes the identity using a credential mapping service, it is important that this login module validates the LTPA token, if present. Calling the following method is sufficient to validate the trust in the LTPA token:

  ```
  com.ibm.wsspi.security.token.WSSecurityPropagationHelper.
  validateLTPAToken(byte[])
  ```

  The receiving server must have the same LTPA keys as the sending server for this validation to be successful. A security exposure is possible if you do not validate this LTPA token, when present.

- A combination of any of the previously mentioned callbacks plus the `callbacks[3] = new com.ibm.wsspi.security.auth.callback.WSTokenHolderCallback("Authz Token List: ");` callback.

  This callback indicates that some propagated attributes arrived at the server. The propagated attributes still require one of the following authentication methods:

  - `callbacks[0] = new javax.security.auth.callback.NameCallback("Username: ");`
  - `callbacks[1] = new javax.security.auth.callback.PasswordCallback("Password: ", false);`
  - `callbacks[2] = new com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl("Credential Token: ");`

If the attributes are added to the Subject from a pure client, then the NameCallback and PasswordCallback callbacks authenticate the information and the objects that are serialized in the token holder are added to the authenticated Subject.

If both CSIv2 identity assertion and propagation are enabled, the application server uses the NameCallback callback and the token holder, which contains all of the propagated attributes, to deserialize most of the objects. The application server uses the NameCallback callback because trust is established with the servers that you indicate in the CSIv2 trusted server list. To specify trusted servers, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, click **CSIv2 inbound authentication**.

A custom login module needs to handle custom serialization. For more information, see ″Security attribute propagation″ in the information center.

In addition to the callbacks that are defined previously, the WEB_INBOUND login configuration can contain the following additional callbacks only:

Callback
```
callbacks[4] = new com.ibm.websphere.security.auth.callback.
WSServletRequestCallback("HttpServletRequest: ");
```

Responsibility
Collects the HTTP servlet request object, if presented. This callback enables login modules to retrieve information from the HTTP request to use during a login.

Callback
```
callbacks[5] = new com.ibm.websphere.security.auth.callback.
WSServletResponseCallback("HttpServletResponse: ");
```

Responsibility
Collects the HTTP servlet response object, if presented. This callback enables login modules to add information into the HTTP response as a result of the login. For example, login modules might add the SingleSignonCookie cookie to the response.

Callback
```
callbacks[6] = new com.ibm.websphere.security.auth.callback.
WSAppContextCallback("ApplicationContextCallback: ");
```

Responsibility
Collects the Web application context used during the login. This callback consists of a hashtable, which if present contains the application name and the redirected Web address.

Callback
callbacks[7] = new WSRealmNameCallbackImpl(″Realm Name: ″, <default_realm>);

Responsibility
Collects the realm name for the login information. The realm information might not always be provided and should be assumed to be the current realm if it is not provided.

Callback
callbacks[8] = new WSX509CertificateChainCallback(″X509Certificate[]: ″);

Responsibility
If the login source is an X509Certificate from SSL client authentication, this callback contains the certificate that was validated by SSL. The ltpaLoginModule calls the same mapping functions as in previous releases. Once it is passed into the login, it provides a custom login module with the opportunity to map the certificate in a custom way. It then perform a hashtable login (see Example: Custom login module for inbound mapping for an example of a hashtable login).

If you want to use security attribute propagation with the WEB_INBOUND login configuration, you can enable **Web inbound security attribute propagation** option on the Single sign-on panel.

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, expand Web security and click **Single sign-on (SSO)**.
3. Select the **Web inbound security attribute propagation** option.

The following login modules are predefined for the RMI_INBOUND, WEB_INBOUND, and DEFAULT system login configurations. You can add custom login modules before, between, or after any of these login modules, but you cannot remove these predefined login modules:

- com.ibm.ws.security.server.lm.ltpaLoginModule

  Performs the primary login when attribute propagation is either enabled or disabled. A primary login uses normal authentication information such as a user ID and password, an LTPA token, or a trust association interceptor (TAI) and a certificate distinguished name (DN). If any of the following scenarios are true, this login module is not used and the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule module performs the primary login:

  – The java.util.Hashtable object with the required user attributes is contained in the Subject.
  – The java.util.Hashtable object with the required user attributes is present in the sharedState HashMap of the LoginContext.
  – The WSTokenHolderCallback callback is present without a specified password. If a user name and a password are present with a WSTokenHolderCallback callback, which indicates propagated information, the request likely originates from either a pure client or a server from a different realm that mapped the existing identity to a user ID and password.

- com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule

  This login module performs the primary login using the normal authentication information if any of the following conditions are true:

  – A java.util.Hashtable object with required user attributes is contained in the Subject.
  – A java.util.Hashtable object with required user attributes is present in the sharedState HashMap of the LoginContext context.
  – The WSTokenHolderCallback callback is present without a PasswordCallback callback.

  When the java.util.Hashtable object is present, the login module maps the object attributes into a valid Subject. When the WSTokenHolderCallback callback is present, the login module deserializes the byte token objects and regenerates the serialized Subject contents. The java.util.Hashtable hashtable takes precedence over all of the other forms of login. Be careful to avoid duplicating or overriding what the application server might have propagated previously.

  By specifying a java.util.Hashtable hashtable to take precedence over other authentication information, the custom login module must have already verified the LTPA token, if present, to establish sufficient trust. The custom login module can use the com.ibm.wsspi.security.token.WSSecurityPropagationHelper.validationLTPAToken(byte[]) method to validate the LTPA token present in the WSCredTokenCallback callback. Failure to validate the LTPA token presents a security risk.

  For more information on adding a hashtable containing well-known and well-formed attributes used by the application server as sufficient login information, see ″Configuring inbound identity mapping″ in the information center.

RMI_OUTBOUND:

Processes Remote Method Invocation (RMI) requests that are sent outbound to another server when either the com.ibm.CSI.rmiOutboundLoginEnabled or the com.ibm.CSIOutboundPropagationEnabled properties are true.

These properties are set in the CSIv2 authentication panel. To access the panel, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, expand RMI/IIOP security and click **CSIv2 outbound authentication**.

To set the com.ibm.CSI.rmiOutboundLoginEnabled property, select **Custom outbound mapping**. To set the com.ibm.CSIOutboundPropagationEnabled property, select the **Security attribute propagation** option.

This login configuration determines the security capabilities of the target server and its security domain. For example, if the application server Version 5.1.1 or later (or 5.1.0.2 for z/OS) communicates with a Version 5.x Application Server, then the Version 5.1.1 Application Server sends the authentication information only, using an LTPA token, to the Version 5.x Application Server. However, if WebSphere Application Server Version 5.1.1 or later communicates with a Version 5.1.x Application Server, the authentication and authorization information is sent to the receiving application server if propagation is enabled at both the sending and receiving servers. When the application server sends both the authentication and authorization information downstream, the application server removes the need to access the user registry again and look up the security attributes of the user for authorization purposes. Additionally, any custom objects that are added at the sending server are present in the Subject at the downstream server.

The following callback is available in the RMI_OUTBOUND login configuration. You can use the `com.ibm.wsspi.security.csiv2.CSIv2PerformPolicy` object that is returned by this callback to query the security policy for this particular outbound request. This query can help determine if the target realm is different than the current realm and if the application server must map the realm. For more information, see "Configuring outbound mapping to a different target realm" in the information center.

Callback
>   callbacks[0] = new WSProtocolPolicyCallback("Protocol Policy Callback: ");

Responsibility

> Provides protocol-specific policy information for the login modules on this outbound invocation. This information is used to determine the level of security, including the target realm, target security requirements, and coalesced security requirements.

> The following method obtains the CSIv2PerformPolicy policy from this specific login module:

>   csiv2PerformPolicy = (CSIv2PerformPolicy)
>   ((WSProtocolPolicyCallback)callbacks[0]).getProtocolPolicy();

> A different protocol other than RMI might have a different type of policy object.

The following login module is predefined in the RMI_OUTBOUND login configuration. You can add custom login modules before, between, or after any of these login modules, but you cannot remove these predefined login modules.

com.ibm.ws.security.lm.wsMapCSIv2OutboundLoginModule
> Retrieves the following tokens and objects before creating an opaque byte that is sent to another server by using the Common Secure Interoperability Version 2 (CSIv2) authorization token layer:
> - Forwardable com.ibm.wsspi.security.token.Token implementations from the Subject
> - Serializable custom objects from the Subject
> - Propagation tokens from the thread

> You can use a custom login module prior to this login module to perform credential mapping. However, it is recommended that the login module change the contents of the Subject that is passed in during the login phase. If this recommendation is followed, the login modules are processed after this login module acts on the new Subject contents.

For more information, see "Configuring outbound mapping to a different target realm" in the information center.

SWAM:

Processes login requests in a single server environment when Simple WebSphere Authentication Mechanism (SWAM) is used as the authentication method.

SWAM does not support forwardable credentials. When SWAM is the authentication method, the application server cannot send requests from server to server. In this case, you must use LTPA.

Note: The SWAM login configuration is deprecated and will be removed in a future release.

wssecurity.IDAssertion:

Processes login configuration requests for Web services security using identity assertion.

This login configuration is for Version 5.x systems. For more information, see ″Identity assertion authentication method″ in the information center.

wssecurity.PKCS7:

Verifies an X.509 certificate with a certificate revocation list in a Public Key Cryptography Standards #7 (PKCS7) object.

This login configuration is for Version 6.0.x systems.

wssecurity.PkiPath:

Verifies an X.509 certificate with a public key infrastructure (PKI) path.

This login configuration is for Version 6.0.x systems.

wssecurity.signature:

Processes login configuration requests for Web services security using digital signature validation.

This login configuration is for Version 5.x systems.

wssecurity.UsernameToken:

Verifies basic authentication (user name and password).

This login configuration is for Version 6.0.x systems.

wssecurity.X509BST:

Verifies an X.509 binary security token (BST) by checking the validity of the certificate and the certificate path.

This login configuration is for Version 6.0.x systems.

LTPA_WEB:

Processes login requests to components in the Web container such as servlets and JavaServer pages (JSP) files.

The com.ibm.ws.security.web.AuthenLoginModule login module is predefined in the LTPA login configuration. You can add custom login modules before or after this module in the LTPA_WEB login configuration.

The LTPA_WEB login configuration can process the HttpServletRequest object, the HttpServletResponse object, and the Web application name that are passed in using a callback handler. For more information, see ″Example: Customizing a server-side Java Authentication and Authorization Service authentication and logon configuration″ in the information center.

LTPA:

Processes login requests that are not handled by the LTPA_WEB login configuration.

This login configuration is used by WebSphere Application Server Version 5.1 and previous versions.

The com.ibm.ws.security.server.lm.ltpaLoginModule login module is predefined in the LTPA login configuration. You can add custom login modules before or after this module in the LTPA login configuration. For more information, see ″Example: Customizing a server-side Java Authentication and Authorization Service authentication and logon configuration″ in the information center.

Login module settings for Java Authentication and Authorization Service:

Use this page to define the login module for a Java Authentication and Authorization Service (JAAS) login configuration.

You can define the JAAS login modules for application and system logins. To define these login modules in the administrative console, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, click **Java Authentication and Authorization Service > Application logins** or **System logins** > *alias_name*.
3. Under Additional properties, click **JAAS login modules**.

Module class name:

Specifies the class name of the given login module.

Data type:	String

Use login module proxy:

Specifies that the Java Authentication and Authorization Service (JAAS) loads the login module proxy class. JAAS then delegates calls to the login module classes that are defined in the Module class name field.

Use this option when you use both Version 5.x and Version 6 Application Servers in the same environment. If you migrate a Version 5.x Application Server to Version 6, WebSphere Application Server Version 6 automatically enables this option. If you have Version 6 only cells in your environment, you might choose to deselect this option.

Default:	Enabled

Proxy class name:

Specifies the name of the proxy login module class.

The default login modules that are defined by the application server use the com.ibm.ws.security.common.auth.module.WSLoginModuleProxy proxy LoginModule class. This proxy class loads the application server login module with the thread context class loader and delegates all the

operations to the real login module implementation. The real login module implementation is specified as the delegate option in the option configuration. The proxy class is needed because the Developer Kit application class loaders do not have visibility of the application server product class loaders.

Data type: String

Authentication strategy:

Specifies the authentication behavior as authentication proceeds down the list of login modules.

A Java Authentication and Authorization Service (JAAS) authentication provider supplies the authentication strategy. In JAAS, an authentication strategy is implemented through the LoginModule interface.

Data type: String
Default: Required
Range: Required, Requisite, Sufficient and Optional

Required
>   The LoginModule module is required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list for each realm.

Requisite
>   The LoginModule module is required to succeed. If authentication is successful, the process continues down the LoginModule list in the realm entry. If authentication fails, control immediately returns to the application. Authentication does not proceed down the LoginModule list.

Sufficient
>   The LoginModule module is not required to succeed. If authentication succeeds, control immediately returns to the application. Authentication does not proceed down the LoginModule list. If authentication fails, the process continues down the list.

Optional
>   The LoginModule module is not required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list.

Specify additional options by clicking **Custom Properties** under Additional Properties. These name and value pairs are passed to the login modules during initialization. This process is one of the mechanisms that is used to passed information to login modules.

Module order:

Specifies the order in which the Java Authentication and Authorization Service (JAAS) login modules are processed.

Click **Set Order** to change the processing order of the login modules.

Login module order settings for Java Authentication and Authorization Service:

Use this page to specify the order in which the application server processes the login configuration modules.

You can specify the order of the login modules for application and system logins. To define these login modules in the administrative console, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.
2. Under Authentication, click **Java Authentication and Authorization Service > Application logins** or **System logins** > *alias*. You can create a new configuration by clicking **New**.

3. Under Additional properties, click **JAAS login modules**.

4. Click **Set order**.

When you select one of the JAAS login module class names, you can move that class name up and down the list. After you click **OK** and save the changes, the new order is reflected on either the Application login configuration or the System login configuration panel.

Login configuration settings for Java Authentication and Authorization Service:

Use this page to configure application login configurations.

To view this administrative console page, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.

2. Under Authentication, click **Java Authentication and Authorization Service > Application logins** or **System logins > *alias_name***.

Click **Apply** to save changes and to add the extra node name that precedes the original alias name. Clicking **OK** does not save the new changes in the `security.xml` file.

Alias:

Specifies the alias name of the application login.

Do not use the forward slash character (/) in the alias name when defining JAAS login configuration entries. The JAAS login configuration parser cannot process the forward slash character.

Data type: String

Managing J2EE Connector Architecture authentication data entries:

This task creates and deletes Java 2 Connector (J2C) authentication data entries.

Java 2 Platform, Enterprise Edition (J2EE) Connector authentication data entries are used by resource adapters and Java DataBase Connectivity (JDBC) data sources. A J2EE Connector authentication data entry contains authentication data, which includes the following information:

Alias   An identifier that identifies the authentication data entry. When configuring resource adapters or data sources, the administrator can specify which authentication data to choose using the corresponding alias.

User ID

A user identity of the intended security domain. For example, if a particular authentication data entry is used to open a new connection to DB2, this entry contains a DB2 user identity.

Password

The password of the user identity is encoded in the configuration repository.

Description

A short text description.

1. Delete a J2C authentication data entry.

   a. Click **Security > Secure administration, applications, and infrastructure**.

   b. Under Java Authentication and Authorization Service, click **J2C authentication data**. The **J2C Authentication Data Entries** panel is displayed.

   c. Select the check boxes for the entries to delete and click **Delete**. Before deleting or removing an authentication data entry, make sure that it is not used or referenced by any resource adapter or data source. If the deleted authentication data entry is used or referenced by a resource, the application that uses the resource adapter or the data source fails to connect to the resources.

2. Create a new J2C authentication data entry.

a. Click **Security > Secure administration, applications, and infrastructure**.

b. Under Java Authentication and Authorization Service, click **J2C authentication data**. The **J2C Authentication Data Entries** panel is displayed.

c. Click **New**.

d. Enter a unique alias, a valid user ID, a valid password, and a short description (optional).

e. Click **OK** or **Apply**. No validation for the user ID and password is required.

f. Click **Save**.

Make sure that a file synchronized operation is performed to propagate the changes to other nodes.

A new J2C authentication data entry is created or an old entry is removed. The newly created entry is visible without restarting the application server process to use in the data source definition. But the entry is only in effect after the server is restarted. Specifically, the authentication data is loaded by an application server when starting an application and is shared among applications in the same application server.

If you create or update a data source that points to a newly created J2C authentication data alias, the test connection fails to connect until you restart the deployment manager. After you restart the deployment manager, the J2C authentication data is reflected in the run-time configuration. Any changes to the J2C authentication data fields require a deployment manager restart for the changes to take effect. The node agent must also be restarted.

This step defines authentication data that you can share among resource adapters and data sources. Use the authentication data entry that is defined in the resource adapters or the data sources.

Java 2 Connector authentication data entry settings:

Use this page as a central place for administrators to define authentication data, which includes user identities and passwords. These values can reference authentication data entries by resource adapters, data sources, and other configurations that require authentication data using an alias.

You can display this page directly from the Java Authentication and Authorization Service (JAAS) configuration page or from other pages for resources that use J2EE Connector (J2C) authentication data entries. To view this administrative page, complete the following steps:

1. Click **Security > Secure administration, applications, and infrastructure**.

2. Under Authentication, click **Java Authentication and Authorization Service > J2C authentication data**.

Deleting authentication data entries: Be careful when deleting authentication data entries. If the deleted authentication data is used by other configurations, the initializing resources process fails.

Define a new authentication data entry by clicking **New**.

Alias:

Specifies the name of the authentication data entry.

Data type:	String
Units:	String
Default:	None

User ID:

Specifies the user identity.

| **Data type:** | String |

Password:

Specifies the password that is associated with the user identity.

This field is not available on the collections panel. However, the panel is available when you create a new J2C authentication data entry.

Data type: String

Description:

Specifies an optional description of the authentication data entry. For example, this authentication data entry is used to connect to DB2.

Data type: String

J2C principal mapping modules:

You can develop your own J2EE Connector (J2C) mapping module if your application requires more sophisticated mapping functions. The mapping login module that you might have developed on WebSphere Application Server Version 5.*x* is still supported in WebSphere Application Server Version 6.0.*x* and later.

You can use the Version 5.*x* login modules in the connection factory mapping configuration. These login modules can also be used in the reference mapping configuration for the resource manager connection factory. A version 5.*x* mapping login module is not able to use the custom mapping properties.

If you want to develop a new mapping login module in Version 6.0.*x* and later, use the programming interface that is described in the following sections.

transition: Migrate your Version 5.*x* mapping login module to use the new programming model and the new custom properties as well as the mapping configuration isolation at application scope. Note that mapping login modules that are developed using WebSphere Application Server Version 6.0.*x* cannot be used in the deprecated mapping configuration for the resource connection factory.

Invoking the login module for the resource reference mapping

A com.ibm.wsspi.security.auth.callback.WSMappingCallbackHandler class, which implements the javax.security.auth.callback.CallbackHandler interface, is a new WebSphere Application Service Provider Programming Interface (SPI) in WebSphere Application Server Version 6.0.*x*.

Application code uses the com.ibm.wsspi.security.auth.callback.WSMappingCallbackHandlerFactory helper class to retrieve a CallbackHandler object:

```
package  com.ibm.wsspi.security.auth.callback;

public class WSMappingCallbackHandlerFactory {
 private WSMappingCallbackHandlerFactory;
 public static CallbackHandler getMappingCallbackHandler(
ManagedConnectionFactory mcf,
HashMap mappingProperties);
}
```

The WSMappingCallbackHandler class implements the CallbackHandler interface:

```
package com.ibm.wsspi.security.auth.callback;

public class WSMappingCallbackHandler implements CallbackHandler {
 public WSMappingCallbackHandler(ManagedConnectionFactory mcf,
HashMap mappingProperties);
 public void handle(Callback[] callbacks) throws IOException,
     UnsupportedCallbackException;
}
```

The WSMappingCallbackHandler handler can manage two new callback types that are defined in Version 6.0.x:

```
com.ibm.wsspi.security.auth.callback.WSManagedConnectionFactoryCallback
com.ibm.wsspi.security.auth.callback.WSMappingPropertiesCallback
```

The new login modules use the two callback types that are used at the reference mapping configuration for the resource manager connection factory. The WSManagedConnectionFactoryCallback callback provides a ManagedConnectionFactory instance that you set in the PasswordCredential credential. With this setting, the ManagedConnectionFactory instance can determine whether a PasswordCredential instance is used for signon to the target Enterprise Information Systems (EIS) instance. The WSMappingPropertiesCallback callback provides a hash map that contains custom mapping properties. The com.ibm.mapping.authDataAlias property name is reserved for setting the authentication data alias.

The WebSphere Application Server WSMappingCallbackHandle handle continues to support the two WebSphere Application Server Version 5.*x* callback types that older mapping login modules can use. The two callbacks defined can be used only by login modules that the login configuration uses at the connection factory. For backward compatibility, WebSphere Application Server Version 6.0.x and later passes the authentication data alias, if defined in the list of custom properties under the com.ibm.mapping.authDataAlias property name using the WSAuthDataAliasCallback callback to Version 5.*x* login modules:

```
com.ibm.ws.security.auth.j2c.WSManagedConnectionFactoryCallback
com.ibm.ws.security.auth.j2c.WSAuthDataAliasCallback
```

Invoking the login module for the connection factory mapping

The WSPrincipalMappingCallbackHandler class handles two callback types:

```
com.ibm.wsspi.security.auth.callback.WSManagedConnectionFactoryCallback
com.ibm.wsspi.security.auth.callback.WSMappingPropertiesCallback
```

The WSPrincipalMappingCallbackHandler handler and the two callbacks are deprecated in WebSphere Application Server Version 6.

Passing the mapping properties for the resource reference to the mapping login module

You can pass arbitrary custom properties to your mapping login module. The following example shows how the WebSphere Application Server default mapping login module looks for the authentication data alias property.

```
  try {
      wspm_callbackHandler.handle(callbacks);
            String userID = null;
            String password = null;
            String alias = null;
            wspm_properties = ((WSMappingPropertiesCallback)callbacks[1]).getProperties();

            if (wspm_properties != null) {
                alias = (String) wspm_properties.get(com.ibm.wsspi.security.auth.callback.
                    Constants.MAPPING_ALIAS);
                if (alias != null) {
                    alias = alias.trim();
```

```
                    }
                }
            } catch (UnsupportedCallbackException unsupportedcallbackexception) {
 . . . // error handling
```

The default mapping login module for WebSphere Application Server Version 6.0.*x* requires one mapping property to define the authentication data alias. The mapping property, which is called MAPPING_ALIAS, is defined in the Constants.class file in the com.ibm.wsspi.security.auth.callback package.

MAPPING_ALIAS   =   ″com.ibm.mapping.authDataAlias″

When you click **Use default method** > **Select authentication data entry authentication** on the Map resource references to resources panel, the administrative console automatically creates a MAPPING_ALIAS entry with the selected authentication data alias value in the mapping properties. If you create your own custom login configuration and then use the default mapping login module, you must set this property manually on the mapping properties for the resource factory reference.

In a custom login module, you can use the WSSubject.getRunAsSubject method to retrieve the subject that represents the identity of the current running thread. The identity of the current running thread is known as the *RunAs* identity. The RunAs subject typically contains a WSPrincipal principal in the principal set and a WSCredential credential in the public credential set. The subject instance that is created by your mapping module contains a Principal instance in the principals set and a PasswordCredential credential or an org.ietf.jgss.GSSCredential instance in the set of private credentials.

The GenericCredential interface that is defined in Java Cryptography Architecture (JCA) Specification Version 1.0 is removed in the JCA Version 1.5 specification. The GenericCredential interface is supported by WebSphere Application Server Version 6.0.*x* to support older resource adapters that might be programmed to the GenericCredential interface.

Customizing an application login to perform an identity assertion:

Using the Java Authentication and Authorization Service (JAAS) login framework, you can create a JAAS login configuration that can be used to perform login to an identity assertion.

You can allow an application or system provider to perform an identity assertion with trust validation. To do this, you use the JAAS login framework, where trust validation is accomplished in one login module and credential creation is accomplished in another module. The two custom login modules allow you to create a JAAS login configuration that can be used to perform a login to an identity assertion.

Two custom login modules are required:

User implemented trust association login module (trust validation)
> The user implemented trust association login module performs whatever trust verification the user requires. When trust is verified, the trust verification status and the login identity should be put into a map in the share state of the login module so that the credential creation login module can use the information. This map should be stored in the property:
>
> com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.state
> > (which consists of)
>
> com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.trusted
> > (which is set to **true** if trusted and **false** if not trusted)
>
> com.ibm.wsspi.security.common.auth.module.IdenityAssertionLoginModule.principal
> > (which contains the principal of the identity)

com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.certificates
(which contains the certificate of the identity)

Identity assertion login module (credential creation)

The com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule performs the credential creation. This module relies on the trust state information being in the login context's shared state. This login module is protected by the Java 2 security runtime permissions for:

- com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.initialize
- com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.login

The identity assertion login module looks for the trust information in the shared state property, com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.state, which contains the trust status and the identity to login and should include:

com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.trusted
(which when *true* indicates trusted and *false* when not trusted)

com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.principal
(which contains the principal of the identity to login, if using a principal)

com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.certificates
(which contains a array of a certificate chain that contains the identity to login, if using a certificate)

A WSLoginFailedException is returned if the state, trust, or identity information is missing. The login module then performs a login of the identity, and the subject will contain the new identity

1. Delegate trust validation to a user implemented plug point. Trust validation must be accomplished in a custom login module. This custom login module should perform any trust validation required, then set the trust and identity information in the shared state to be passed on to the identity assertion login module. A map is required in the shared state key, com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.state. If the state is missing then a WSLoginFailedException is thrown by the IdentityAssertionLoginModule. This map must include:

- A trust key called com.ibm.wsspi.secuirty.common.auth.module.IdentityAssertionLoginModule.trust. If the key is set to *true*, then trust is established. If the key is set to *false*, then no trust is established. If the trust key is not set to true, then the IdentityAssertionLoginModule will throw a WSLoginFailedException.
- • An identity key is set: A java.security.Principal can be set in the com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.principal key.
- Or a java.security.cert.X509Certificate[] can be set in the com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.certficates key

If both a principal and certificate are supplied, then the principal is used and a warning is issued.

2. Create a new JAAS configuration for application logins The JAAS configuration will contain the user implemented trust validation custom login module and the IdentityAssertionLoginModule. Then to configure an application login configuration, perform the following on the administration console:

   a. Expand **Security > Secure administration, applications, and infrastructure**
   b. Expand **Java authentication and authorization services > Application logins**
   c. Select **New**.
   d. Give the JAAS configuration an alias.
   e. Click **Apply**.
   f. Select **JAAS Login Modules**
   g. Select **New**.
   h. Enter the **Module class name** of the user implemented trust validation custom login module.

    i. Click **Apply**.

    j. Enter the **Module class name** of
com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule

    k. Make sure the **Module class name** classes are in the correct order. The user implemented trust validation login module should be first and the IdentityAssertionLoginModule should be the second class in the list.

    l. Click **Save**.

    This JAAS configuration is then used by the application to perform an Identity Assertion.

3. Perform the programmable identity assertion. A program can now use the JAAS login configuration to perform a programmatic identity assertion. The application program can create a login context for the JAAS configuration created in step 2, then login to that login context with the identity they would assert to. If the login is successful then that identity can be set in the current running process. Here is a example of how such code would operate:

```
MyCallbackHandler handler = new MyCallbackHandler(new MyPrincipal("Joe"));
LoginContext lc = new LoginContext("MyAppLoginConfig", handler);
lc.login();  //assume successful
Subject s = lc.getSubject();
WSSubject.setRunAsSubject(s);
// From here on , the runas identity is "Joe"
```

Using the JAAS login framework and two user implemented login modules, you can create a JAAS login configuration that can be used to perform login to an identity assertion.

Customization of a server-side Java Authentication and Authorization Service authentication and login configuration:

WebSphere Application Server supports plugging in a custom Java Authentication and Authorization Service (JAAS) login module before or after the WebSphere Application Server system login module. However, WebSphere Application Server does not support the replacement of the WebSphere Application Server system login modules, which are used to create the WSCredential credential and WSPrincipal principal in the Subject. By using a custom login module, you can either make additional authentication decisions or add information to the Subject to make additional, potentially finer-grained, authorization decisions inside a Java 2 Platform, Enterprise Edition (J2EE) application.

WebSphere Application Server enables you to propagate information downstream that is added to the Subject by a custom login module. For more information, see Security attribute propagation. To determine which login configuration to use for plugging in your custom login modules, see the descriptions of the login configurations that are located in the "System login configuration entry settings for Java Authentication and Authorization Service" on page 927.

WebSphere Application Server supports the modification of the system login configuration through the administrative console and by using the wsadmin scripting utility. To configure the system login configuration using the administrative console, click **Security > Secure administration, applications, and infrastructure**. Under Java Authentication and Authorization Service, click **System logins**.

Refer to the following code sample to configure a system login configuration using the wsadmin tool. The following sample Jacl script adds a custom login module into the Lightweight Third-party Authentication (LTPA) Web system login configuration:

Attention: Lines 32, 33, and 34 in the following code sample are split into two lines.

```
1. #######################################
2. #
3. # Open security.xml
4. #
5. #######################################
6.
```

```
7.
8. set sec [$AdminConfig getid /Cell:hillside/Security:/]
9.
10.
11. #######################################
12. #
13. # Locate systemLoginConfig
14. #
15. #######################################
16.
17.
18. set slc [$AdminConfig showAttribute $sec systemLoginConfig]
19.
20. set entries [lindex [$AdminConfig showAttribute $slc entries] 0]
21.
22.
23. #######################################
24. #
25. # Append a new LoginModule to LTPA_WEB
26. #
27. #######################################
28.
29. foreach entry $entries {
30. set alias [$AdminConfig showAttribute $entry alias]
31. if {$alias == "LTPA_WEB"} {
32.     set newJAASLoginModuleId [$AdminConfig create JAASLoginModule
         $entry {{moduleClassName
         "com.ibm.ws.security.common.auth.module.proxy.WSLoginModuleProxy"}}]
33.     set newPropertyId [$AdminConfig create Property
         $newJAASLoginModuleId {{name delegate}{value
         "com.ABC.security.auth.CustomLoginModule"}}]
34.     $AdminConfig modify $newJAASLoginModuleId
         {{authenticationStrategy REQUIRED}}
35.     break
36.     }
37. }
38.
39.
40. #######################################
41. #
42. # save the change
43. #
44. #######################################
45.
46. $AdminConfig save
47.
```

Attention: The wsadmin scripting utility inserts a new object to the end of the list. To insert the custom login module before the AuthenLoginModule login module, delete the AuthenLoginModule login module and recreate it after inserting the custom login module. Save the sample script into a `sample.jacl` file, and run the sample script using the following command:

```
wsadmin -f sample.jacl
```

You can use the following sample Jacl script to remove the current LTPA_WEB login configuration and all the login modules:

```
48. #######################################
49. #
50. # Open security.xml
51. #
52. #######################################
53.
```

```
54.
55. set sec [$AdminConfig getid /Cell:hillside/Security:/]
56.
57.
58. #######################################
59. #
60. # Locate systemLoginConfig
61. #
62. #######################################
63.
64.
65. set slc [$AdminConfig showAttribute $sec systemLoginConfig]
66.
67. set entries [lindex [$AdminConfig showAttribute $slc entries] 0]
68.
69.
70. #######################################
71. #
72. # Remove the LTPA_WEB login configuration
73. #
74. #######################################
75.
76. foreach entry $entries {
77.     set alias [$AdminConfig showAttribute $entry alias]
78.     if {$alias == "LTPA_WEB"} {
79.         $AdminConfig remove $entry
80.         break
81.     }
82. }
83.
84.
85. #######################################
86. #
87. # save the change
88. #
89. #######################################
90.
91. $AdminConfig save
```

You can use the following sample Jacl script to recover the original LTPA_WEB configuration:

Attention: Lines 122, 124, and 126 in the following code sample are split into two or more lines for illustrative purposes only.

```
92. #######################################
93. #
94. # Open security.xml
95. #
96. #######################################
97.
98.
99. set sec [$AdminConfig getid /Cell:hillside/Security:/]
100.
101.
102. #######################################
103. #
104. # Locate systemLoginConfig
105. #
106. #######################################
107.
108.
109. set slc [$AdminConfig showAttribute $sec systemLoginConfig]
110.
```

```
111. set entries [lindex [$AdminConfig showAttribute $slc entries] 0]
112.
113.
114.
115. ########################################
116. #
117. # Recreate the LTPA_WEB login configuration
118. #
119. ########################################
120.
121.
122. set newJAASConfigurationEntryId [$AdminConfig create JAASConfigurationEntry
     $slc {{alias LTPA_WEB}}]
123.
124. set newJAASLoginModuleId [$AdminConfig create JAASLoginModule
     $newJAASConfigurationEntryId
     {{moduleClassName
     "com.ibm.ws.security.common.auth.module.proxy.WSLoginModuleProxy"}}]
125.
126. set newPropertyId [$AdminConfig create Property
     $newJAASLoginModuleId {{name delegate}
     {value "com.ibm.ws.security.web.AuthenLoginModule"}}]
127.
128. $AdminConfig modify $newJAASLoginModuleId {{authenticationStrategy REQUIRED}}
129.
130.
131. ########################################
132. #
133. # save the change
134. #
135. ########################################
136.
137. $AdminConfig save
```

The WebSphere Application Server Version ltpaLoginModule and AuthenLoginModule login modules use the shared state to save state information so that custom login modules can modify the information. The ltpaLoginModule login module initializes the callback array in the login method using the following code. The callback array is created by the ltpaLoginModule login module only if an array is not defined in the shared state area. In the following code sample, the error handling code is removed to make the sample concise. If you insert a custom login module before the ltpaLoginModule login module, the custom login module might follow the same style to save the callback into the shared state.

Attention: In the following code sample, several lines of code are split into two lines for illustrative purposes only.

```
138.        Callback callbacks[] = null;
139.        if (!sharedState.containsKey(
              com.ibm.wsspi.security.auth.callback.Constants.
              CALLBACK_KEY)) {
140.          callbacks = new Callback[3];
141.          callbacks[0] = new NameCallback("Username: ");
142.          callbacks[1] = new PasswordCallback("Password: ", false);
143.          callbacks[2] = new com.ibm.websphere.security.auth.callback.
              WSCredTokenCallbackImpl( "Credential Token: ");
144.          try {
145.              callbackHandler.handle(callbacks);
146.          } catch (java.io.IOException e) {
147.              . . .
148.          } catch (UnsupportedCallbackException uce) {
149.              . . .
150.          }
151.          sharedState.put(
```

```
                    com.ibm.wsspi.security.auth.callback.Constants.CALLBACK_KEY,
                    callbacks);
152.            } else {
153.                callbacks = (Callback [])
                    sharedState.get( com.ibm.wsspi.security.auth.callback.
                    Constants.CALLBACK_KEY);
154.            }
```

The ltpaLoginModule and AuthenLoginModule login modules generate both a WSPrincipal object and a
WSCredential object to represent the authenticated user identity and security credentials. The WSPrincipal
and WSCredential objects also are saved in the shared state. A JAAS login uses a two-phase commit
protocol.

First, the login methods in login modules, which are configured in the login configuration, are called. Then,
their commit methods are called. A custom login module, which is inserted after the ltpaLoginModule and
the AuthenLoginModule login modules, can modify the WSPrincipal and WSCredential objects before
these objects are committed. The WSCredential and WSPrincipal objects must exist in the Subject after
the login is completed. Without these objects in the Subject, WebSphere Application Server run-time code
rejects the Subject to make security decisions.

AuthenLoginModule uses the following code to initialize the callback array:

Attention: In the following code sample, several lines of code are split into two lines for illustrative
purposes only.

```
155.            Callback callbacks[] = null;
156.            if (!sharedState.containsKey(
                    com.ibm.wsspi.security.auth.callback.Constants.
                    CALLBACK_KEY)) {
157.                callbacks = new Callback[6];
158.                callbacks[0] = new NameCallback("Username: ");
159.                callbacks[1] = new PasswordCallback("Password: ", false);
160.                callbacks[2] =
                    new com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl(
                    "Credential Token: ");
161.                callbacks[3] =
                    new com.ibm.wsspi.security.auth.callback.WSServletRequestCallback(
                    "HttpServletRequest: ");
162.                callbacks[4] =
                    new com.ibm.wsspi.security.auth.callback.WSServletResponseCallback(
                    "HttpServletResponse: ");
163.                callbacks[5] =
                    new com.ibm.wsspi.security.auth.callback.WSAppContextCallback(
                    "ApplicationContextCallback: ");
164.                try {
165.                    callbackHandler.handle(callbacks);
166.                } catch (java.io.IOException e) {
167.                    . . .
168.                } catch (UnsupportedCallbackException uce {
169.                    . . .
170.                }
171.                sharedState.put( com.ibm.wsspi.security.auth.callback.
                    Constants.CALLBACK_KEY, callbacks);
172.            } else {
173.                callbacks = (Callback []) sharedState.get(
                    com.ibm.wsspi.security.auth.callback.
                    Constants.CALLBACK_KEY);
174.            }
```

Three more objects, which contain callback information for the login, are passed from the Web container to
the AuthenLoginModule login module: a java.util.Map, an HttpServletRequest, and an HttpServletResponse

object. These objects represent the Web application context. The WebSphere Application Server Version 5.1 application context, java.util.Map object, contains the application name and the error page web address. You can obtain the application context, java.util.Map object, by calling the getContext method on the WSAppContextCallback object. The java.util.Map object is created with the following deployment descriptor information.

Attention: In the following code sample, several lines of code are split into two lines for illustrative purposes only.

```
175.        HashMap appContext = new HashMap(2);
176.        appContext.put(
            com.ibm.wsspi.security.auth.callback.Constants.WEB_APP_NAME,
            web_application_name);
177.        appContext.put(
            com.ibm.wsspi.security.auth.callback.Constants.REDIRECT_URL,
            errorPage);
```

The application name and the HttpServletRequest object might be read by the custom login module to perform mapping functions. The error page of the form-based login might be modified by a custom login module. In addition to the JAAS framework, WebSphere Application Server supports the trust association interface (TAI).

Other credential types and information can be added to the caller Subject during the authentication process using a custom login module. The third-party credentials in the caller Subject are managed by WebSphere Application Server as part of the security context. The caller Subject is bound to the running thread during the request processing. When a Web or an Enterprise JavaBeans (EJB) module is configured to use the caller identity, the user identity is propagated to the downstream service in an EJB request. The WSCredential credential and any third-party credentials in the caller Subject are not propagated downstream. Instead, some of the information can be regenerated at the target server based on the propagated identity. Add third-party credentials to the caller Subject at the authentication stage. The caller Subject, which is returned from the WSSubject.getCallerSubject method, is read-only and cannot be modified. For more information on the WSSubject subject, see "Example: Getting the caller subject from the thread" on page 961.

Custom login module development for a system login configuration:

For WebSphere Application Server, multiple Java Authentication and Authorization Service (JAAS) plug-in points exist for configuring system logins. WebSphere Application Server uses system login configurations to authenticate incoming requests, outgoing requests, and internal server logins.

Application login configurations are called by Java 2 Platform, Enterprise Edition (J2EE) applications for obtaining a Subject that is based on specific authentication information. This login configuration enables the application to associate the Subject with a specific protected remote action. The Subject is picked up on the outbound request processing. The following list identifies the main system plug-in points. If you write a login module that adds information to the Subject of a system login, these are the main login configurations to plug in:

- WEB_INBOUND
- RMI_OUTBOUND
- RMI_INBOUND
- DEFAULT

WEB_INBOUND login configuration

The WEB_INBOUND login configuration authenticates Web requests. Figure 1 shows an example of a configuration using a trust association interceptor (TAI) that creates a Subject with the initial information that is passed into the WEB_INBOUND login configuration. If the trust association interceptor is not

configured, the authentication process goes directly to the WEB_INBOUND system login configuration, which consists of all the login modules combined in Figure 1. Figure 1 shows where you can plug in custom login modules and where the ltpaLoginModule and the wsMapDefaultInboundLoginModule login modules are required.

Figure 1

Single application server
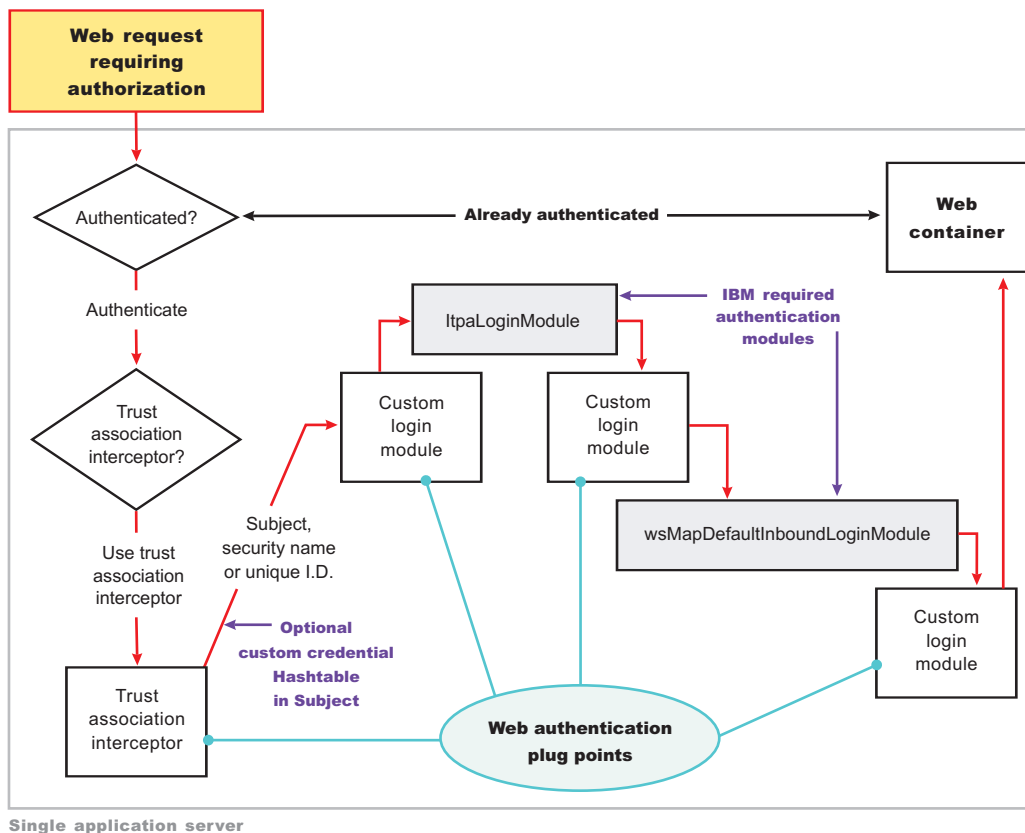
For more detailed information on the WEB_INBOUND configuration including its associated callbacks, see ″RMI_INBOUND, WEB_INBOUND, DEFAULT″ in "System login configuration entry settings for Java Authentication and Authorization Service" on page 927.

RMI_OUTBOUND login configuration

The RMI_OUTBOUND login configuration is a plug point for handling outbound requests. WebSphere Application Server uses this plug point to create the serialized information that is sent downstream based on the invocation Subject passed in and other security context information such as propagation tokens. A custom login module can use this plug point to change the identity. For more information, see Configuring outbound mapping to a different target realm. Figure 2 shows where you can plug in custom login modules and shows where the wsMapCSIv2OutboundLoginModule login module is required.

Figure 2

Outbound RMI request

Already established

Common Secure Interoperability version 2 session established

Remote enterprise bean container

RMI outbound authentication plug points

Authenticate

Custom login module

wsMapCSIv2OutoundLoginModule

Custom login module

IBM required authentication module

Possibly modified Subject and propagation attributes. Opportunity for mapping, if needed.

Single application server

For more information on the RMI_OUTBOUND login configuration, including its associated callbacks, see ″RMI_OUTBOUND″ in "System login configuration entry settings for Java Authentication and Authorization Service" on page 927.

RMI_INBOUND login configuration

The RMI_INBOUND login configuration is a plug point that handles inbound authentication for enterprise bean requests. WebSphere Application Server uses this plug point for either an initial login or a propagation login. For more information about these two login types, see Security attribute propagation. During a propagation login, this plug point is used to deserialize the information that is received from an upstream server. A custom login module can use this plug point to change the identity, handle custom tokens, add custom objects into the Subject, and so on. For more information on changing the identity using a Hashtable object, which is referenced in figure 3, see Configuring inbound identity mapping. Figure 3 shows where you can plug in custom login modules and shows that the ltpaLoginModule and the wsMapDefaultInboundLoginModule login modules are required.

Figure 3

Single application server
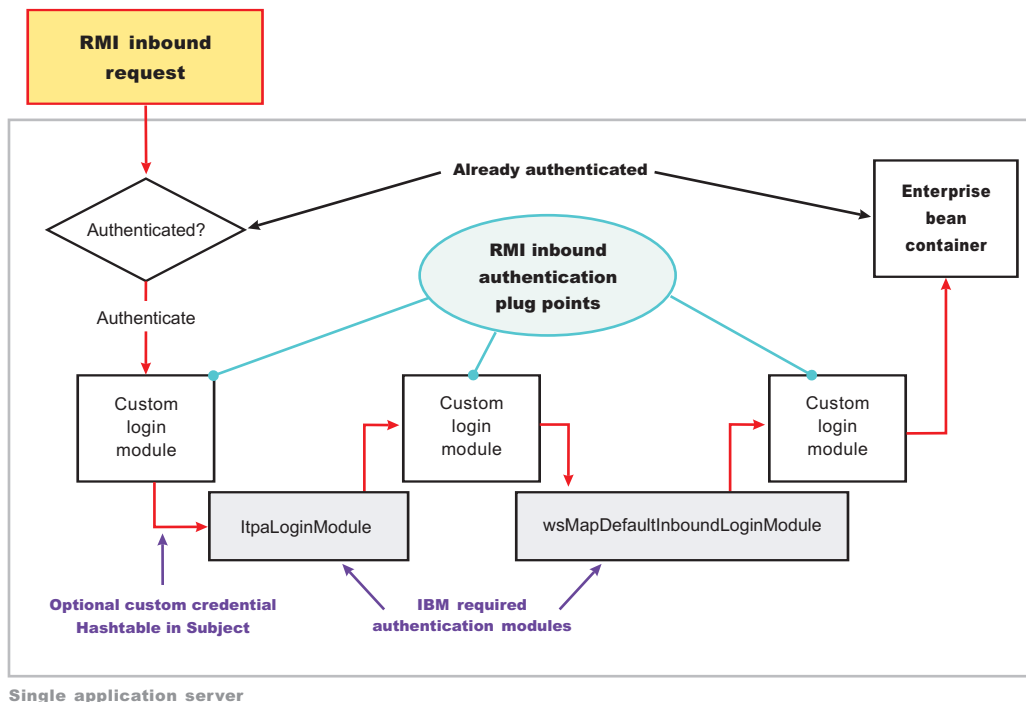
For more information on the RMI_INBOUND login configuration, including its associated callbacks, see ″RMI_INBOUND, WEB_INBOUND, DEFAULT″ in "System login configuration entry settings for Java Authentication and Authorization Service" on page 927.

DEFAULT login configuration

The DEFAULT login configuration is a plug point that handles all of the other types of authentication requests, including administrative SOAP requests and internal authentication of the server ID. Propagation logins typically do not occur at this plug point.

For more information on the DEFAULT login configuration including its associated callbacks, see ″RMI_INBOUND, WEB_INBOUND, DEFAULT″ in "System login configuration entry settings for Java Authentication and Authorization Service" on page 927.

Writing a login module

When you write a login module that plugs into a WebSphere Application Server application login or system login configuration, read the JAAS programming model, which is located at: http://java.sun.com/products/ jaas. The JAAS programming model provides basic information about JAAS. However, before writing a login module for the WebSphere Application Server environment, read the following sections in this article:
- Useable callbacks
- Shared state variables
- Initial versus propagation logins
- Sample custom login module

Useable callbacks

Each login configuration must document the callbacks that are recognized by the login configuration. However, the callbacks are not always passed data. The login configuration must contain logic to know when specific information is present and how to use the information. For example, if you write a custom login module that can plug into all four of the pre-configured system login configurations mentioned

previously, three sets of callbacks might be presented to authenticate a request. Other callbacks might be present for other reasons, including propagation and making other information available to the login configuration.

Login information can be presented in the following combinations:

User name (NameCallback) and password (PasswordCallback)
> This information is a typical authentication combination.

User name only (NameCallback)
> This information is used for identity assertion, trust association interceptor (TAI) logins, and certificate logins.

Token (WSCredTokenCallbackImpl)
> This information is for Lightweight Third Party Authentication (LTPA) token validation.

Propagation token list (WSTokenHolderCallback)
> This information is used for a propagation login.

The first three combinations are used for typical authentication. However, when the WSTokenHolderCallback callback is present in addition to one of the first three information combinations, the login is called a *propagation login*. A propagation login means that some security attributes are propagated to this server from another server. The servers can reuse these security attributes if the authentication information validates successfully. In some cases, a WSTokenHolderCallback callback might not have sufficient attributes for a full login. Check the requiresLogin method on the WSTokenHolderCallback callback to determine if a new login is required. You can always ignore the information returned by the requiresLogin method, but, as a result, you might duplicate information. The following list contains the callbacks that might be present in the system login configurations. The list includes the callback name and a description of their responsibility.

callbacks[0] = new javax.security.auth.callback.NameCallback(″Username: ″);
> This callback handler collects the user name for the login. The result can be the user name for a basic authentication login (user name and password) or a user name for an identity assertion login.

callbacks[1] = new javax.security.auth.callback.PasswordCallback(″Password: ″, false);
> This callback handler collects the password for the login.

callbacks[2] = new com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl(″Credential Token: ″);
> This callback handler collects the Lightweight Third Party Authentication (LTPA) token or other token type for the login. This callback handler is typically present when a user name and password are not present.

callbacks[3] = new com.ibm.wsspi.security.auth.callback.WSTokenHolderCallback(″Authz Token List: ″);
> This callback handler collects the ArrayList of TokenHolder objects that are returned from a call to the WSOpaqueTokenHelper.createTokenHolderListFromOpaqueToken API using the Common Secure Interoperability Version 2 (CSIv2) authorization token as input.

callbacks[4] = new com.ibm.websphere.security.auth.callback.WSServletRequestCallback(″HttpServletRequest: ″);
> This callback handler collects the HTTP servlet request object, if present. This callback handler enables login modules to get information from the HTTP request for use in the login, and is presented from the WEB_INBOUND login configuration only.

callbacks[5] = new com.ibm.websphere.security.auth.callback.WSServletResponseCallback(″HttpServletResponse: ″);
> This callback handler collects the HTTP servlet response object, if present. This callback handler enables login modules to put information into the HTTP response as a result of the login. An

example of this situation might be adding the SingleSignonCookie cookie to the response.This callback handler is presented from the WEB_INBOUND login configuration only.

callbacks[6] = new com.ibm.websphere.security.auth.callback.WSAppContextCallback(″**ApplicationContextCallback:** ″**);**

> This callback handler collects the Web application context that is used during the login. This callback handler consists of a HashMap object, which contains the application name and the redirect web address, if present. The callback handler is presented from the WEB_INBOUND login configuration only.

callbacks[7] = new WSRealmNameCallbackImpl(″**Realm Name:** ″**,** *default_realm***);**

> This callback handler collects the realm name for the login information. The realm information might not always be provided. If the realm information is not provided, assume that it is the current realm.

callbacks[8] = new WSX509CertificateChainCallback(″**X509Certificate[]:** ″**);**

> This callback handler contains the certificate that was validated by Secure Sockets Layer (SSL) if the login source is an X509Certificate from SSL client authentication. The ltpaLoginModule calls the same mapping functions as WebSphere Application Server releases prior to version 6.1. However, having it passed into the login gives a custom login module the opportunity to map the certificate in a custom way. Then, it performs a Hashtable login. See Configuring inbound identity mapping for more information on a Hashtable login.

Shared state variables

Shared state variables are used to share information between login modules during the login phase. The following list contains recommendations for using the shared state variables:

- When you have a custom login module, use the shared state variables to communicate to a WebSphere Application Server login module using a documented shared state variable, as shown in the following table.
- Try not to update the Subject until the commit phase. If you call the abort method, you must remove any objects added to the Subject.
- Enable the login module that adds information into the shared state Map during login to remove this information during commit in case the same shared state is used for another login.
- If a stop or logout occurs, clean up the information in the login configuration for the shared state and the Subject.

The com.ibm.wsspi.security.token.AttributeNameConstants.WSCREDENTIAL_PROPERTIES_KEY shared state variable can inform the WebSphere Application Server login configurations about asserted privilege attributes. This variable references the com.ibm.wsspi.security.cred.propertiesObject property. Associate a java.util.Hashtable with this property. This hashtable contains properties that are used by WebSphere Application Server for login purposes and ignores the callback information. This hashtable enables a custom login module, which is carried out first in the login configuration to map user identities or enable WebSphere Application Server to avoid making unnecessary user registry calls if you already have the required information. For more information, see Configuring inbound identity mapping.

If you want to access the objects that WebSphere Application Server creates during a login, refer to the following shared state variables. The variables are set in the following login modules: ltpaLoginModule, swamLoginModule, and wsMapDefaultInboundLoginModule.

Shared state variable

> com.ibm.wsspi.security.auth.callback.Constants.WSPRINCIPAL_KEY

Purpose

> Specifies the com.ibm.websphere.security.auth.WSPrincipal object. See the WebSphere

Application Server API documentation for application programming interface (API) usage. This shared state variable is for read-only purposes. Do not set this variable in the shared state for custom login modules.

The login module in which variables are set
ltpaLoginModule, swamLoginModule, and wsMapDefaultInboundLoginModule

Shared state variable
com.ibm.wsspi.security.auth.callback.Constants.WSCREDENTIAL_KEY

Purpose
Specifies the com.ibm.websphere.security.cred.WSCredential object. See the WebSphere Application Server API documentation for API usage. This shared state variable is for read-only purposes. Do not set this variable in the shared state for custom login modules.

Login module in which variables are set
wsMapDefaultInboundLoginModule

Shared state variable
com.ibm.wsspi.security.auth.callback.Constants.WSAUTHZTOKEN_KEY

Purpose
Specifies the default com.ibm.wsspi.security.token.AuthorizationToken object. Login modules can use this object to set custom attributes plugged in after the wsMapDefaultInboundLoginModule login module. The information set here is propagated downstream and is available to the application. See the WebSphere Application Server API documentation for API usage.

Initial versus propagation logins

As mentioned previously, some logins are considered initial logins because of the following reasons:
- It is the first time authentication information is presented to WebSphere Application Server.
- The login information is received from a server that does not propagate security attributes so this information must be gathered from a user registry.

Other logins are considered propagation logins when a WSTokenHolderCallback callback is present and contains sufficient information from a sending server to recreate all the required objects needed by WebSphere Application Server runtime. In cases where there is sufficient information for the WebSphere Application Server runtime, the information you might add to the Subject is likely to exist from the previous login. To verify if your object is present, you can get access to the ArrayList object that is present in the WSTokenHolderCallback callback, and search through this list looking at each TokenHolder getName method. This search is used to determine if WebSphere Application Server is deserializing your custom object during this login. Check the class name returned from the getName method using the String startsWith method because the runtime might add additional information at the end of the name to know which Subject is set to add the custom object after deserialization.

The following code snippet can be used in your login() method to determine when sufficient information is present. For another example, see Configuring inbound identity mapping.

```
// This is a hint provided by WebSphere Application Server that
// sufficient propagation information does not exist and, therefore,
// a login is required to provide the sufficient information. In this
// situation, a Hashtable login might be used.
boolean requiresLogin = ((com.ibm.wsspi.security.auth.callback.
WSTokenHolderCallback) callbacks[1]).requiresLogin();

if (requiresLogin)
{
// Check to see if your object exists in the TokenHolder list,
if not, add it.
java.util.ArrayList authzTokenList = ((WSTokenHolderCallback) callbacks[6]).
getTokenHolderList();boolean found = false;
```

```
if (authzTokenList != null)
{
Iterator tokenListIterator = authzTokenList.iterator();

while (tokenListIterator.hasNext())
{
com.ibm.wsspi.security.token.TokenHolder th = (com.ibm.wsspi.security.token.
TokenHolder) tokenListIterator.next();

if (th != null && th.getName().startsWith("com.acme.myCustomClass"))
{
found=true;
break;
}
}
if (!found)
{
// go ahead and add your custom object.
}
}
}
else
{
// This code indicates that sufficient propagation information is present.
// User registry calls are not needed by WebSphere Application Server to
// create a valid Subject. This code might be a no-op in your login module.
}
```

Sample custom login module

You can use the following sample to get ideas on how to use some of the callbacks and shared state variables.

```
{
 // Defines your login module variables
 com.ibm.wsspi.security.token.AuthenticationToken customAuthzToken = null;
 com.ibm.wsspi.security.token.AuthenticationToken defaultAuthzToken = null;
 com.ibm.websphere.security.cred.WSCredential credential = null;
 com.ibm.websphere.security.auth.WSPrincipal principal = null;
 private javax.security.auth.Subject _subject;
 private javax.security.auth.callback.CallbackHandler _callbackHandler;
 private java.util.Map _sharedState;
 private java.util.Map _options;

 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _subject = subject;
  _callbackHandler = callbackHandler;
  _sharedState = sharedState;
  _options = options;
 }

 public boolean login() throws LoginException
 {
  boolean succeeded = true;

  // Gets the CALLBACK information
  javax.security.auth.callback.Callback callbacks[] = new javax.security.
      auth.callback.Callback[7];
  callbacks[0] = new javax.security.auth.callback.NameCallback(
      "Username: ");
  callbacks[1] = new javax.security.auth.callback.PasswordCallback(
      "Password: ", false);
  callbacks[2] = new com.ibm.websphere.security.auth.callback.
      WSCredTokenCallbackImpl ("Credential Token: ");
```

```
callbacks[3] = new com.ibm.wsspi.security.auth.callback.
      WSServletRequestCallback ("HttpServletRequest: ");
callbacks[4] = new com.ibm.wsspi.security.auth.callback.
      WSServletResponseCallback ("HttpServletResponse: ");
callbacks[5] = new com.ibm.wsspi.security.auth.callback.
      WSAppContextCallback ("ApplicationContextCallback: ");
callbacks[6] = new com.ibm.wsspi.security.auth.callback.
       WSTokenHolderCallback ("Authz Token List: ");

try
{
 callbackHandler.handle(callbacks);
}
catch (Exception e)
{
 // Handles exceptions
 throw new WSLoginFailedException (e.getMessage(), e);
}

// Sees which callbacks contain information
uid = ((NameCallback) callbacks[0]).getName();
char password[] = ((PasswordCallback) callbacks[1]).getPassword();
byte[] credToken = ((WSCredTokenCallbackImpl) callbacks[2]).getCredToken();
javax.servlet.http.HttpServletRequest request = ((WSServletRequestCallback)
      callbacks[3]).getHttpServletRequest();
javax.servlet.http.HttpServletResponse response = ((WSServletResponseCallback)
      callbacks[4]).getHttpServletResponse();
java.util.Map appContext = ((WSAppContextCallback)
      callbacks[5]).getContext();
java.util.List authzTokenList = ((WSTokenHolderCallback)
      callbacks[6]).getTokenHolderList();

// Gets the SHARED STATE information
principal = (WSPrincipal) _sharedState.get(com.ibm.wsspi.security.
      auth.callback.Constants.WSPRINCIPAL_KEY);
credential = (WSCredential) _sharedState.get(com.ibm.wsspi.security.
      auth.callback.Constants.WSCREDENTIAL_KEY);
defaultAuthzToken = (AuthorizationToken) _sharedState.get(com.ibm.
      wsspi.security.auth.callback.Constants.WSAUTHZTOKEN_KEY);

   // What you tend to do with this information depends upon the scenario
   // that you are trying to accomplish. This example demonstrates how to
   // access various different information:
   // - Determine if a login is initial versus propagation
   // - Deserialize a custom authorization token (For more information, see
   //   Security attribute propagation
   // - Add a new custom authorization token (For more information, see
   //   Security attribute propagation
   // - Look for a WSCredential and read attributes, if found.
   // - Look for a WSPrincipal and read attributes, if found.
   // - Look for a default AuthorizationToken and add attributes, if found.
   // - Read the header attributes from the HttpServletRequest, if found.
   // - Add an attribute to the HttpServletResponse, if found.
   // - Get the Web application name from the appContext, if found.

   // - Determines if a login is initial versus propagation. This is most
   //   useful when login module is first.
boolean requiresLogin = ((WSTokenHolderCallback) callbacks[6]).requiresLogin();

// initial login - asserts privilege attributes based on user identity
if (requiresLogin)
{

 // If you are validating a token from another server, there is an
    // application programming interface  (API) to get the uniqueID from it.
 if (credToken != null && uid == null)
 {
```

```
  try
  {
   String uniqueID = WSSecurityPropagationHelper.
             validateLTPAToken(credToken);
   String realm = WSSecurityPropagationHelper.getRealmFromUniqueID
               (uniqueID);
          // Now set it to the UID so you can use that to either map or
          // login with.
   uid = WSSecurityPropagationHelper.getUserFromUniqueID (uniqueID);
  }
  catch (Exception e)
  {
   // handle exception
  }
}
    // Adds a Hashtable to shared state.
    // Note: You can perform custom mapping on the NameCallback value returned
    // to change the identity based upon your own mapping rules.
uid = mapUser (uid);

// Gets the default InitialContext for this server.
javax.naming.InitialContext ctx = new javax.naming.InitialContext();

// Gets the local UserRegistry object.
com.ibm.websphere.security.UserRegistry reg = (com.ibm.websphere.security.
        UserRegistry) ctx.lookup("UserRegistry");

    // Gets the user registry uniqueID based on the uid specified in the
    // NameCallback.
String uniqueid = reg.getUniqueUserId(uid);
 uid = WSSecurityPropagationHelper.getUserFromUniqueID (uniqueID);

// Gets the display name from the user registry based on the uniqueID.
String securityName = reg.getUserSecurityName(uid);

// Gets the groups associated with this uniqueID.
java.util.List groupList = reg.getUniqueGroupIds(uid);

    // Creates the java.util.Hashtable with the information you gathered from
    // the UserRegistry.
java.util.Hashtable hashtable = new java.util.Hashtable();
hashtable.put(com.ibm.wsspi.security.token.AttributeNameConstants.
        WSCREDENTIAL_UNIQUEID, uniqueid);
hashtable.put(com.ibm.wsspi.security.token.AttributeNameConstants.
        WSCREDENTIAL_SECURITYNAME, securityName);
hashtable.put(com.ibm.wsspi.security.token.AttributeNameConstants.
        WSCREDENTIAL_GROUPS, groupList);

    // Adds a cache key that is used as part of the lookup mechanism for
    // the created Subject. The cache key can be an Object, but should
    // implement the toString() method. Make sure the cacheKey contains
    // enough information to scope it to the user and any additional
    // attributes that you use. If you do not specify this property the
    // Subject is scoped to the WSCREDENTIAL_UNIQUEID returned, by default.
hashtable.put(com.ibm.wsspi.security.token.AttributeNameConstants.
        WSCREDENTIAL_CACHE_KEY,
   "myCustomAttribute" + uniqueid);

  // Adds the hashtable to the sharedState of the Subject.
  _sharedState.put(com.ibm.wsspi.security.token.AttributeNameConstants.
          WSCREDENTIAL_PROPERTIES_KEY,hashtable);
}
// propagation login - process propagated tokens
else
{
 // - Deserializes a custom authorization token. For more information, see
     //     Security attribute propagation.
```

```
      //     This can be done at any login module plug in point (first,
      //       middle, or last).
  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 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());

    }
   }
  }
      // - Adds a new custom authorization token (For more information,
      //   see Security attribute propagation)
      //     This can be done at any login module plug in point (first, middle,
      //     or last).
  else
  {
   // Gets the PRINCIPAL from the default AuthenticationToken. This must
         // match all of the 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");
   }
  }
 }

// - Looks for a WSCredential and read attributes, if found.
//   This is most useful when plugged in as the last login module.
if (credential != null)
{
 try
 {
  // Reads some data from the credential
  String securityName = credential.getSecurityName();
  java.util.ArrayList = credential.getGroupIds();
 }
 catch (Exception e)
 {
  // Handles exceptions
  throw new WSLoginFailedException (e.getMessage(), e);
 }
```

```
}

// - Looks for a WSPrincipal and read attributes, if found.
//   This is most useful when plugged as the last login module.
if (principal != null)
{
 try
 {
  // Reads some data from the principal
  String principalName = principal.getName();
 }
 catch (Exception e)
 {
  // Handles exceptions
  throw new WSLoginFailedException (e.getMessage(), e);
 }
}

// - Looks for a default AuthorizationToken and add attributes, if found.
//   This is most useful when plugged in as the last login module.
if (defaultAuthzToken != null)
{
 try
 {
  // Reads some data from the defaultAuthzToken
  String[] myCustomValue = defaultAuthzToken.getAttributes ("myKey");
  // Adds some data if not present in the defaultAuthzToken
  if (myCustomValue == null)
   defaultAuthzToken.addAttribute ("myKey", "myCustomData");
 }
 catch (Exception e)
 {
  // Handles exceptions
  throw new WSLoginFailedException (e.getMessage(), e);
 }
}

// - Reads the header attributes from the HttpServletRequest, if found.
//  This can be done at any login module plug in point (first, middle,
   //  or last).
if (request != null)
{
 java.util.Enumeration headerEnum = request.getHeaders();
 while (headerEnum.hasMoreElements())
 {
  System.out.println ("Header element: " + (String)headerEnum.nextElement());
 }
}

// - Adds an attribute to the HttpServletResponse, if found
   //  This can be done at any login module plug in point (first, middle,
   //  or last).
if (response != null)
{
 response.addHeader ("myKey", "myValue");
}

// - Gets the Web application name from the appContext, if found
//   This can be done at any login module plug in point (first, middle,
   //   or last).
if (appContext != null)
{
 String appName = (String) appContext.get(com.ibm.wsspi.security.auth.
       callback.Constants.WEB_APP_NAME);
}

 return succeeded;
```

```
}

public boolean commit() throws LoginException
{
 boolean succeeded = true;

 // Add any objects here that you have created and belong in the
 // Subject. Make sure the objects are not already added. If you added
 // any sharedState variables, remove them before you exit. If the abort()
    // method gets called, make sure you cleanup anything added to the
 // Subject here.

 if (customAuthzToken != null)
 {
  // Sets the customAuthzToken token into the Subject
  try
  {
   // 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);
  }
 }

 return succeeded;
}

public boolean abort() throws LoginException
{
 boolean succeeded = true;

 // Makes sure to remove all objects that have already been added (both into the
    // Subject and shared state).

 if (customAuthzToken != null)
 {
  // remove the customAuthzToken token from the Subject
  try
  {
   final AuthorizationToken customAuthzTokenPriv = customAuthzToken;
   // Do this in a doPrivileged block so that application code does not need
        // to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
```

```
  {
   public Object run()
   {
    try
    {
     // Removes the custom authorization token if it is not
              // null and not already in the Subject
                  if ((customAuthzTokenPriv != null) &&
                   (_subject.getPrivateCredentials().
                     contains(customAuthzTokenPriv)))
     {
      _subject.getPrivateCredentials().
                remove(customAuthzTokenPriv);
     }
    }
    catch (Exception e)
    {
     throw new WSLoginFailedException (e.getMessage(), e);
    }

    return null;
   }
  });
 }
 catch (Exception e)
 {
  throw new WSLoginFailedException (e.getMessage(), e);
 }
}

 return succeeded;
}


public boolean logout() throws LoginException
{
 boolean succeeded = true;

 // Makes sure to remove all objects that have already been added
    // (both into the Subject and shared state).

 if (customAuthzToken != null)
 {
  // Removes the customAuthzToken token from the Subject
  try
  {
   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
    {
     // Removes the custom authorization token if it is not null and not
              //  already in the Subject
                  if ((customAuthzTokenPriv != null) && (_subject.
                  getPrivateCredentials().
                  contains(customAuthzTokenPriv)))
     {
      _subject.getPrivateCredentials().remove(customAuthzTokenPriv);
     }
    }
    catch (Exception e)
    {
```

```
        throw new WSLoginFailedException (e.getMessage(), e);
      }

      return null;
    }
  });
}
catch (Exception e)
{
 throw new WSLoginFailedException (e.getMessage(), e);
}
}

  return succeeded;

 }

}
```

After developing your custom login module for a system login configuration, you can configure the system login using either the administrative console or using the wsadmin utility. To configure the system login using the administrative console, click **Security > Secure administration, applications, and infrastructure**. Under Java Authentication and Authorization Service, click **System logins**. For more information on using the wsadmin utility for system login configuration, see "Customization of a server-side Java Authentication and Authorization Service authentication and login configuration" on page 942. Also refer to the "Customization of a server-side Java Authentication and Authorization Service authentication and login configuration" on page 942 article for information on system login modules and to determine whether to add additional login modules.

Example: Getting the caller subject from the thread:

The Caller subject (or "received subject") contains the user authentication information that is used in the call for this request. This subject is returned after issuing the WSSubject.getCallerSubject application programming interface (API) to prevent replacing existing objects. The subject is marked read-only. This API can be used to get access to the WSCredential credential so that you can put or set data in the hashmap within the credential.

Most data within the subject is not propagated downstream to another server. Only the credential token within the WSCredential credential is propagated downstream and a new caller subject is generated.

```
try
{
 javax.security.auth.Subject caller_subject;
 com.ibm.websphere.security.cred.WSCredential caller_cred;

 caller_subject = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();

 if (caller_subject != null)
 {
  caller_cred = caller_subject.getPublicCredentials
      (com.ibm.websphere.security.cred.WSCredential.class).iterator().next();
  String CALLERDATA = (String) caller_cred.get ("MYKEY");
  System.out.println("My data from the Caller credential is:  " + CALLERDATA);
 }
}
catch (WSSecurityException e)
{
 // log error
}
```

```
catch (Exception e)
{
 // log error
}
```

Requirement: You need the following Java 2 security permissions to runthis API: `permission`
`javax.security.auth.AuthPermission "wssecurity.getCallerSubject;"`.

Example: Getting the RunAs subject from the thread:

The RunAs subject or invocation subject contains the user authentication information for the RunAs mode
set in the application deployment descriptor for this method.

The RunAs subject (or invocation subject) contains the user authentication information for the RunAs
mode set in the application deployment descriptor for this method. This subject is marked read-only when
returned from the WSSubject.getRunAsSubject application programming interface (API) to prevent
replacing existing objects. You can use this API to get access to the WSCredential credential, which is
documented in the API documentation, so that you can put or set data in the hashmap within the
credential.

Most data within the Subject is not propagated downstream to another server. Only the credential token
within the WSCredential credential is propagated downstream and a new Caller subject is generated.

```
try
{
 javax.security.auth.Subject runas_subject;
 com.ibm.websphere.security.cred.WSCredential runas_cred;

 runas_subject = com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

 if (runas_subject != null)
 {
  runas_cred = runas_subject.getPublicCredentials(
                com.ibm.websphere.security.cred.WSCredential.class).iterator().next();
  String RUNASDATA = (String) runas_cred.get ("MYKEY");
  System.out.println("My data from the RunAs credential is:  " + RUNASDATA );
 }
}
catch (WSSecurityException e)
{
 // log error
}
catch (Exception e)
{
 // log error
}
```

Requirements: You need the Java 2 security permissions to run this API: permission
javax.security.auth.AuthPermission "wssecurity.getRunAsSubject;".

Example: Overriding the RunAs subject on the thread:

To extend the function that is provided by the Java Authentication and Authorization Service (JAAS)
application programming interfaces (APIs), you can set the RunAs subject or invocation subject with a
different valid entry that is used for outbound requests on this running thread.

This extension gives you the flexibility to associate the Subject with all the remote calls on this thread
whether you use a WSSubject.doAs method to associate the subject with the remote action. For example:

```
try
{
javax.security.auth.Subject runas_subject, caller_subject;

 runas_subject = com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();
 caller_subject = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();

 // set a new RunAs subject for the thread, overriding the one declaratively set
 com.ibm.websphere.security.auth.WSSubject.setRunAsSubject(caller_subject);

 // do some remote calls

 // restore back to the previous runAsSubject
 com.ibm.websphere.security.auth.WSSubject.setRunAsSubject(runas_subject);
}
catch (WSSecurityException e)
{
 // log error
}
catch (Exception e)
{
 // log error
}
```

You need the following Java 2 security permissions to run these APIs:

- permission javax.security.auth.AuthPermission ″wssecurity.getRunAsSubject″;
- permission javax.security.auth.AuthPermission ″wssecurity.getCallerSubject″;
- permission javax.security.auth.AuthPermission ″wssecurity.setRunAsSubject″;

Example: User revocation from a cache:

In WebSphere Application Server, Version 5.0.2 and later, revocation of a user from the security cache using an MBean interface is supported.

This procedure can be called from another JACL script. The following Java Command Language (JACL) revokes a user when given the realm and the user ID, and cycles through all the security administration MBean instances that are returned for the entire cell when run from the deployment manager wsadmin command. The command also purges the user from the cache during each process.

Note: When a user is removed from authentication cache, the user can still login to WebSphere Application Server at any time. Removing the cache only removes the user from the runtime cache. It does not remove the user from registry, nor does it lock out the user.

Attention: In some of the following lines of code, the lines are split into two or more lines for illustrative purposes only.

Enabling identity assertion with trust validation:

By enabling identity assertion with trust validation, an application can use the JAAS login configuration to perform a programmatic identity assertion.

To enable an identity assertion with trust validation, follow these steps:
1. Create a custom login module to perform a trust validation. The login module must set trust and identity information in the shared state, which is then passed on to the IdentityAssertionLoginModule. The trust and identity information is stored in a map in the shared state under the key,

com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.state. If this key is missing from the shared state, a WSLoginFailedException error is thrown by the IdentityAssertionLoginModule module. The custom login module should include the following:

- A trust key named com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.trust. If the trust key is set to `true`, trust is established. If the trust key is set to `false`, the IdentityAssertionLoginModule module creates a WSLoginFailedException error.
- The identity of the java.security.Principal type set in the com.ibm.wsspi.security.common.auth.module.IdenityAssertionLoginModule.principal key.
- The identity in the form of a java,security.cert.X509Certificate[] certificate set in the com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule.certificates key.

Note: If both a principal and a certificate are supplied, the principal is used, and a warning is issued.

2. Create a new Java Authentication and Authorization Service (JAAS) configuration for application logins. It contains the user-implemented trust validation custom login module and the IdentityAssertionLoginModule module. To configure an application login configuration from the administrative console, complete the following steps:

   a. Click **Security > Secure administration, applications, and infrastructure**.
   b. Under Java Authentication and Authorization Service, click **Application logins > New**.
   c. Supply the JAAS configuration with an alias, and then click **Apply**.
   d. Under Additional properties, click **JAAS Login Modules > New**.
   e. Enter the module class name of the user-implemented trust validation custom login module, and then click **Apply**.
   f. Enter the com.ibm.wsspi.security.common.auth.module.IdentityAssertionLoginModule module class name.
   g. Make sure that the module class name classes are in the correct order. The user-implemented trust validation login module must be the first class in the list, and the IdentityAssertionLoginModule module must be the second class.
   h. Click **Save**. The new JAAS configuration is used by the application to perform an identity assertion.

An application can now use the JAAS login configuration to perform a programmatic identity assertion. The application can create a login context for the JAAS configuration created in step 2, then login to that login context with the identity it asserts to. If the login is successful, that identity can be set in the current running process, as in the following example:

```
MyCallbackHandler handler = new MyCallbackHandler(new MyPrincipal("Joe"));
LoginContext lc = new LoginContext("MyAppLoginConfig", handler);
lc.login();  //assume successful
Subject s = lc.getSubject();
WSSubject.setRunAsSubject(s);
// From here on, the runas identity is "Joe"
```

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 5 API documentation that is located at `http://java.sun.com/j2se/1.5.0/docs/api/index.html`.

Several JSSE providers ship with the J2SDK 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 6.1. These transports take advantage of the J2SDK Verison 5 network input/output (NIO) asynchronous channels.

The HTTP and JMS transports use a new IBMJSSE2 provider. All other transports in WebSphere Application Server Version 6.*x* currently use the IBMJSSE2 provider, but can be switched to the old IBMJSSE provider, if necessary (specified in the SSL repertoire configuration).

For more information on the new IBMJSSE2 provider, please review the documentation located in the `http://www.ibm.com/developerworks/java/jdk/security/142/jsse2docs.zip` file. After it is unzipped, the JSSE2 Reference Guide can be found at `jsse2Docs/JSSE2RefGuide.html`, the JSSE2 API documentation can be found at `jsse2Docs/api/index.html` and finally, the JSSE2 samples can be found at `jsse2Docs/samples`.

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	IBMJSSE	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, which is located in the *app_server_root*/java/jre/lib/ security directory. The security property is propertyName=propertyValue. Dynamically set the Java security property by calling the java.security.Security.setProperty method in your code.

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

You can find more information documented for the JSSE APIs if you access the following information:

Version 1.4.2

1. Access the http://www.ibm.com/developerworks/java/jdk/security/ Web site.
2. Click **Java 1.4.2**.
3. Click **Javadoc HTML documentation** in the Java Secure Socket Extension (JSSE) Guide section.

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 1.4.2

1. Access the http://www.ibm.com/developerworks/java/jdk/security/ Web site.
2. Click **Java 1.4.2**.
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, provided that the debug version of JSSE is installed.

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/142/.

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

There is more information documented for the JCE APIs on the `http://www.ibm.com/developerworks/java/jdk/security/` Web site.

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.

Configuring Federal Information Processing Standard Java Secure Socket Extension files:

Use this topic to configure Federal Information Processing Standard Java Secure Socket Extension files.

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 6.1 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 6.1Version 6.1 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 insert the IBMJCEFIPS provider (com.ibm.crypto.fips.provider.IBMJCEFIPS) before the IBMJCE provider, and also renumber the other providers in 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.

   The IBM SDK `java.security` file looks like the following example after completing this step:

   ```
   security.provider.1=com.ibm.crypto.fips.provider.IBMJCEFIPS
   security.provider.2=com.ibm.crypto.provider.IBMJCE
   security.provider.3=com.ibm.jsse.IBMJSSEProvider
   security.provider.4=com.ibm.jsse2.IBMJSSEProvider2
   security.provider.5=com.ibm.security.jgss.IBMJGSSProvider
   security.provider.6=com.ibm.security.cert.IBMCertPath
   security.provider.7=com.ibm.crypto.pkcs11.provider.IBMPKCS11
   security.provider.8=com.ibm.security.cmskeystore.CMSProvider
   security.provider.9=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
   ```

   If you are using the Sun JDK, the java.security file looks like the following example after completing this step:

```
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
```

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.

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.

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.

   Tip: 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 972.
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 1006. 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 977 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 customer 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 token. 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 "Custom login module development for a system login configuration" on page 947

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 971.

```
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)
```

```java
    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
     // "Custom login module development for a system login configuration" on page 947.)
 }

 public boolean login() throws LoginException
 {
  // (For more information on what to do during login, see
     // "Custom login module development for a system login configuration" on page 947.)

  // 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
    // "Custom login module development for a system login configuration" on page 947
 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.

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.

Tip: 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"

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 1006. 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 1006. 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 985 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 "Custom login module development for a system login configuration" on page 947

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 979.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
```

980   Developing and deploying applications

```
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 "Custom login module development for a system login configuration" on page 947.

```
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.

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.

   Tip: 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 989

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 994, 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.

   Restriction:
   - HTTP cookies have a size limitation so do not add too much data to this token.
   - 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 996 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 "Custom login module development for a system login configuration" on page 947.

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 987.

```
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)
 {
```

Chapter 4. Developing WebSphere applications   **989**

```
  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 "Custom login module development for a system login configuration" on page 947.

```
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 "Custom login module development for a system login configuration" on page 947.

```
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.

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).

Important: 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.

Tip: 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 1006. 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 1004, 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 "Custom login module development for a system login configuration" on page 947.

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 "Custom login module development for a system login configuration" on page 947.

```
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.

Prior to completing this task, verify that security propagation is enabled in the administrative console.

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.

Important: Whenever you plug a custom login module into the login infrastructure of WebSphere Application Server, make sure that the code is trusted. When you add the login module into the *install_root*/classes 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 *install_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 "Custom login module development for a system login configuration" on page 947.

```
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 Default authorization token.

   PropagationToken
   > Add attributes that are specific to an invocation. For more information, see 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, make sure that the class definition for that custom object exists in the `install_root`/classes 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 "Custom login module development for a system login configuration" on page 947.

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 > Secure administration, applications, and infrastructure > 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.

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.

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.

WebSphere Application Server version 4 and WebSphere Application Server version 5.*x* support the com.ibm.websphere.security.TrustAssociationInterceptor.java interface described in this article. WebSphere Application Server version 6.0 and later support the com.ibm.wsspi.security.tai.TrustAssociationInterceptor interface described in "Developing a custom trust association interceptor" on page 140.

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 *app_server_root*/properties directory, set the following properties:
>> - com.ibm.websphere.security.trustassociation.types = myTAItype
>> - com.ibm.websphere.security.trustassociation.myTAItype.config = *app_server_root*/properties/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*/properties/myTAI.properties
>>
>> Type the previous code as one continuous line.

The location of the properties file is fully qualified (for example, *app_server_root*/properties/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 *app_server_root*/properties directory, then set the following properties:

3. Compile the implementation once you have implemented it. For example, *app_server_root*/java/bin/javac -classpath *install_root*/lib/wssec.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).

      This file needs to be copied to each node class path and each cell class path.

   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.

Refer to the Security: Resources for Learning article, which references the WebSphere Application Server version 5 Redbook to view 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.
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.

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.

Enabling custom password encryption:
To view an example code sample that illustrates the com.ibm.wsspi.security.crypto.CustomPasswordEncryption interface, see Plug point for custom password encryption.

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:

- **property 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. 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`

3. 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.

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 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.

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.

The Naming service is based on the Java Naming and Directory Interface (JNDI) 1.2.1 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 name space 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 may 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 1028.

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 1023.

5. **Optional:** If your application must access applications in other cells, configure foreign cell bindings for the other cells.

6. Configure name space bindings. This step is necessary in these cases:
   - Your deployed application is to be accessed by legacy client applications running on previous versions of WebSphere Application Server. 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 J2EE 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 name space 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 1023. For more information on configured name bindings, refer to "Configured name bindings" on page 1025.

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 name space.

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 *name space*. 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 name space.

The name space 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 name space 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 name space. For more information, see Naming roles.

Typically, objects bound to the name space 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 name space. An application can bind objects to transient or persistent partitions, depending on requirements.

In J2EE environments, some JNDI operations are performed with `java:` URL names. Names bound under these names are bound to a completely different name space which is local to the calling process. However, some lookups on the `java:` name space may trigger indirect lookups to the name server.

Name space logical view

The name space 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 name space.

The various server roots and persistent partitions of the name space are interconnected by a system name space. You can use the system name space structure to traverse to any context in a the cell's name space. A logical view of the name space is shown in the following diagram.

Logical View of a Cell's Name Space
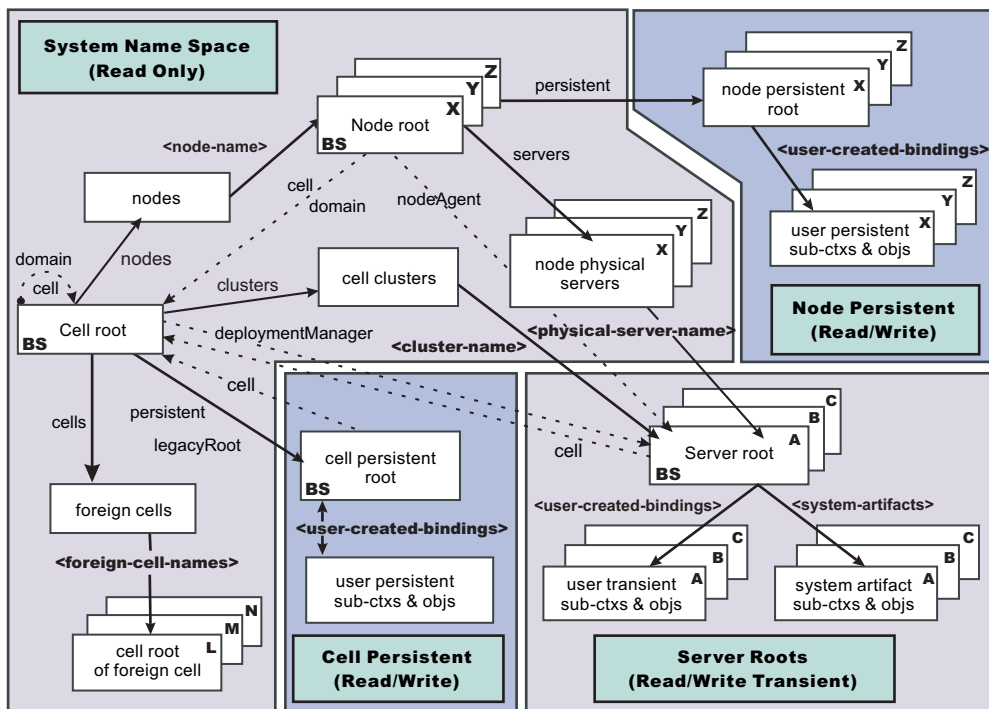
The bindings in the preceding diagram appear with solid arrows, labeled in bold, and dashed arrows, labeled in gray. Solid arrows represent *primary bindings*. A primary binding is formed when the associated subcontext is created. Dashed arrows show *linked bindings*. A linked binding is formed when 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 name space 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 name space. A name server constructs this view at startup by reading configuration information. Each name server has its own local in-memory copy of the name space 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.

Name space partitions

There are four major partitions in a cell name space:
- System name space partition
- Server roots partition
- Cell persistent partition
- Node persistent partition

System name space partition

> The system name space contains a structure of contexts based on the cell topology. The system structure supports traversal to all parts of a cell name space 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 name space 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 name space. System artifacts, such as 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 name space. Use the initial context to perform naming operations, such as looking up and binding objects in the name space.

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 name space 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 name space 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 name space lookups with a non-WebSphere Application Server 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 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 name space cannot necessarily be resolved from another context in the name space. 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 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 J2EE 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 name space 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 J2EE 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 name space partition in a cell's name space reflects the cell's topology. This structure can be navigated to reach any object bound into the cell's name space. 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 J2EE 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 single server or to a server cluster, 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 J2EE 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.

JNDI support in WebSphere Application Server

IBM WebSphere Application Server 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 WebSphere Application Server name server through the 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`

WebSphere Application Server's JNDI implementation is based on version 1.2 of the JNDI interface, and was tested with Version 1.2.1 of 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 name space. 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 name space through the configuration. Name servers add these configured bindings to the name space 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 name space. 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 name space 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 name space. 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

Name space federation

Federating name spaces involves binding contexts from one name space into another name space.

For example, assume that a name space, Name Space 1, contains a context under the name `a/b`. Also assume that a second name space, Name Space 2, contains a context under the name `x/y`. (See the following illustration.) If context `x/y` in Name Space 2 is bound into context `a/b` in Name Space 1 under the name `f2`, the two name spaces are federated. Binding `f2` is a federated binding because the context associated with that binding comes from another name space. From Name Space 1, a lookup of the name `a/b/f2` returns the context bound under the name `x/y` in Name Space 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 Name Space 1 with the lookup name `a/b/f2/ejb1`. Notice that the name crosses name spaces. This fact is transparent to the naming client.

Federated Name Spaces
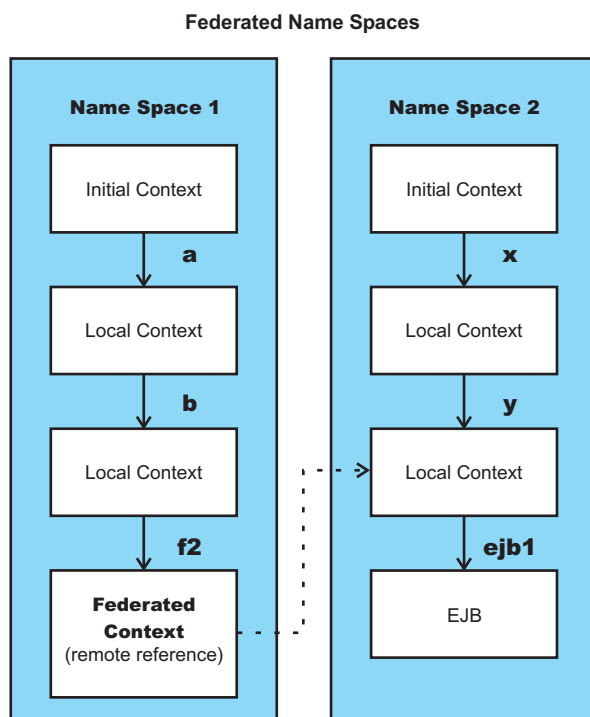
In a WebSphere Application Server name space, you can create federated bindings with the following restrictions:

- Federation is limited to CosNaming name servers. A WebSphere Application Server 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 name space, 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-WebSphere Application Server 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-WebSphere Application Server name spaces. This restriction affects the performance of lookup operations only.
  - If security is enabled, WebSphere Application Server does not support federated bindings to non-WebSphereApplication Server name spaces.

- Do not federate two WebSphere Application Server stand-alone server name spaces. Incorrect behavior might result. If you want to federate WebSphere Application Server name spaces, use servers running under the Network Deployment package of WebSphere Application Server.

- When federating the name spaces 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 29. 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 30. 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 31. 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 32. CosNamingDelete role packages and interface methods

Package	Interface methods
javax.naming	Context.destroySubcontext
org.omg.CosNaming	NamingContext.destroy

Important: 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 name space binding configured with a name in the name space 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 Version 6 is the same as that provided for Version 5, thus information on the Version 5 naming and directories applies to Version 6.

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.

Refer to "Web resources for learning" on page 15 for links to information applicable to WebSphere Application Server generally, such as lists of IBM technical papers, Redbooks and samples.

Programming instructions and examples
- Naming in WebSphere Application Server V5: Impact on Migration and Interoperability
- WebSphere Application Server V6 System Management & Configuration Handbook
- *IBM WebSphere Developer Technical Journal:* Co-hosting multiple versions of J2EE applications

Programming specifications
- Java Naming and Directory Interface^{TM} 1.2.1 Specification
- Object Management Group (OMG) Interoperable Naming specifications
  - Naming Service Specification
  - Common Object Request Broker: Architecture and Specification
  - Interoperable Naming Service revised chapters, which presents a consolidated view of all of the elements that comprise interoperable naming

Developing applications that use JNDI

References to EJB homes and other artifacts such as data sources are bound to the WebSphere Application Server name space. These objects can be obtained through the JNDI interface. 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 WebSphere Application Server name space.

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
- Looking up a JavaMail session 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 WebSphere Application Server 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);
```

2. **Optional:** Specify the name syntax

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 the command line by entering the actual string value. For example:

```
java -Dcom.ibm.websphere.naming.name.syntax=ins
```

- 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.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.

- Within a Java program by using the PROPS.NAME_SYNTAX* Java constants, defined in the `com.ibm.websphere.naming.PROPS` file. 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: 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 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 WebSphere Application Server provides a CosNaming JNDI plug-in implementation for JNDI clients to perform naming operations on WebSphere Application Server name spaces. The WebSphere Application Server 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 WebSphere Application Server initial context factory, `com.ibm.websphere.naming.WsnInitialContextFactory`, is typically used by WebSphere Application Server applications to perform JNDI operations. The WebSphere Application Server runtime environment is set up to use this WebSphere Application Server 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 WebSphere Application Server 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 WebSphere Application Server 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 of WebSphere Application Server 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 WebSphere Application Server 5 server as the bootstrap server.
  - Specify `corbaname:rir:/NameService#domain/legacyRoot` if the ORB is initialized to use a WebSphere Application Server 4.0.x server as the bootstrap server.
  - Specify `corbaloc:rir:/NameService` if the ORB is initialized to use a server other than a WebSphere Application Server 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, 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 name space 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 WebSphere Application Server's initial context factory:
* A CORBA object URL (new for J2EE 1.3)
* 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" on page 1037
* "Using a CORBA object URL from a non-WebSphere Application Server JNDI implementation" on page 1037
* "Using an IIOP URL" on page 1038

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 name space 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 name space root property: You can also select the initial root context by passing a name space 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	Name Space Root Property Value
Server Root	bootstrapserverroot
Cell Persistent Root	cellpersistentroot
Cell Root	cellroot
Node Root	bootstrapnoderoot

The initial context factory ignores the name space root property if the provider URL contains an object key other than `NameService`.

The following example shows use of the name space 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 with JNDI

Most applications that use 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 1040
- "JNDI lookup with a corbaname URL" on page 1041

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 name space. 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:` name space 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 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:` name space 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 name space 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. A topology based name depends on whether the object resides in a single server or a server cluster. Examples of each form of qualified name follow.

- **Topology-based qualified names**

  Topology-based qualified names traverse through the system name space 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 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 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 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/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 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 name space and cannot be located with a qualifiied name, a corbaname can be a convenient way to look up the object. 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
    ...
}
```

Example: Looking up a JavaMail session with JNDI
A program can conduct a JNDI lookup of a JavaMail resource. Deployment descriptors of the application determine the lookup name that you specify.

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");
```

JNDI interoperability considerations
You must take extra steps to enable your programs to interoperate with non-WebSphere Application Server JNDI clients and to bind resources from MQSeries to a name space.

EJB clients running in an environment other than WebSphere Application Server accessing EJB applications running on WebSphere Application Server V5 or V6 servers

When an EJB application running in WebSphere Application Server V5 or V6 is accessed by a non-WebSphere Application Server EJB client, the JNDI initial context factory is presumed to be a non-WebSphere Application Server 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 V5, the MQSeries jmsadmin tool could be used to bind resources to the name space. When used with a WebSphere Application Server V5 or V6 name space, the resource is bound within a transient partition in the name space 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 WebSphere Application Server administrative console, under **Resources** in the console navigation tree.

JNDI caching
To increase the performance of JNDI operations, the WebSphere Application Server 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 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 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 name space of a different name server. By default, objects from each bootstrap address are cached separately, since they each involve independent name spaces 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.

Example: Controlling JNDI cache behavior from a program
You can specify JNDI cache properties in a program to control the behavior of a JNDI cache.

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");
// ...
```

JNDI name syntax

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.

INS name syntax

INS syntax is designed for JNDI clients that need to interoperate with CORBA applications.

The INS 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 to CORBA name mapping considerations

WebSphere Application Server name servers are an implementation of the CORBA CosNaming interface. WebSphere Application Server provides a 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.

Example: Setting the syntax used to parse name strings

JNDI clients that must interoperate with CORBA applications might need to use INS name syntax to represent names in string format.

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");
...
```

Developing applications that use CosNaming (CORBA Naming interface)

CORBA clients can perform naming operations on WebSphere Application Server name servers through the CosNaming interface.

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.

Usage scenario

```
...
import java.util.Properties;
import org.omg.CORBA.ORB;
...
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.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:

Usage scenario

```
...
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.

Usage scenario

```
...
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 1022.

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.

Usage scenario

```
...
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 name space 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 name space 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 name space 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.

Usage scenario

```
// 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 name space 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);
```

Object Request Broker

Managing Object Request Brokers

Use this task to manage Object Request Brokers (ORB). An ORB manages the interaction between clients and servers using the Internet InterORB Protocol (IIOP).

Default property values are set when the product starts and the Java Object Request Broker (ORB) service 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. It might be necessary to modify some ORB settings under certain conditions.

Every request or response exchange consists of a client-side ORB and a server-side ORB. It is important to set the ORB properties for both sides as necessary.

After an ORB instance has been established in a process, changes to ORB properties do not affect the behavior of the running ORB instance. The process must be stopped and restarted for the modified properties to take effect.

A list of possible tasks for managing ORB follows. These steps can be performed in any order.

1. Adjust timeout settings to improve handling of network failures. See "Object Request Broker service settings" on page 1052 for more information. Before making these adjustments, read Object Request Broker tuning guidelines.
2. Adjust thread-pool settings used by the ORB for handling Internet InterORB Protocol (IIOP) connections. See Thread pool settings for more information.
3. If problems with the ORB arise, see "Object request broker troubleshooting tips" on page 1070.

   For help in troubleshooting, see "Object Request Broker communications trace" on page 1063.

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.

Logical pool distribution

The Logical pool distribution (LPD) thread pool mechanism implements a strategy for improving the performance of requests that have shorter run times. Do not configure LPD unless you have already configured it in a previous release of WebSphere Application Server. LPD is a deprecated function and will be removed in a future version of the product.

The need for LPD is indicated by a mixture of Enterprise JavaBeans (EJB) requests where the run times vary across the request types, and the ORB thread pool must be constrained for performance reasons. In this case, longer run time requests might tend to prolong the response times for shorter requests by denying them adequate access to threads in the thread pool. LPD provides a mechanism that allows shorter requests greater access to the threads.

LPD divides the Object Request Broker (ORB) thread pool into logical pools, as configured by the administrator using ORB custom properties starting that start with the following:

```
com.ibm.websphere.threadpool.strategy.*
```

The size of each pool is a percentage of the maximum number of ORB threads. The sum of the logical pool percentages must equal 100.

When LPD is active, incoming ORB requests are vectored, or pointed, to a pool based on historical run time history for the request type. The request type is determined by the method, which is qualified internally as unique across components. The LPD mechanism adjusts pool targets at runtime to optimize the distribution of requests across logical pools.

The LPD mechanism can be tuned after it is enabled. Response time, throughput measurements, and statistics produced by the LPD mechanism drive the tuning process.

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 > 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

Request retries count:

Specifies the number of times that the ORB attempts to send a request if a server fails. Retrying sometimes enables recovery from transient network failures. This field is ignored on the z/OS platform.

If you use command-line scripting, the full name of this system property is com.ibm.CORBA.requestRetriesCount.

Data type	int
Default	1
Range	1 to 10

Request retries delay:

Specifies the number of milliseconds between request retries. This field is ignored on the z/OS platform.

If you use command-line scripting, the full name of this system property is com.ibm.CORBA.requestRetriesDelay.

Data type	int
Units	Milliseconds
Default	0
Range	0 to 60,000

Connection cache maximum:

Specifies the maximum number of entries that can occupy the ORB connection cache before the ORB starts to remove inactive connections from the cache. This field is ignored on the z/OS platform.

It is possible that the number of active connections in the cache will temporarily exceed this threshold value. If necessary, the ORB will continue to add connections as long as resources are available.

For use in command-line scripting, the full name of this system property is com.ibm.CORBA.MaxOpenConnections.

Data type	Integer
Units	Connections
Default	240
Range	10 - largest integer recognized by Java

Connection cache minimum:

Specifies the minimum number of entries in the ORB connection cache. This field is ignored on the z/OS platform.

The ORB will not remove inactive connections when the number of entries is below this value.

For use in command-line scripting, the full name of this system property is com.ibm.CORBA.MinOpenConnections.

Data type	Integer
Units	Connections
Default	100

Range	Any integer that is at least 5 less than the value specified for the Connection cache maximum property.

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)

Locate request timeout:

Specifies the number of seconds to wait before timing out on a LocateRequest message. This field is ignored on the z/OS platform.

If you use command-line scripting, the full name of this system property is com.ibm.CORBA.LocateRequestTimeout.

Data type	int
Units	Seconds
Default	180
Range	0 to 300

Force tunneling:

Controls how the client ORB attempts to use HTTP tunneling. This field is ignored on the z/OS platform.

If you use command-line scripting, the full name of this system property is com.ibm.CORBA.ForceTunnel.

Data type	String
Default	NEVER
Range	Valid values are ALWAYS, NEVER, or WHENREQUIRED.

Considering the following information when choosing the valid value:

ALWAYS
Use HTTP tunneling immediately, without trying TCP connections first.

NEVER
Disable HTTP tunneling. If a TCP connection fails, a CORBA system exception (COMM_FAILURE) occurs.

WHENREQUIRED
Use HTTP tunneling if TCP connections fail.

Tunnel agent URL:

Specifies the Web address of the servlet to use in support of HTTP tunneling. This field is ignored on the z/OS platform.

This Web address must be a proper format:

http://w3.mycorp.com:81/servlet/com.ibm.CORBA.services.IIOPTunnelServlet

For applets: http://applethost:port/servlet/com.ibm.CORBA.services.IIOPTunnelServlet.

This field is required if HTTP tunneling is set. If you use command-line scripting, the full name of this system property is com.ibm.CORBA.TunnelAgentURL.

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 WebSphere Application Server instance, 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.

Important: 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 WebSphere Application Server 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 > 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 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 ORB custom properties to configure an ORB, you must understand that there are two types of default values for some of these properties: JDK default values and 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 WebSphere Application Server sets for a property in one of the following files:
* The orb.properties file when an application server is installed.
* The server.xml 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 is 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, see the JDK Diagnostic Guide for the version of the JDK that you are using. These guides are available at http://www-128.ibm.com/developerworks/java/jdk/diagnosis/.

The WebSphere Application Server orb.properties file, that is located in the WebSphere Application Server *was_home*/java/jre/lib directory, contains WebSphere Application server ORB custom properties that are initially set to WebSphere Application Server default values during the WebSphere Application Server installation process.

You can specify new values for the ORB custom properties in the administrative console. Any value 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 WebSphere Application Server 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, in the administrative console, click **Servers >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 WebSphere Application Server installation. A list of additional properties associated with the Java ORB service follows. Unless otherwise indicated, the default values 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. This setting is deprecated and is scheduled for removal in a future release.

For a command-line or programmatic alternative, see "Client-side programming tips for the Java Object Request Broker service" on page 1066.

com.ibm.CORBA.BootstrapPort:

Specifies the port to which the ORB connects for bootstrapping, the port of the machine on which the initial server contact for this client listens. This setting is deprecated and is scheduled for removal in a future release.

For a command-line or programmatic alternative, see "Client-side programming tips for the Java Object Request Broker service" on page 1066.

Default	2809

com.ibm.CORBA.ConnectTimeout:

The com.ibm.CORBA.ConnectTimeout property specifies the maximum time in seconds that the client ORB waits before timing out when attempting to establish an IIOP connection with a remote server ORB. Generally, client applications use this property. The property is not used by the application server by default. 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. This 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 (seconds)
Default	0 (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 very small, setting this property to false might improve performance because this setting change can reduce the GIOP traffic by as much as half. If most of your initial method invocations are large, you should set this property to true so that the small locate request message is sent instead of the large message. The large message is then sent to the proper target after the desired object is found.

WebSphere Application Server default	true
JDK default	false

com.ibm.CORBA.FragmentSize:

Specifies the size of General Inter-ORB Protocol (GIOP) fragments used by the ORB. If the total size of a request exceeds the set value, the ORB breaks up and sends multiple fragments until the entire request is sent. Set this property on the client side with a -D system property if you use a stand-alone Java application.

Adjust the `com.ibm.CORBA.FragmentSize` property if the amount of data that is sent over Internet Inter-ORB Protocol (IIOP) in most General Inter-ORB Protocol (GIOP) requests exceeds one kilobyte or if thread dumps show that most client-side threads seem to be waiting while sending or receiving data. Adjust this property so that most messages have few or no fragments.

If you want to instruct the ORB not to break up any of the requests or replies it sends, set this property to `0` (zero). However, setting the value to zero does not prevent the ORB from receiving GIOP fragments in requests or replies sent by another existing ORB.

Units	Bytes.
Default	1024
Range	From 64 to the largest value of a Java integer type that is divisible by 8

com.ibm.CORBA.ListenerPort:

Specifies the port on which this server listens for incoming requests. The setting of this property is valid for client-side ORBs only.

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 WebSphere Application Server sets this property to the host name or IP address specified for the BOOTSTRAP_ADDRESS end point in serverindex.xml. 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.numJNIReaders:

Specifies the number of JNI reader threads to be allocated in the ORB's JNI reader thread pool. Each thread can handle up to 1024 connections.

Attention: Before setting this property, make sure that a JSSE provider has been selected as the provider for the SSL repertoire associated with the port on which the ORB service listens for incoming requests. IBMJSSE2 is the default provider setting for SSL repertoires and does not provide the file descriptor that JNIReader Threads require.

Valid Range	1-2147483647
Default	4

com.ibm.CORBA.ORBPluginClass.com.ibm.ws.orbimpl.transport.JNIReaderPoolImpl:

Specifies that JNI reader threads will be used. The property name specifies the class name of the ORB component that manages the pool of JNI reader threads and interacts with the native OS library used to process multiple connections simultaneously.

Attention:

- Make sure the library is in the WebSphere Application Server bin directory.

  For an Intel platform, the library is Selector.dll and for a UNIX platform, it is libSelector.a or libSelector.so.

  For the UNIX platform, if the prefix ″lib″ is missing, the file should be renamed.

- When specifying this property using the administrative console, enter com.ibm.CORBA.ORBPluginClass.com.ibm.ws.orbimpl.transport.JNIReaderPoolImpl for the property name and an empty string (″″) for the value.

  When specifying this property on the java command, do not include a value:

  ```
  -Dcom.ibm.CORBA.ORBPluginClass.com.ibm.ws.orbimpl.transport.JNIReaderPoolImpl
  ```

Valid Range	not applicable
Default	none

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 WebSphere Application Server RAS processing.

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 remain unhandled by the WebSphere Application Server ORB before the Application Server 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, usually either the port on the target machine isn't 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 dfault	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 V5.1.1.

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 removing 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.

See the *Migrating, coexisting, and interoperating* PDF for instructions on how to perform an incremental cell upgrade.

com.ibm.websphere.threadpool.strategy.implementation:

Specifies the logical pool distribution (LPD) thread pool strategy the next time you start the application server, and is enabled if set to com.ibm.ws.threadpool.strategy.LogicalPoolDistribution.

Attention: This is a deprecated function and will be removed in a future version of the product. Do not configure logical pool distribution unless you have already configured it for a previous release of WebSphere Application Server.

Some requests have shorter start times than others. LPD is a mechanism for providing these shorter requests greater access to start threads. For more information, see the information center.

com.ibm.websphere.threadpool.strategy.LogicalPoolDistribution.calcinterval:

Specifies how often the logical pool distribution (LPD) mechanism readjusts the pool start target times. This property cannot be turned off after this support is installed.

Attention: Do not configure logical pool distribution unless you have already configured it with a previous release of WebSphere Application Server. This function is deprecated and will be removed in a future version of the product.

LPD must be enabled (see com.ibm.websphere.threadpool.strategy.implementation).

Data type	Integer
Units	Milliseconds
Default	30
Range	20,000 milliseconds minimum

com.ibm.websphere.threadpool.strategy.LogicalPoolDistribution.lruinterval:

Specifies how long the logical pool distribution internal data is kept for inactive requests. The mechanism tracks several statistics for each request type that is received. Consider removing requests that have been inactive for awhile.

Attention: Do not configure logical pool distribution unless you have already configured it with a previous release of WebSphere Application Server. This function is deprecated and will be removed in a future version of the product.

LPD must be enabled (see com.ibm.websphere.threadpool.strategy.implementation).

Data type	Integer
Units	Milliseconds
Default	300,000 (5 minutes)
Range	60,000 (1 minute) minimum

com.ibm.websphere.threadpool.strategy.LogicalPoolDistribution.outqueues:

Specifies how many pools are created and how many threads are allocated to each pool in the logical pool distribution mechanism.

Attention: Do not configure logical pool distribution unless you have already configured it with a previous release of WebSphere Application Server. This function is deprecated and will be removed in a future version of the product.

The ORB parameter for max threads controls the total number of threads. The outqueues parameter is specified as a comma separated list of percentages that add up to 100. For example, the list 25,25,25,25 sets up 4 pools, each allocated 25% of the available ORB thread pool. The pools are indexed left to right from 0 to n-1. Each outqueue is dynamically assigned a target start time by the calculation mechanism. Target start times are assigned to outqueues in increasing order so pool 0 gets the requests with the least start time and pool n-1 gets requests with the highest start times.

LPD must be enabled (see com.ibm.websphere.threadpool.strategy.implementation).

Data type	Integers in comma separated list
Default	25,25,25,25
Range	Percentages in list must total 100 percent

com.ibm.websphere.threadpool.strategy.LogicalPoolDistribution.statsinterval:

Specifies that statistics are dumped to stdout after this interval expires, but only if requests are processed. This process keeps the mechanism from filling the log files with redundant information. These statistics are beneficial for tuning the logical pool distribution mechanism.

Attention: Do not configure logical pool distribution unless you have already configured it with a previous release of WebSphere Application Server. This function is deprecated and will be removed in a future version of the product.

LPD must be enabled (see com.ibm.websphere.threadpool.strategy.implementation).

Data type	Integer
Units	Milliseconds
Default	0 (off)
Range	30,000 (30 seconds) minimum

com.ibm.websphere.threadpool.strategy.LogicalPoolDistribution.workqueue:

Specifies the size of a new queue where incoming requests wait for dispatch. Pertains to the logical pool distribution mechanism.

Attention: Do not configure logical pool distribution unless you have already configured it with a previous release of WebSphere Application Server. This function is deprecated and will be removed in a future version of the product.

LPD must be enabled (see com.ibm.websphere.threadpool.strategy.implementation).

Data type	Integer
Default	96
Range	10 minimum

com.ibm.ws.orb.services.redirector.MaxOpenSocketsPerEndpoint:

Specifies the maximum number of connections that the IIOP Tunnel Servlet should maintain in its connection cache for each target host or port. If the number of concurrent client requests to a single host or 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 improperly 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 should 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 would normally timeout 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 WebSphere Application Server ORB binds to all network interfaces in the system. If true is specified, the ORB binds to all network interfaces that are available to it. If false is specified, the ORB only binds to the network interface 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 WebSphere Application Server 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.

Object Request Broker communications trace

The Object Request Broker (ORB) communications trace, typically referred to as *CommTrace*, contains the sequence of General InterORB Protocol (GIOP) messages sent and received by the ORB when the application is running.

It might be necessary to understand the low-level sequence of client-to-server or server-to-server interactions during problem determination. This topic uses trace entries from log examples to explain the contents of the log and help you understand the interaction sequence. It focuses only in the GIOP messages and does not discuss in detail additional trace information that displays when intervening with the GIOP-message boundaries.

Location

When ORB tracing is enabled, this information is placed in the *app_server_root*/profiles/*profile_name*/logs/*server_name*/trace.log directory.

About the ORB trace file

The following are properties of the file that is created when ORB tracing is enabled.
* Read-only
* Updated by the administrative function
* Use this file to localize and resolve ORB-related problems.

How to interpret the output

The following sections refer to sample log output found later in this topic.

Identifying information

The start of a GIOP message is identified by a line that contains either OUT GOING: or IN COMING: depending on whether the message is sent or received by the process.

Following the identifying line entry is a series of items, formatted for convenience, with information extracted from the raw message that identifies the endpoints in this particular message interaction. See lines 3-13 in both examples. The formatted items include the following:
- GIOP message type (line 3)
- Date and time that the message was recorded (line 4)
- Information that is useful to identify the thread that is running when the message records, and other thread-specific information (line 5)
- Local and remote TCP/IP ports used for the interaction (lines 6 through 9)
- GIOP version, byte order, an indication of whether the message is a fragment, and message size (lines 10 through 13)

Request ID, response expected and reply status

Following the introductory message information, the request ID is an integer generated by the ORB. It is used to identify and associate each request with its corresponding reply. This association is necessary because the ORB can receive requests from multiple clients and must be able to associate each reply with the corresponding originating request.
- Lines 15-17 in the request example show the request ID, followed by an indication to the receiving endpoint that a response is expected (CORBA supports sending one-way requests for which a response is not expected.)
- Line 15 in the *Sample Log Entry - GIOP Reply* shows the request ID; line 33 shows the reply status received after completing the previously sent request.

Object Key

Lines 18-20 in the request example show the object key, the internal representation used by the ORB to identify and locate the target object intended to receive the request message. Object keys are not standardized.

Operation

Line 21 in the request example shows the name of the operation that the target object starts in the receiving endpoint. In this example, the specific operation requested is named _get_value.

Service context information

The service contexts in the message are also formatted for convenience. Each GIOP message might contain a sequence of service contexts sent and received by each endpoint. Service contexts, identified uniquely with an ID, contain data used in the specific interaction, such as security, character code set conversion, and ORB version information. The content of some of the service contexts is standardized and specified by OMG, while other service contexts are proprietary and specified by each vendor. IBM-specific service contexts are identified with IDs that begin with 0x4942.

Lines 22-41 in the request example illustrate typical service context entries. Three service contexts are in the request message, as shown in line 22. The ID, length of data, and raw data for each service context is printed next. Lines 23-25 show an IBM-proprietary context, as indicated by the 0x49424D12 ID. Lines 26-41 show two standard service contexts, identified by 0x6 ID (line 26) and the 0x1 ID (line 39).

Lines 16-32 in the *Sample Log Entry - GIOP Reply* illustrate two service contexts, one IBM-proprietary (line 17) and one standardized (line 20).

For the definition of the standardized service contexts, see the CORBA specification. Service context 0x1 (CORBA::IOP::CodeSets) is used to publish the character code sets supported by the ORB in order to negotiate and determine the code set used to transmit character data. Service context 0x6 (CORBA::IOP::SendingContextRunTime) is used by Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) to provide the receiving endpoint with the IOR for the

SendingContextRuntime object. IBM service context 0x49424D12 is used to publish ORB PartnerVersion information to support release-to-release interoperability between sending and receiving ORBs.

Data offset

Line 42 in the request example shows the offset, relative to the beginning of the GIOP message, where the remainder body of the request or reply message is located. This portion of the message is specific to each operation and varies from operation to operation. Therefore, it is not formatted, as the specific contents are not known by the ORB. The offset is printed as an aid to quickly locating the operation-specific data in the raw GIOP message dump, which follows the data offset.

Raw GIOP message dump

Starting at line 45 in the request example and line 36 in the *Sample Log Entry - GIOP Reply*, a raw dump of the entire GIOP message is printed in hexadecimal format. Request messages contain the parameters required by the given operation and reply messages contain the return values and content of output parameters as required by the given operation. For brevity, not all of the raw data is in the figures.

Sample Log Entry - GIOP Request

```
1.  OUT GOING:

3.  Request Message
4.  Date:           April 17, 2002 10:00:43 PM CDT
5.  Thread Info:    P=842115:O=1:CT
6.  Local Port:     1243 (0x4DB)
7.  Local IP:       jdoe.austin.ibm.com/192.168.1.101
8.  Remote Port:    1242 (0x4DA)
9.  Remote IP:      jdoe.austin.ibm.com/192.168.1.101
10.  GIOP Version:  1.2
11. Byte order:     big endian
12. Fragment to follow: No
13. Message size:   268 (0x10C)
--
15. Request ID:        5
16. Response Flag:     WITH_TARGET
17. Target Address:     0
18. Object Key:        length = 24 (0x18)
                       4B4D4249 00000010 BA4D6D34 000E0008
                       00000000 00000000
21. Operation:        _get_value
22. Service Context:  length = 3 (0x3)
23. Context ID: 1229081874 (0x49424D12)
24. Context data: length = 8 (0x8)
                       00000000 13100003
26. Context ID:  6 (0x6)
27. Context data: length = 164 (0xA4)
                       00000000 00000028 49444C3A 6F6D672E
                       6F72672F 53656E64 696E6743 6F6E7465
                       78742F43 6F646542 6173653A 312E3000
                       00000001 00000000 00000068 00010200
                       0000000E 3139322E 3136382E 312E3130
                       310004DC 00000018 4B4D4249 00000010
                       BA4D6D69 000E0008 00000000 00000000
                       00000002 00000001 00000018 00000000
                       00010001 00000001 00010020 00010100
                       00000000 49424D0A 00000008 00000000
                       13100003
39. Context ID:  1 (0x1)
40. Context data: length = 12 (0xC)
                       00000000 00010001 00010100
42. Data Offset:      118
```

```
45. 0000: 47494F50 01020000 0000010C 00000005   GIOP............
46. 0010: 03000000 00000000 00000018 4B4D4249   ............KMBI
47. 0020: [remainder of message body deleted for brevity]
```

Sample Log Entry - GIOP Reply

```
1.  IN COMING:

3.  Reply Message
4.  Date:          April 17, 2002 10:00:47 PM CDT
5.  Thread Info:   RT=0:P=842115:O=1:com.ibm.rmi.transport.TCPTransportConnection
5a (line 5 broken for publication).   remoteHost=192.168.1.101 remotePort=1242 localPort=1243
6.  Local Port:    1243 (0x4DB)
7.  Local IP:      jdoe.austin.ibm.com/192.168.1.101
8.  Remote Port:   1242 (0x4DA)
9.  Remote IP:     jdoe.austin.ibm.com/192.168.1.101
10. GIOP Version:  1.2
11. Byte order:    big endian
12. Fragment to follow: No
13. Message size:  208 (0xD0)
--
15. Request ID:        5
16. Service Context:   length = 2 (0x2)
17. Context ID:  1229081874 (0x49424D12)
18. Context data:  length = 8 (0x8)
                    00000000 13100003
20. Context ID:  6 (0x6)
21. Context data:  length = 164 (0xA4)
                    00000000 00000028 49444C3A 6F6D672E
                    6F72672F 53656E64 696E6743 6F6E7465
                    78742F43 6F646542 6173653A 312E3000
                    00000001 00000000 00000068 00010200
                    0000000E 3139322E 3136382E 312E3130
                    310004DA 00000018 4B4D4249 00000010
                    BA4D6D34 000E0008 00000001 00000000
                    00000002 00000001 00000018 00000000
                    00010001 00000001 00010020 00010100
                    00000000 49424D0A 00000008 00000000
                    13100003
33. Reply Status:      NO_EXCEPTION


36. 0000: 47494F50 01020001 000000D0 00000005   GIOP............
37. 0010: 00000000 00000002 49424D12 00000008   ........IBM.....
38. 0020: [remainder of message body deleted for brevity]
```

Client-side programming tips for the Java Object Request Broker service

This topic includes programming tips for applications that communicate with the client-side Object Request Broker (ORB) that is part of the Java ORB service.

Resolution of initial references to services

Client applications can use the ORBInitRef and ORBDefaultInitRef properties to configure the network location that the Java 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 2 Platform, Enterprise Edition
(J2EE) 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 J2EE 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 J2EE applications (including enterprise beans, J2EE 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 J2EE components that are
running in the same Java virtual machine process.

CORBA approach: Because thin-client applications do not run in a J2EE 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 J2EE application components

For performance reasons, it often makes sense to share a single ORB instance among components in a
J2EE application. As required by the J2EE 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 J2EE 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.

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	

Java name	OSF registry ID	Comments
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	
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.

Object request broker troubleshooting tips
Use these tips to diagnose problems related to the WebSphere Application Server Object Request Broker (ORB).
- "Enabling tracing for the Object Request Broker component"
- "Log files and messages associated with Object Request Broker" on page 1071
- "Adjusting object request broker timeout values" on page 1072
- "Java packages containing the Object Request Broker service" on page 1072
- "Tools used with Object Request Broker" on page 1073
- "Object Request Broker properties" on page 1073
- "CORBA minor codes" on page 1073

Enabling tracing for the Object Request Broker component

The object request broker (ORB) service is one of the WebSphere Application Server run time services. Tracing messages 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 WebSphere Application Server installation, and for each application client.

This tracing is referred to by WebSphere Application Server support as a *comm trace*, and is different from the general purpose trace facility. The trace facility, which shows the detailed run-time behavior of product components, may be used alongside comm trace for other product components, or for the ORB component. The trace string associated with the ORB service is `ORBRas=all=enabled`.

You can enable and disable comm tracing using the administrative console or by manually editing the server.xml file for the server be trace. You must stop and restart the server for the configuration change to take effect.

For example, using the administrative console:
- Click **Servers > Application servers >** *server_name* **> Container services > ORB service**, and select the ORB tracing. Click **OK**, and then click **Save** to save your settings. Restart the server for the new settings to take effect. Or,
- Locate the server.xml file for the selected server, for example: *profile_root*/config/cells/*node_name*/nodes/*node_name*/servers/*server_name*/server.xml.
- Locate the services entry for the ORB service, `xmi:type=orb:ObjectRequestBroker`, and set `commTraceEnabled=true`.

ORB tracing for client applications requires that both the ORBRas component trace and the ORB comm trace are enabled. If only the ORB comm trace is enabled, no trace output is generated for client-side ORB traces. You can use one of the following options to enable both traces in the command line used to launch the client application:
- If you are using the WebSphere Application Server launcher, launchClient, use the **-CCD** option.
- If you are using the **java** command specify these parameters:

```
-trace=ORBRas=all=enabled
-tracefile=filename
-Dcom.ibm.CORBA.Debug=true
-Dcom.ibm.CORBA.CommTrace=true
```

ORB tracing output for thin clients can be directed by setting the `com.ibm.CORBA.Debug.Output =` `debugOutputFilename` parameter in the command line.

Important: When using launchClient on operating systems like AIX or Linux, because ORB tracing is integrated with the WebSphere Application Server general component trace, both the component and comm ORB traces are written to the same file as the rest of the WebSphere Application Server traces. The name of this file is specified on the `-CCtracefile=`*filename* parameter.

When using launchClient on a Windows operating systems, you get a separate ORB trace file, called orbtrc.*timestamp*.txt. This file is located in the current directory.

Log files and messages associated with Object Request Broker

Messages and trace information for the ORB are saved in the following logs:
- The *profile_root*/logs/*server_name*/trace.log file for output from communications tracing and tracing the behavior of the ORBRas component
- The JVM logs for each application server, for WebSphere Application Server error and warning messages

The following message in the SystemOut.log file indicates the successful start of the application server and its ORB service:

WSVR0001I: Server server1 open for e-business

When communications tracing is enabled, a message similar to the following example in the *profile_root*/logs/*server_name*/trace.log file, indicates that the ORB service has started successfully. The message also shows the start of a listener thread, which is waiting for requests on the specified local port.

com.ibm.ws.orbimpl.transport.WSTransport startListening(ServerConnectionData connectionData) P=693799:O=0:CT a new ListenerThread has been started for ServerSocket[addr=0.0.0.0/ 0.0.0.0,port=0,localport=1360]

When the `com.ibm.ejs.oa.*=all=enabled` parameter is specified, tracing of the Object Adapter is enabled. The following message in the trace.log indicates that the ORB service started successfully:

EJSORBImpl < initializeORB

The ORB service is one of the first services started during the WebSphere Application Server initialization process. If it is not properly configured, other components such as naming, security, and node agent, are not likely to start successfully. This is obvious in the JVM logs or trace.log of the affected application server.

Adjusting object request broker timeout values

If Web clients that access Java applications running in the product environment are consistently experiencing problems with their requests, and the problem cannot be traced to other sources and addressed through other solutions, consider setting an ORB timeout value and adjusting it for your environment. A list of timeout scenarios follows:
*   Web browsers vary in their language for indicating that they have timed out. Usually, the problem is announced as a connection failure or a no-path-to-server message.
*   Set an ORB timeout value to less than the time after which a Web client eventually times out. Because it can be difficult to tell how long Web clients wait before timing out, setting an ORB timeout value requires experimentation. The ideal testing environment features some simulated network failures for testing the proposed setting value.
*   Empirical results from limited testing indicate that 30 seconds is a reasonable starting value. Ensure that this setting is not too low. To fine tune the setting, find a value that is not too low. Gradually decrease the setting until reaching the threshold at which the value becomes too low. Set the value a little higher than the threshold.
*   When an ORB timeout value is set too low, the symptom is numerous CORBA NO_RESPONSE exceptions, which occur even for some valid requests. The value is likely to be too low if requests that should have been successful, for example, the server is not down, are being lost or refused.

Timeout adjustments: Do not adjust an ORB timeout value unless you have a problem. Configuring a value that is inappropriate for the environment can create a problem. An incorrect value can produce results worse than the original problem.

You can adjust timeout intervals for the product Java ORB through the following administrative settings:
*   **Request timeout**, the number of seconds to wait before timing out on most pending ORB requests if the network fails
*   **Locate request timeout**, the number of seconds to wait before timing out on a locate-request message

See "Object Request Broker service settings" on page 1052 for more information.

Java packages containing the Object Request Broker service

The ORB service resides in the following Java packages:
*   com.ibm.com.CORBA.*
*   com.ibm.rmi.*
*   com.ibm.ws.orb.*
*   com.ibm.ws.orbimpl.*
*   org.omg.CORBA.*
*   javax.rmi.CORBA.*

JAR files that contain the previously mentioned packages include:
*   *app_server_root*/java/jre/lib/ext/ibmorb.jar
*   *app_server_root*/java/jre/lib/ext/iwsorbutil.jar
*   *app_server_root*/lib/iwsorb.jar

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 Java packages:
- com.ibm.tools.rmic.*
- com.ibm.idl.*

The JAR file that contains the packages is *app_server_root*/java/lib/ibmtools.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 it is recommended that only your system administrator modify them when required.. Consult IBM Support personnel for assistance. The properties reside in the orb.properties file, located in *app_server_root*/java/jre/lib/orb.properties.

CORBA minor codes

The CORBA specification defines standard minor exception codes for use by the ORB when a system exception is thrown. In addition, the object management group (OMG) assigns each vendor a unique prefix value for use in vendor-proprietary minor exception codes. Minor code values assigned to IBM and used by the ORB in the WebSphere Application Server follow. The minor code value is in decimal and hexadecimal formats. The column labeled minor code reason gives a short description of the condition causing the exception. Currently there is no documentation for these errors beyond the minor code reason. If you require technical support from IBM, the minor code helps support engineers determine the source of the problem.

Table 33. Decimal minor exception codes 1229066320 to 1229066364

Decimal	Hexadecimal	Minor code reason
1229066320	0x49421050	HTTP_READER_FAILURE
1229066321	0x49421051	COULD_NOT_INSTANTIATE_CLIENT_SSL_SOCKET_FACTORY
1229066322	0x49421052	COULD_NOT_INSTANTIATE_SERVER_SSL_SOCKET_FACTORY
1229066323	0x49421053	CREATE_LISTENER_FAILED_1
1229066324	0x49421054	CREATE_LISTENER_FAILED_2
1229066325	0x49421055	CREATE_LISTENER_FAILED_3
1229066326	0x49421056	CREATE_LISTENER_FAILED_4
1229066327	0x49421057	CREATE_LISTENER_FAILED_5
1229066328	0x49421058	INVALID_CONNECTION_TYPE
1229066329	0x49421059	HTTPINPUTSTREAM_NO_ACTIVEINPUTSTREAM
1229066330	0x4942105a	HTTPOUTPUTSTREAM_NO_OUTPUTSTREAM
1229066331	0x4942105b	CONNECTIONINTERCEPTOR_INVALID_CLASSNAME
1229066332	0x4942105c	NO_CONNECTIONDATA_IN_CONNECTIONDATACARRIER
1229066333	0x4942105d	CLIENT_CONNECTIONDATA_IS_INVALID_TYPE
1229066334	0x4942105e	SERVER_CONNECTIONDATA_IS_INVALID_TYPE
1229066335	0x4942105f	NO_OVERLAP_OF_ENABLED_AND_DESIRED_CIPHER_SUITES
1229066352	0x49421070	CAUGHT_EXCEPTION_WHILE_CONFIGURING_SSL_CLIENT_SOCKET
1229066353	0x49421071	GETCONNECTION_KEY_RETURNED_FALSE
1229066354	0x49421072	UNABLE_TO_CREATE_SSL_SOCKET
1229066355	0x49421073	SSLSERVERSOCKET_TARGET_SUPPORTS_LESS_THAN_1
1229066356	0x49421074	SSLSERVERSOCKET_TARGET_REQUIRES_LESS_THAN_1
1229066357	0x49421075	SSLSERVERSOCKET_TARGET_LESS_THAN_TARGET_REQUIRES
1229066358	0x49421076	UNABLE_TO_CREATE_SSL_SERVER_SOCKET
1229066359	0x49421077	CAUGHT_EXCEPTION_WHILE_CONFIGURING_SSL_SERVER_SOCKET
1229066360	0x49421078	INVALID_SERVER_CONNECTION_DATA_TYPE

Table 33. Decimal minor exception codes 1229066320 to 1229066364 (continued)

1229066361	0x49421079	GETSERVERCONNECTIONDATA_RETURNED_NULL
1229066362	0x4942107a	GET_SSL_SESSION_RETURNED_NULL
1229066363	0x4942107b	GLOBAL_ORB_EXISTS
1229066364	0x4942107c	CONNECT_TIME_OUT

Table 34. Decimal minor exception codes 1229123841 to 1229124249

Decimal	Hexadecimal	Minor code reason
1229123841	0x4942f101	DSIMETHOD_NOTCALLED
1229123842	0x4942f102	BAD_INV_PARAMS
1229123843	0x4942f103	BAD_INV_RESULT
1229123844	0x4942f104	BAD_INV_CTX
1229123845	0x4942f105	ORB_NOTREADY
1229123879	0x4942f127	PI_NOT_POST_INIT
1229123880	0x4942f128	INVALID_EXTENDED_PI_CALL
1229123969	0x4942f181	BAD_OPERATION_EXTRACT_SHORT
1229123970	0x4942f182	BAD_OPERATION_EXTRACT_LONG
1229123971	0x4942f183	BAD_OPERATION_EXTRACT_USHORT
1229123972	0x4942f184	BAD_OPERATION_EXTRACT_ULONG
1229123973	0x4942f185	BAD_OPERATION_EXTRACT_FLOAT
1229123974	0x4942f186	BAD_OPERATION_EXTRACT_DOUBLE
1229123975	0x4942f187	BAD_OPERATION_EXTRACT_LONGLONG
1229123976	0x4942f188	BAD_OPERATION_EXTRACT_ULONGLONG
1229123977	0x4942f189	BAD_OPERATION_EXTRACT_BOOLEAN
1229123978	0x4942f18a	BAD_OPERATION_EXTRACT_CHAR
1229123979	0x4942f18b	BAD_OPERATION_EXTRACT_OCTET
1229123980	0x4942f18c	BAD_OPERATION_EXTRACT_WCHAR
1229123981	0x4942f18d	BAD_OPERATION_EXTRACT_STRING
1229123982	0x4942f18e	BAD_OPERATION_EXTRACT_WSTRING
1229123983	0x4942f18f	BAD_OPERATION_EXTRACT_ANY
1229123984	0x4942f190	BAD_OPERATION_INSERT_OBJECT_1
1229123985	0x4942f191	BAD_OPERATION_INSERT_OBJECT_2
1229123986	0x4942f192	BAD_OPERATION_EXTRACT_OBJECT_1
1229123987	0x4942f193	BAD_OPERATION_EXTRACT_OBJECT_2
1229123988	0x4942f194	BAD_OPERATION_EXTRACT_TYPECODE
1229123989	0x4942f195	BAD_OPERATION_EXTRACT_PRINCIPAL
1229123990	0x4942f196	BAD_OPERATION_EXTRACT_VALUE
1229123991	0x4942f197	BAD_OPERATION_GET_PRIMITIVE_TC_1
1229123992	0x4942f198	BAD_OPERATION_GET_PRIMITIVE_TC_2
1229123993	0x4942f199	BAD_OPERATION_INVOKE_NULL_PARAM_1
1229123994	0x4942f19a	BAD_OPERATION_INVOKE_NULL_PARAM_2
1229123995	0x4942f19b	BAD_OPERATION_INVOKE_DEFAULT_1
1229123996	0x4942f19c	BAD_OPERATION_INVOKE_DEFAULT_2
1229123997	0x4942f19d	BAD_OPERATION_UNKNOWN_BOOTSTRAP_METHOD
1229123998	0x4942f19e	BAD_OPERATION_EMPTY_ANY
1229123999	0x4942f19f	BAD_OPERATION_STUB_DISCONNECTED
1229124000	0x4942f1a0	BAD_OPERATION_TIE_DISCONNECTED
1229124001	0x4942f1a1	BAD_OPERATION_DELEGATE_DISCONNECTED
1229124097	0x4942f201	NULL_PARAM_1
1229124098	0x4942f202	NULL_PARAM_2
1229124099	0x4942f203	NULL_PARAM_3
1229124100	0x4942f204	NULL_PARAM_4

Table 34. Decimal minor exception codes 1229123841 to 1229124249 (continued)

1229124101	0x4942f205	NULL_PARAM_5
1229124102	0x4942f206	NULL_PARAM_6
1229124103	0x4942f207	NULL_PARAM_7
1229124104	0x4942f208	NULL_PARAM_8
1229124105	0x4942f209	NULL_PARAM_9
1229124106	0x4942f20a	NULL_PARAM_10
1229124107	0x4942f20b	NULL_PARAM_11
1229124108	0x4942f20c	NULL_PARAM_12
1229124109	0x4942f20d	NULL_PARAM_13
1229124110	0x4942f20e	NULL_PARAM_14
1229124111	0x4942f20f	NULL_PARAM_15
1229124112	0x4942f210	NULL_PARAM_16
1229124113	0x4942f211	NULL_PARAM_17
1229124114	0x4942f212	NULL_PARAM_18
1229124115	0x4942f213	NULL_PARAM_19
1229124116	0x4942f214	NULL_PARAM_20
1229124117	0x4942f215	NULL_IOR_OBJECT
1229124118	0x4942f216	NULL_PI_NAME
1229124119	0x4942f217	NULL_SC_DATA
1229124126	0x4942f21e	BAD_SERVANT_TYPE
1229124127	0x4942f21f	BAD_EXCEPTION
1229124128	0x4942f220	BAD_MODIFIER_LIST
1229124129	0x4942f221	NULL_PROP_MGR
1229124130	0x4942f222	INVALID_PROPERTY
1229124131	0x4942f223	ORBINITREF_FORMAT
1229124132	0x4942f224	ORBINITREF_MISSING_OBJECTURL
1229124133	0x4942f225	ORBDEFAULTINITREF_FORMAT
1229124134	0x4942f226	ORBDEFAULTINITREF_VALUE
1229124135	0x4942f227	OBJECTKEY_SERVERUUID_LENGTH
1229124136	0x4942f228	OBJECTKEY_SERVERUUID_NULL
1229124137	0x4942f229	BAD_REPOSITORY_ID
1229124138	0x4942f22a	BAD_PARAM_LOCAL_OBJECT
1229124139	0x4942f22b	NULL_OBJECT_IOR
1229124140	0x4942f22c	WRONG_ORB
1229124141	0x4942f22d	NULL_OBJECT_KEY
1229124142	0x4942f22e	NULL_OBJECT_URL
1229124143	0x4942f22f	NOT_A_NAMING_CONTEXT
1229124225	0x4942f281	TYPECODEIMPL_CTOR_MISUSE_1
1229124226	0x4942f282	TYPECODEIMPL_CTOR_MISUSE_2
1229124227	0x4942f283	TYPECODEIMPL_NULL_INDIRECTTYPE
1229124228	0x4942f284	TYPECODEIMPL_RECURSIVE_TYPECODES
1229124235	0x4942f28b	TYPECODEIMPL_KIND_INVALID_1
1229124236	0x4942f28c	TYPECODEIMPL_KIND_INVALID_2
1229124237	0x4942f28d	TYPECODEIMPL_NATIVE_1
1229124238	0x4942f28e	TYPECODEIMPL_NATIVE_2
1229124239	0x4942f28f	TYPECODEIMPL_NATIVE_3
1229124240	0x4942f290	TYPECODEIMPL_KIND_INDIRECT_1
1229124241	0x4942f291	TYPECODEIMPL_KIND_INDIRECT_2
1229124242	0x4942f292	TYPECODEIMPL_NULL_TYPECODE
1229124243	0x4942f293	TYPECODEIMPL_BODY_OF_TYPECODE
1229124244	0x4942f294	TYPECODEIMPL_KIND_RECURSIVE_1

Table 34. Decimal minor exception codes 1229123841 to 1229124249 (continued)

1229124245	0x4942f295	TYPECODEIMPL_COMPLEX_DEFAULT_1
1229124246	0x4942f296	TYPECODEIMPL_COMPLEX_DEFAULT_2
1229124247	0x4942f297	TYPECODEIMPL_INDIRECTION
1229124248	0x4942f298	TYPECODEIMPL_SIMPLE_DEFAULT
1229124249	0x4942f299	TYPECODEIMPL_NOT_CDROS

Table 35. Decimal minor exception codes 1229124250 to 1229125764

Decimal	Hexadecimal	Minor code reason
1229124250	0x4942f29a	TYPECODEIMPL_NO_IMPLEMENT_1
1229124251	0x4942f29b	TYPECODEIMPL_NO_IMPLEMENT_2
1229124357	0x4942f305	CONN_PURGE_REBIND
1229124358	0x4942f306	CONN_PURGE_ABORT
1229124359	0x4942f307	CONN_NOT_ESTABLISH
1229124360	0x4942f308	CONN_CLOSE_REBIND
1229124368	0x4942f310	WRITE_ERROR_SEND
1229124376	0x4942f318	GET_PROPERTIES_ERROR
1229124384	0x4942f320	BOOTSTRAP_SERVER_NOT_AVAIL
1229124392	0x4942f328	INVOKE_ERROR
1229124481	0x4942f381	BAD_HEX_DIGIT
1229124482	0x4942f382	BAD_STRINGIFIED_IOR_LEN
1229124483	0x4942f383	BAD_STRINGIFIED_IOR
1229124485	0x4942f385	BAD_MODIFIER_1
1229124486	0x4942f386	BAD_MODIFIER_2
1229124488	0x4942f388	CODESET_INCOMPATIBLE
1229124490	0x4942f38a	LONG_DOUBLE_NOT_IMPLEMENTED_1
1229124491	0x4942f38b	LONG_DOUBLE_NOT_IMPLEMENTED_2
1229124492	0x4942f38c	LONG_DOUBLE_NOT_IMPLEMENTED_3
1229124496	0x4942f390	COMPLEX_TYPES_NOT_IMPLEMENTED
1229124497	0x4942f391	VALUE_BOX_NOT_IMPLEMENTED
1229124498	0x4942f392	NULL_STRINGIFIED_IOR
1229124865	0x4942f501	TRANS_NS_CANNOT_CREATE_INITIAL_NC_SYS
1229124866	0x4942f502	TRANS_NS_CANNOT_CREATE_INITIAL_NC
1229124867	0x4942f503	GLOBAL_ORB_EXISTS
1229124868	0x4942f504	PLUGINS_ERROR
1229124869	0x4942f505	INCOMPATIBLE_JDK_VERSION
1229124993	0x4942f581	BAD_REPLYSTATUS
1229124994	0x4942f582	PEEKSTRING_FAILED
1229124995	0x4942f583	GET_LOCAL_HOST_FAILED
1229124996	0x4942f584	CREATE_LISTENER_FAILED
1229124997	0x4942f585	BAD_LOCATE_REQUEST_STATUS
1229124998	0x4942f586	STRINGIFY_WRITE_ERROR
1229125000	0x4942f588	BAD_GIOP_REQUEST_TYPE_1
1229125001	0x4942f589	BAD_GIOP_REQUEST_TYPE_2
1229125002	0x4942f58a	BAD_GIOP_REQUEST_TYPE_3
1229125003	0x4942f58b	BAD_GIOP_REQUEST_TYPE_4
1229125005	0x4942f58d	NULL_ORB_REFERENCE
1229125006	0x4942f58e	NULL_NAME_REFERENCE
1229125008	0x4942f590	ERROR_UNMARSHALING_USEREXC
1229125009	0x4942f591	SUBCONTRACTREGISTRY_ERROR
1229125010	0x4942f592	LOCATIONFORWARD_ERROR
1229125011	0x4942f593	BAD_READER_THREAD

Table 35. Decimal minor exception codes 1229124250 to 1229125764 (continued)

1229125013	0x4942f595	BAD_REQUEST_ID
1229125014	0x4942f596	BAD_SYSTEMEXCEPTION
1229125015	0x4942f597	BAD_COMPLETION_STATUS
1229125016	0x4942f598	INITIAL_REF_ERROR
1229125017	0x4942f599	NO_CODEC_FACTORY
1229125018	0x4942f59a	BAD_SUBCONTRACT_ID
1229125019	0x4942f59b	BAD_SYSTEMEXCEPTION_2
1229125020	0x4942f59c	NOT_PRIMITIVE_TYPECODE
1229125021	0x4942f59d	BAD_SUBCONTRACT_ID_2
1229125022	0x4942f59e	BAD_SUBCONTRACT_TYPE
1229125023	0x4942f59f	NAMING_CTX_REBIND_ALREADY_BOUND
1229125024	0x4942f5a0	NAMING_CTX_REBINDCTX_ALREADY_BOUND
1229125025	0x4942f5a1	NAMING_CTX_BAD_BINDINGTYPE
1229125026	0x4942f5a2	NAMING_CTX_RESOLVE_CANNOT_NARROW_TO_CTX
1229125032	0x4942f5a8	TRANS_NC_BIND_ALREADY_BOUND
1229125033	0x4942f5a9	TRANS_NC_LIST_GOT_EXC
1229125034	0x4942f5aa	TRANS_NC_NEWCTX_GOT_EXC
1229125035	0x4942f5ab	TRANS_NC_DESTROY_GOT_EXC
1229125036	0x4942f5ac	TRANS_BI_DESTROY_GOT_EXC
1229125042	0x4942f5b2	INVALID_CHAR_CODESET_1
1229125043	0x4942f5b3	INVALID_CHAR_CODESET_2
1229125044	0x4942f5b4	INVALID_WCHAR_CODESET_1
1229125045	0x4942f5b5	INVALID_WCHAR_CODESET_2
1229125046	0x4942f5b6	GET_HOST_ADDR_FAILED
1229125047	0x4942f5b7	REACHED_UNREACHABLE_PATH
1229125048	0x4942f5b8	PROFILE_CLONE_FAILED
1229125049	0x4942f5b9	INVALID_LOCATE_REQUEST_STATUS
1229125050	0x4942f5ba	NO_UNSAFE_CLASS
1229125051	0x4942f5bb	REQUEST_WITHOUT_CONNECTION_1
1229125052	0x4942f5bc	REQUEST_WITHOUT_CONNECTION_2
1229125053	0x4942f5bd	REQUEST_WITHOUT_CONNECTION_3
1229125054	0x4942f5be	REQUEST_WITHOUT_CONNECTION_4
1229125055	0x4942f5bf	REQUEST_WITHOUT_CONNECTION_5
1229125056	0x4942f5c0	NOT_USED_1
1229125057	0x4942f5c1	NOT_USED_2
1229125058	0x4942f5c2	NOT_USED_3
1229125059	0x4942f5c3	NOT_USED_4
1229125512	0x4942f788	BAD_CODE_SET
1229125520	0x4942f790	INV_RMI_STUB
1229125521	0x4942f791	INV_LOAD_STUB
1229125522	0x4942f792	INV_OBJ_IMPLEMENTATION
1229125523	0x4942f793	OBJECTKEY_NOMAGIC
1229125524	0x4942f794	OBJECTKEY_NOSCID
1229125525	0x4942f795	OBJECTKEY_NOSERVERID
1229125526	0x4942f796	OBJECTKEY_NOSERVERUUID
1229125527	0x4942f797	OBJECTKEY_SERVERUUIDKEY
1229125528	0x4942f798	OBJECTKEY_NOPOANAME
1229125529	0x4942f799	OBJECTKEY_SETSCID
1229125530	0x4942f79a	OBJECTKEY_SETSERVERID
1229125531	0x4942f79b	OBJECTKEY_NOUSERKEY
1229125532	0x4942f79c	OBJECTKEY_SETUSERKEY

Table 35. Decimal minor exception codes 1229124250 to 1229125764 (continued)

1229125533	0x4942f79d	INVALID_INDEXED_PROFILE_1
1229125534	0x4942f79e	INVALID_INDEXED_PROFILE_2
1229125762	0x4942f882	UNSPECIFIED_MARSHAL_1
1229125763	0x4942f883	UNSPECIFIED_MARSHAL_2
1229125764	0x4942f884	UNSPECIFIED_MARSHAL_3

Table 36. Decimal minor exception codes 1299125765 to 1229125906

Decimal	Hexadecimal	Minor code reason
1229125765	0x4942f885	UNSPECIFIED_MARSHAL_4
1229125766	0x4942f886	UNSPECIFIED_MARSHAL_5
1229125767	0x4942f887	UNSPECIFIED_MARSHAL_6
1229125768	0x4942f888	UNSPECIFIED_MARSHAL_7
1229125769	0x4942f889	UNSPECIFIED_MARSHAL_8
1229125770	0x4942f88a	UNSPECIFIED_MARSHAL_9
1229125771	0x4942f88b	UNSPECIFIED_MARSHAL_10
1229125772	0x4942f88c	UNSPECIFIED_MARSHAL_11
1229125773	0x4942f88d	UNSPECIFIED_MARSHAL_12
1229125774	0x4942f88e	UNSPECIFIED_MARSHAL_13
1229125775	0x4942f88f	UNSPECIFIED_MARSHAL_14
1229125776	0x4942f890	UNSPECIFIED_MARSHAL_15
1229125777	0x4942f891	UNSPECIFIED_MARSHAL_16
1229125778	0x4942f892	UNSPECIFIED_MARSHAL_17
1229125779	0x4942f893	UNSPECIFIED_MARSHAL_18
1229125780	0x4942f894	UNSPECIFIED_MARSHAL_19
1229125781	0x4942f895	UNSPECIFIED_MARSHAL_20
1229125782	0x4942f896	UNSPECIFIED_MARSHAL_21
1229125783	0x4942f897	UNSPECIFIED_MARSHAL_22
1229125784	0x4942f898	UNSPECIFIED_MARSHAL_23
1229125785	0x4942f899	UNSPECIFIED_MARSHAL_24
1229125786	0x4942f89a	UNSPECIFIED_MARSHAL_25
1229125787	0x4942f89b	UNSPECIFIED_MARSHAL_26
1229125788	0x4942f89c	UNSPECIFIED_MARSHAL_27
1229125789	0x4942f89d	UNSPECIFIED_MARSHAL_28
1229125790	0x4942f89e	UNSPECIFIED_MARSHAL_29
1229125791	0x4942f89f	UNSPECIFIED_MARSHAL_30
1229125792	0x4942f8a0	UNSPECIFIED_MARSHAL_31
1229125793	0x4942f8a1	UNSPECIFIED_MARSHAL_32
1229125794	0x4942f8a2	UNSPECIFIED_MARSHAL_33
1229125795	0x4942f8a3	UNSPECIFIED_MARSHAL_34
1229125796	0x4942f8a4	UNSPECIFIED_MARSHAL_35
1229125797	0x4942f8a5	UNSPECIFIED_MARSHAL_36
1229125798	0x4942f8a6	UNSPECIFIED_MARSHAL_37
1229125799	0x4942f8a7	UNSPECIFIED_MARSHAL_38
1229125800	0x4942f8a8	UNSPECIFIED_MARSHAL_39
1229125801	0x4942f8a9	UNSPECIFIED_MARSHAL_40
1229125802	0x4942f8aa	UNSPECIFIED_MARSHAL_41
1229125803	0x4942f8ab	UNSPECIFIED_MARSHAL_42
1229125804	0x4942f8ac	UNSPECIFIED_MARSHAL_43
1229125805	0x4942f8ad	UNSPECIFIED_MARSHAL_44
1229125806	0x4942f8ae	UNSPECIFIED_MARSHAL_45
1229125807	0x4942f8af	UNSPECIFIED_MARSHAL_46

Table 36. Decimal minor exception codes 1299125765 to 1229125906 (continued)

1229125808	0x4942f8b0	UNSPECIFIED_MARSHAL_47
1229125809	0x4942f8b1	UNSPECIFIED_MARSHAL_48
1229125810	0x4942f8b2	UNSPECIFIED_MARSHAL_49
1229125811	0x4942f8b3	UNSPECIFIED_MARSHAL_50
1229125812	0x4942f8b4	UNSPECIFIED_MARSHAL_51
1229125813	0x4942f8b5	UNSPECIFIED_MARSHAL_52
1229125814	0x4942f8b6	UNSPECIFIED_MARSHAL_53
1229125815	0x4942f8b7	UNSPECIFIED_MARSHAL_54
1229125816	0x4942f8b8	UNSPECIFIED_MARSHAL_55
1229125817	0x4942f8b9	UNSPECIFIED_MARSHAL_56
1229125818	0x4942f8ba	UNSPECIFIED_MARSHAL_57
1229125819	0x4942f8bb	UNSPECIFIED_MARSHAL_58
1229125820	0x4942f8bc	UNSPECIFIED_MARSHAL_59
1229125821	0x4942f8bd	UNSPECIFIED_MARSHAL_60
1229125822	0x4942f8be	UNSPECIFIED_MARSHAL_61
1229125823	0x4942f8bf	UNSPECIFIED_MARSHAL_62
1229125824	0x4942f8c0	UNSPECIFIED_MARSHAL_63
1229125825	0x4942f8c1	UNSPECIFIED_MARSHAL_64
1229125826	0x4942f8c2	UNSPECIFIED_MARSHAL_65
1229125827	0x4942f8c3	UNSPECIFIED_MARSHAL_66
1229125828	0x4942f8c4	READ_OBJECT_EXCEPTION_2
1229125841	0x4942f8d1	UNSUPPORTED_IDLTYPE
1229125842	0x4942f8d2	DSI_RESULT_EXCEPTION
1229125844	0x4942f8d4	IIOPINPUTSTREAM_GROW
1229125847	0x4942f8d7	NO_CHAR_CONVERTER_1
1229125848	0x4942f8d8	NO_CHAR_CONVERTER_2
1229125849	0x4942f8d9	CHARACTER_MALFORMED_1
1229125850	0x4942f8da	CHARACTER_MALFORMED_2
1229125851	0x4942f8db	CHARACTER_MALFORMED_3
1229125852	0x4942f8dc	CHARACTER_MALFORMED_4
1229125854	0x4942f8de	INCORRECT_CHUNK_LENGTH
1229125856	0x4942f8e0	CHUNK_OVERFLOW
1229125858	0x4942f8e2	CANNOT_GROW
1229125859	0x4942f8e3	CODESET_ALREADY_SET
1229125860	0x4942f8e4	REQUEST_CANCELLED
1229125861	0x4942f8e5	WRITE_TO_STREAM_1
1229125862	0x4942f8e6	WRITE_TO_STREAM_2
1229125863	0x4942f8e7	WRITE_TO_STREAM_3
1229125864	0x4942f8e8	WRITE_TO_STREAM_4
1229125865	0x4942f8e9	PROXY_MARSHAL_FAILURE
1229125866	0x4942f8ea	PROXY_UNMARSHAL_FAILURE
1229125867	0x4942f8eb	INVALID_START_VALUE
1229125868	0x4942f8ec	INVALID_CHUNK_STATE
1229125869	0x4942f8ed	NULL_CODEBASE_IOR
1229125889	0x4942f901	DSI_NOT_IMPLEMENTED
1229125890	0x4942f902	GETINTERFACE_NOT_IMPLEMENTED
1229125891	0x4942f903	SEND_DEFERRED_NOTIMPLEMENTED
1229125893	0x4942f905	ARGUMENTS_NOTIMPLEMENTED
1229125894	0x4942f906	RESULT_NOTIMPLEMENTED
1229125895	0x4942f907	EXCEPTIONS_NOTIMPLEMENTED
1229125896	0x4942f908	CONTEXTLIST_NOTIMPLEMENTED

Table 36. Decimal minor exception codes 1299125765 to 1229125906 (continued)

1229125902	0x4942f90e	CREATE_OBJ_REF_BYTE_NOTIMPLEMENTED
1229125903	0x4942f90f	CREATE_OBJ_REF_IOR_NOTIMPLEMENTED
1229125904	0x4942f910	GET_KEY_NOTIMPLEMENTED
1229125905	0x4942f911	GET_IMPL_ID_NOTIMPLEMENTED
1229125906	0x4942f912	GET_SERVANT_NOTIMPLEMENTED

Table 37. Decimal minor exception codes 1229125907 to 1229126567

Decimal	Hexadecimal	Minor code reason
1229125907	0x4942f913	SET_ORB_NOTIMPLEMENTED
1229125908	0x4942f914	SET_ID_NOTIMPLEMENTED
1229125909	0x4942f915	GET_CLIENT_SUBCONTRACT_NOTIMPLEMENTED
1229125913	0x4942f919	CONTEXTIMPL_NOTIMPLEMENTED
1229125914	0x4942f91a	CONTEXT_NAME_NOTIMPLEMENTED
1229125915	0x4942f91b	PARENT_NOTIMPLEMENTED
1229125916	0x4942f91c	CREATE_CHILD_NOTIMPLEMENTED
1229125917	0x4942f91d	SET_ONE_VALUE_NOTIMPLEMENTED
1229125918	0x4942f91e	SET_VALUES_NOTIMPLEMENTED
1229125919	0x4942f91f	DELETE_VALUES_NOTIMPLEMENTED
1229125920	0x4942f920	GET_VALUES_NOTIMPLEMENTED
1229125922	0x4942f922	GET_CURRENT_NOTIMPLEMENTED_1
1229125923	0x4942f923	GET_CURRENT_NOTIMPLEMENTED_2
1229125924	0x4942f924	CREATE_OPERATION_LIST_NOTIMPLEMENTED_1
1229125925	0x4942f925	CREATE_OPERATION_LIST_NOTIMPLEMENTED_2
1229125926	0x4942f926	GET_DEFAULT_CONTEXT_NOTIMPLEMENTED_1
1229125927	0x4942f927	GET_DEFAULT_CONTEXT_NOTIMPLEMENTED_2
1229125928	0x4942f928	SHUTDOWN_NOTIMPLEMENTED
1229125929	0x4942f929	WORK_PENDING_NOTIMPLEMENTED
1229125930	0x4942f92a	PERFORM_WORK_NOTIMPLEMENTED
1229125931	0x4942f92b	COPY_TK_ABSTRACT_NOTIMPLEMENTED
1229125932	0x4942f92c	PI_CLIENT_GET_POLICY_NOTIMPLEMENTED
1229125933	0x4942f92d	PI_SERVER_GET_POLICY_NOTIMPLEMENTED
1229125934	0x4942f92e	ADDRESSING_MODE_NOTIMPLEMENTED_1
1229125935	0x4942f92f	ADDRESSING_MODE_NOTIMPLEMENTED_2
1229125936	0x4942f930	SET_OBJECT_RESOLVER_NOTIMPLEMENTED
1229125937	0x4942f931	DISCONNECTED_SERVANT_1
1229125938	0x4942f932	DISCONNECTED_SERVANT_2
1229125939	0x4942f933	DISCONNECTED_SERVANT_3
1229125940	0x4942f934	DISCONNECTED_SERVANT_4
1229125941	0x4942f935	DISCONNECTED_SERVANT_5
1229125942	0x4942f936	DISCONNECTED_SERVANT_6
1229125943	0x4942f937	DISCONNECTED_SERVANT_7
1229125944	0x4942f938	GET_INTERFACE_DEF_NOT_IMPLEMENTED
1229126017	0x4942f981	MARSHAL_NO_MEMORY_1
1229126018	0x4942f982	MARSHAL_NO_MEMORY_2
1229126019	0x4942f983	MARSHAL_NO_MEMORY_3
1229126020	0x4942f984	MARSHAL_NO_MEMORY_4
1229126021	0x4942f985	MARSHAL_NO_MEMORY_5
1229126022	0x4942f986	MARSHAL_NO_MEMORY_6
1229126023	0x4942f987	MARSHAL_NO_MEMORY_7
1229126024	0x4942f988	MARSHAL_NO_MEMORY_8
1229126025	0x4942f989	MARSHAL_NO_MEMORY_9

Table 37. Decimal minor exception codes 1229125907 to 1229126567 (continued)

1229126026	0x4942f98a	MARSHAL_NO_MEMORY_10
1229126027	0x4942f98b	MARSHAL_NO_MEMORY_11
1229126028	0x4942f98c	MARSHAL_NO_MEMORY_12
1229126029	0x4942f98d	MARSHAL_NO_MEMORY_13
1229126030	0x4942f98e	MARSHAL_NO_MEMORY_14
1229126031	0x4942f98f	MARSHAL_NO_MEMORY_15
1229126032	0x4942f990	MARSHAL_NO_MEMORY_16
1229126033	0x4942f991	MARSHAL_NO_MEMORY_17
1229126034	0x4942f992	MARSHAL_NO_MEMORY_18
1229126035	0x4942f993	MARSHAL_NO_MEMORY_19
1229126036	0x4942f994	MARSHAL_NO_MEMORY_20
1229126037	0x4942f995	MARSHAL_NO_MEMORY_21
1229126038	0x4942f996	MARSHAL_NO_MEMORY_22
1229126039	0x4942f997	MARSHAL_NO_MEMORY_23
1229126040	0x4942f998	MARSHAL_NO_MEMORY_24
1229126041	0x4942f999	MARSHAL_NO_MEMORY_25
1229126042	0x4942f99a	MARSHAL_NO_MEMORY_26
1229126043	0x4942f99b	MARSHAL_NO_MEMORY_27
1229126044	0x4942f99c	MARSHAL_NO_MEMORY_28
1229126045	0x4942f99d	MARSHAL_NO_MEMORY_29
1229126046	0x4942f99e	MARSHAL_NO_MEMORY_30
1229126047	0x4942f99f	MARSHAL_NO_MEMORY_31
1229126401	0x4942fb01	RESPONSE_TIMED_OUT
1229126402	0x4942fb02	FRAGMENT_TIMED_OUT
1229126529	0x4942fb81	NO_SERVER_SC_IN_DISPATCH
1229126530	0x4942fb82	NO_SERVER_SC_IN_LOOKUP
1229126531	0x4942fb83	NO_SERVER_SC_IN_CREATE_DEFAULT_SERVER
1229126532	0x4942fb84	NO_SERVER_SC_IN_SETUP
1229126533	0x4942fb85	NO_SERVER_SC_IN_LOCATE
1229126534	0x4942fb86	NO_SERVER_SC_IN_DISCONNECT
1229126539	0x4942fb8b	ORB_CONNECT_ERROR_1
1229126540	0x4942fb8c	ORB_CONNECT_ERROR_2
1229126541	0x4942fb8d	ORB_CONNECT_ERROR_3
1229126542	0x4942fb8e	ORB_CONNECT_ERROR_4
1229126543	0x4942fb8f	ORB_CONNECT_ERROR_5
1229126544	0x4942fb90	ORB_CONNECT_ERROR_6
1229126545	0x4942fb91	ORB_CONNECT_ERROR_7
1229126546	0x4942fb92	ORB_CONNECT_ERROR_8
1229126547	0x4942fb93	ORB_CONNECT_ERROR_9
1229126548	0x4942fb94	ORB_REGISTER_1
1229126549	0x4942fb95	ORB_REGISTER_2
1229126553	0x4942fb99	ORB_REGISTER_LOCAL_1
1229126554	0x4942fb9a	ORB_REGISTER_LOCAL_2
1229126555	0x4942fb9b	LOCAL_SERVANT_LOOKUP
1229126556	0x4942fb9c	POA_LOOKUP_ERROR
1229126557	0x4942fb9d	POA_INACTIVE
1229126558	0x4942fb9e	POA_NO_SERVANT_MANAGER
1229126559	0x4942fb9f	POA_NO_DEFAULT_SERVANT
1229126560	0x4942fba0	POA_WRONG_POLICY
1229126561	0x4942fba1	FINDPOA_ERROR
1229126562	0x4942fba2	ADAPTER_ACTIVATOR_EXCEPTION

Table 37. Decimal minor exception codes 1229125907 to 1229126567 (continued)

1229126563	0x4942fba3	POA_SERVANT_ACTIVATOR_LOOKUP_FAILED
1229126564	0x4942fba4	POA_BAD_SERVANT_MANAGER
1229126565	0x4942fba5	POA_SERVANT_LOCATOR_LOOKUP_FAILED
1229126566	0x4942fba6	POA_UNKNOWN_POLICY
1229126567	0x4942fba7	POA_NOT_FOUND

Table 38. Decimal minor exception codes 1229126568 to 1330446377

Decimal	Hexadecimal	Minor code reason
1229126568	0x4942fba8	SERVANT_LOOKUP
1229126569	0x4942fba9	SERVANT_IS_ACTIVE
1229126570	0x4942fbaa	SERVANT_DISPATCH
1229126571	0x4942fbab	WRONG_CLIENTSC
1229126572	0x4942fbac	WRONG_SERVERSC
1229126573	0x4942fbad	SERVANT_IS_NOT_ACTIVE
1229126574	0x4942fbae	POA_WRONG_POLICY_1
1229126575	0x4942fbaf	POA_NOCONTEXT_1
1229126576	0x4942fbb0	POA_NOCONTEXT_2
1229126577	0x4942fbb1	POA_INVALID_NAME_1
1229126578	0x4942fbb2	POA_INVALID_NAME_2
1229126579	0x4942fbb3	POA_INVALID_NAME_3
1229126580	0x4942fbb4	POA_NOCONTEXT_FOR_PREINVOKE
1229126581	0x4942fbb5	POA_NOCONTEXT_FOR_CHECKING_PREINVOKE
1229126582	0x4942fbb6	POA_NOCONTEXT_FOR_POSTINVOKE
1229126657	0x4942fc01	LOCATE_UNKNOWN_OBJECT
1229126658	0x4942fc02	BAD_SERVER_ID_1
1229126659	0x4942fc03	BAD_SERVER_ID_2
1229126660	0x4942fc04	BAD_IMPLID
1229126665	0x4942fc09	BAD_SKELETON_1
1229126666	0x4942fc0a	BAD_SKELETON_2
1229126673	0x4942fc11	SERVANT_NOT_FOUND_1
1229126674	0x4942fc12	SERVANT_NOT_FOUND_2
1229126675	0x4942fc13	SERVANT_NOT_FOUND_3
1229126676	0x4942fc14	SERVANT_NOT_FOUND_4
1229126677	0x4942fc15	SERVANT_NOT_FOUND_5
1229126678	0x4942fc16	SERVANT_NOT_FOUND_6
1229126679	0x4942fc17	SERVANT_NOT_FOUND_7
1229126687	0x4942fc1f	SERVANT_DISCONNECTED_1
1229126688	0x4942fc20	SERVANT_DISCONNECTED_2
1229126689	0x4942fc21	NULL_SERVANT
1229126690	0x4942fc22	ADAPTER_ACTIVATOR_FAILED
1229126692	0x4942fc24	ORB_DESTROYED
1229126693	0x4942fc25	DYNANY_DESTROYED
1229127170	0x4942fe02	CONNECT_FAILURE_1
1229127171	0x4942fe03	CONNECT_FAILURE_2
1229127172	0x4942fe04	CONNECT_FAILURE_3
1229127173	0x4942fe05	CONNECT_FAILURE_4
1229127297	0x4942fe81	UNKNOWN_CORBA_EXC
1229127298	0x4942fe82	RUNTIMEEXCEPTION
1229127299	0x4942fe83	UNKNOWN_SERVER_ERROR
1229127300	0x4942fe84	UNKNOWN_DSI_SYSEX
1229127301	0x4942fe85	UNEXPECTED_CHECKED_EXCEPTION

Table 38. Decimal minor exception codes 1229126568 to 1330446377 (continued)

1229127302	0x4942fe86	UNKNOWN_CREATE_EXCEPTION_RESPONSE
1229127312	0x4942fe90	UNKNOWN_PI_EXC
1229127313	0x4942fe91	UNKNOWN_PI_EXC_2
1229127314	0x4942fe92	PI_ARGS_FAILURE
1229127315	0x4942fe93	PI_EXCEPTS_FAILURE
1229127316	0x4942fe94	PI_CONTEXTS_FAILURE
1229127317	0x4942fe95	PI_OP_CONTEXT_FAILURE
1229127326	0x4942fe9e	USER_DEFINED_ERROR
1229127327	0x4942fe9f	UNKNOWN_RUNTIME_IN_BOOTSTRAP
1229127328	0x4942fea0	UNKNOWN_THROWABLE_IN_BOOTSTRAP
1229127329	0x4942fea1	UNKNOWN_RUNTIME_IN_INSAGENT
1229127330	0x4942fea2	UNKNOWN_THROWABLE_IN_INSAGENT
1229127331	0x4942fea3	UNEXPECTED_IN_PROCESSING_CLIENTSIDE_INTERCEPTOR
1229127332	0x4942fea4	UNEXPECTED_IN_PROCESSING_SERVERSIDE_INTERCEPTOR
1229127333	0x4942fea5	UNEXPECTED_PI_LOCAL_REQUEST
1229127334	0x4942fea6	UNEXPECTED_PI_LOCAL_RESPONSE
1330446336	0x4f4d0000	OMGVMCID
1330446337	0x4f4d0001	FAILURE_TO_REGISTER_OR_LOOKUP_VALUE_FACTORY
1330446338	0x4f4d0002	RID_ALREADY_DEFINED_IN_IFR
1330446339	0x4f4d0003	IN_INVOCATION_CONTEXT
1330446340	0x4f4d0004	ORB_SHUTDOWN
1330446341	0x4f4d0005	NAME_CLASH_IN_INHERITED_CONTEXT
1330446342	0x4f4d0006	SERVANT_MANAGER_EXISTS
1330446343	0x4f4d0007	INS_BAD_SCHEME_NAME
1330446344	0x4f4d0008	INS_BAD_ADDRESS
1330446345	0x4f4d0009	INS_BAD_SCHEME_SPECIFIC_PART
1330446346	0x4f4d000a	INS_OTHER
1330446348	0x4f4d000c	POLICY_FACTORY_EXISTS
1330446350	0x4f4d000e	INVALID_PI_CALL
1330446351	0x4f4d000f	SERVICE_CONTEXT_ID_EXISTS
1330446353	0x4f4d0011	POA_DESTROYED
1330446359	0x4f4d0017	NO_TRANSMISSION_CODE
1330446362	0x4f4d001a	INVALID_SERVICE_CONTEXT
1330446363	0x4f4d001b	NULL_OBJECT_ON_REGISTER
1330446364	0x4f4d001c	INVALID_COMPONENT_ID
1330446365	0x4f4d001d	INVALID_IOR_PROFILE
1330446375	0x4f4d0027	INVALID_STREAM_FORMAT_1
1330446376	0x4f4d0028	NOT_VALUE_OUTPUT_STREAM
1330446377	0x4f4d0029	NOT_VALUE_INPUT_STREAM

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 do not 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. You should also refer to this page before opening a PMR because it contains documents that can save you time gathering information needed to resolve a problem.

Transactions

Using the transaction service

These topics provide information about using transactions with WebSphere applications

WebSphere 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.

In WebSphere Application Server, transactions are handled by three main components:
- A transaction manager. The transaction manager supports the enlistment of recoverable XAResources and ensures that each such resource is driven to a consistent outcome either at the end of a transaction or after a failure and restart of the application server.
- A container in which the J2EE 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. This allows such application components to control the demarcation of their own transactions.

For more information about using transactions with WebSphere applications, see the following topics:
- "Transaction support in WebSphere Application Server"
- "Developing components to use transactions" on page 1110
- Configuring transaction properties for an application server
- Configuring transaction properties for peer recovery
- Managing manual peer recovery of the transaction service
- "Use of local transactions" on page 1105
- Managing active and prepared transactions
- Managing transaction logging for optimum server availability
- Interoperating transactionally between application servers
- Configuring an intermediary node for Web services transactions
- Enabling WebSphere Application Server to use an intermediary node for Web services transactions
- Configuring a server to use business activity support
- "Creating an application that exploits the business activity support" on page 1114
- "The business activity API" on page 1107
- Troubleshooting transactions
- "Transaction service exceptions" on page 1109
- "UserTransaction interface - methods available" on page 1110
- "Using one-phase and two-phase commit resources in the same transaction" on page 1116

- "Using the ActivitySession service" on page 1120

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, multiple SQL statements to a relational database are committed atomically by the database during the processing of an SQL COMMIT statement. 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, then an external transaction manager is required to coordinate the individual resource managers. A transaction that spans multiple resource managers are

referred to as a *global transaction*. WebSphere Application Server is a transaction manager that can coordinate global transactions, be a participant in a received global transaction and also provides 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 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.

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 J2EE 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.

Resource managers that offer transaction support can be categorized into those that support two-phase coordination (by offering an XAResource interface) 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 of 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 J2EE programming model. EJBs 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.

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 J2EE Connector Architecture 1.0.
- 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. J2EE connector resource adapters that include support for local transactions provide a LocalTransaction interface to enable applications to request that the resource adapter commit or rollback 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, then an external transaction manager is needed to coordinate the updates to both resource managers in a *global tansaction*.

Global transaction support is available to web and enterprise bean J2EE components and, with some limitation, to application client components. Enterprise bean components can be subdivided into beans that exploit container-managed transactions (CMT) or 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. The UserTransaction interface can be obtained by a JNDI lookup of `java:comp/UserTransaction` or from the SessionContext object using the getUserTransaction method..

The UserTransaction is not available to the following components:
- CMT enterprise beans. Any attempt by such beans to obtain the interface results in an exception in accordance with the 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 appropriate for the EJB version.

Before the EJB 1.1 specification, the JNDI location of the UserTransaction interface was not specified. Each EJB container implementor defined it in an implementation-specific manner. Earlier versions of WebSphere Application Server, up to and including Version 3.5.x (without EJB 1.1), bind the UserTransaction interface to a JNDI location of jta/usertransaction. WebSphere Application Server Version 4, and later releases, bind the UserTransaction interface at the location defined by EJB 1.1, which is java:comp/UserTransaction. WebSphere Application Server, from Version 5 no longer provides the jta/usertransaction binding within Web and EJB containers to applications at a J2EE level of 1.3 or later. For example, from EJB 2.0 applications can use only the java:comp/UserTransaction location.

A web component or enterprise bean (CMT or BMT) can get the ExtendedJTATransaction interface through a lookup of `java:comp/websphere/ExtendedJTATransaction`. This interface provides access to the transaction identity and a mechanism to receive notification of transaction completion.

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 2.0 (or later) specification; for example, at http://java.sun.com/products/ejb/2.0.html.)

A LTC is a bounded unit-of-work scope within which zero, one, or more resource manager local transactions (RMLTs) 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. 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 a J2EE 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; rather LTCs are managed exclusively by the container and configured by the application deployer through transaction attributes in the application deployment descriptor.

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 for J2EE components at J2EE 1.3 or later. The only exceptions to this are as follows:
- Where application component dispatch occurs without container interposition; for example, for a stateless session bean `create` or a servlet-initiated thread.
- J2EE 1.2 web components.
- J2EE 1.2 BMT enterprise beans.

A local transaction containment can be scoped to an ActivitySession context that lives 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 JDBC data sources or connection factories, that are configured through the Resources view of the WebSphere Application Server 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 (RMLTs), but an XA data source can support two-phase commit coordination, as well as local transactions.

If an application uses two or more resource providers that support only RMLTs, then 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 and should access them within a global transaction.

If an application uses only one RMLT, the atomic behavior can be guaranteed by the resource manager, which can be accessed under a local transaction containment context.

An application can also access a single resource manager under a global transaction context, even if that resource manager does not support the XA coordination. An application can do this, because WebSphere Application Server performs an "only resource optimization" and interacts with the resource manager under a RMLT. Within 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 never both. An instance of an enterprise bean can change from running under 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 a J2EE client container can explicitly demarcate transaction boundaries as described in Using component-managed transactions. Application clients cannot perform, directly within 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, within 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 Application Server Toolkit information, in the navigation pane of this infocenter.

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.

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.
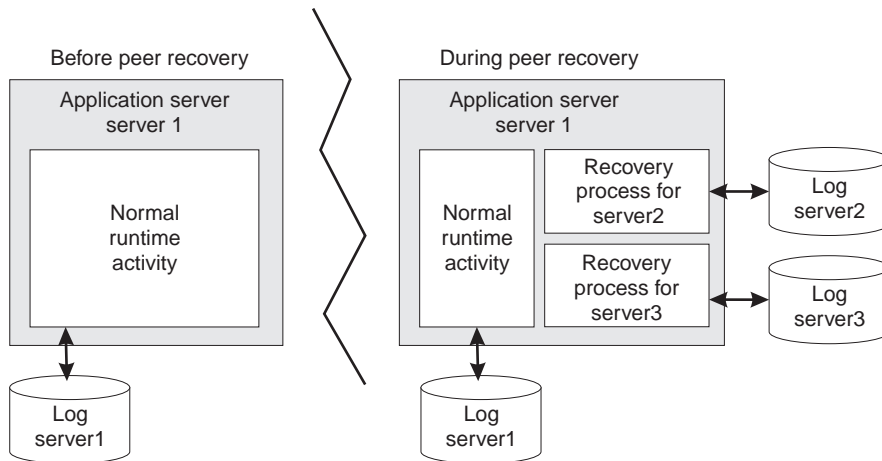
Figure 13. Peer recovery

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.
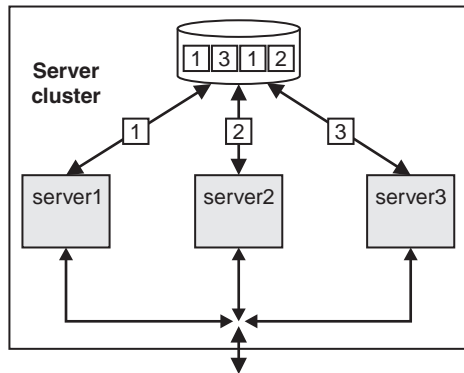
Figure 14. Server cluster up and running, just before server failure

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.
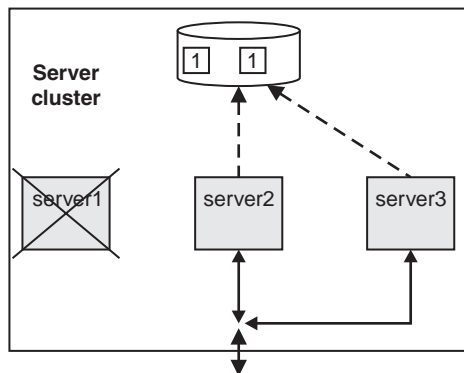
Figure 15. Server 1 fails. Servers 2 and 3 become blocked as a result.

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.
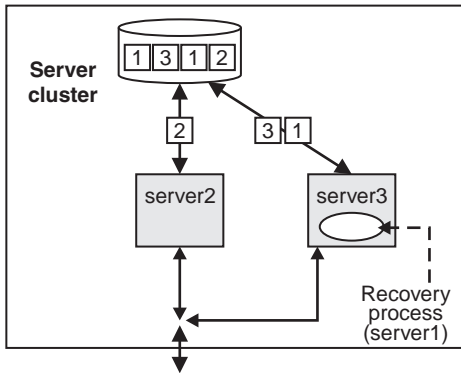
Figure 16. Peer recovery process started in server 3

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.
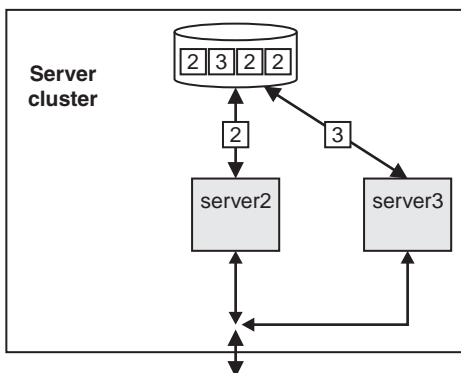
Figure 17. Server cluster stable again with just two servers: server 2 and server 3.

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. 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.
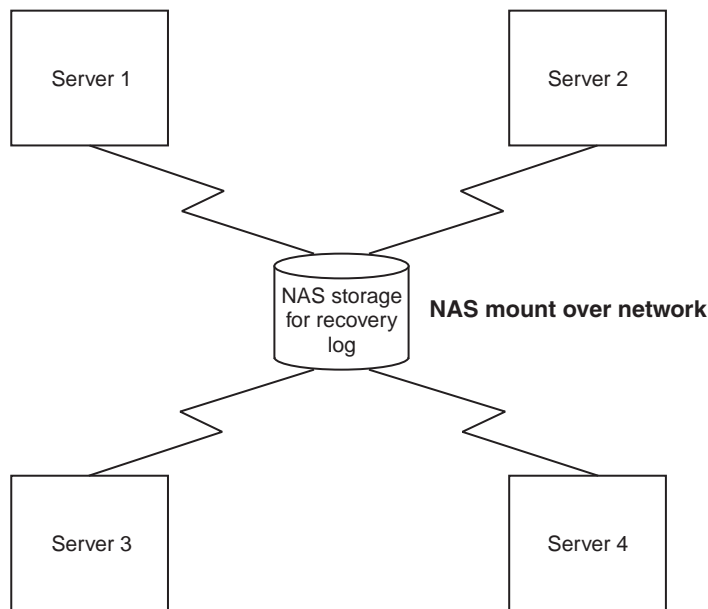
Figure 18. Recovery logs on NAS storage are available to all servers

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.
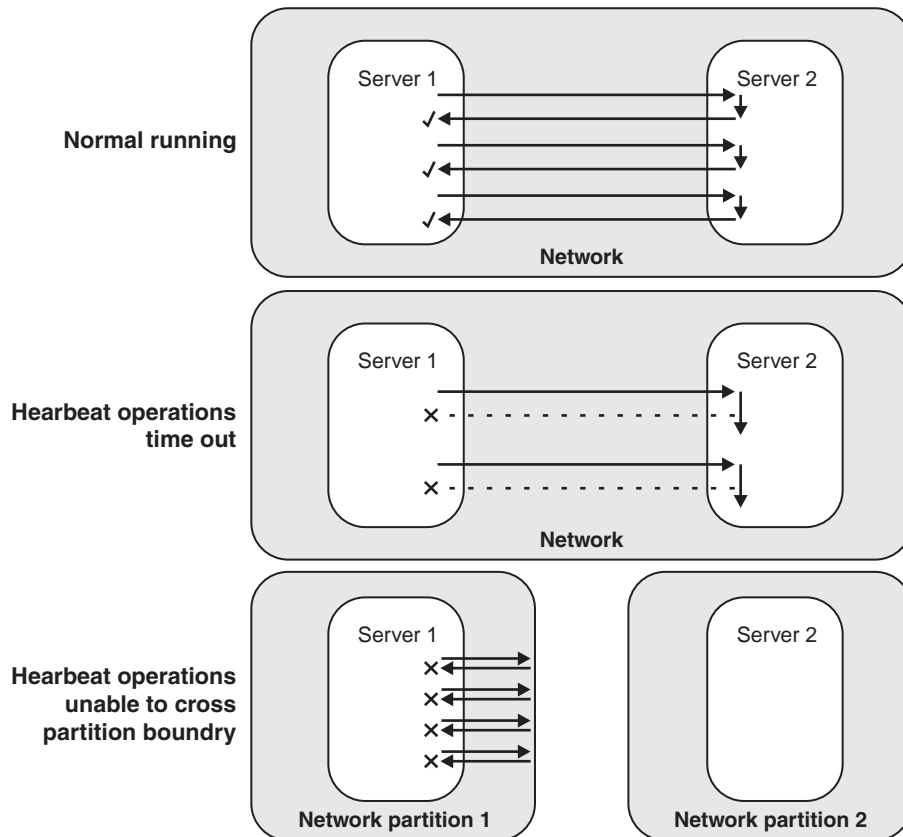
During normal running, two servers on the network exchange heartbeats. During system overloading, heartbeat operations time out, giving the appearance of a server failure. After network partitioning, each server is in a separate network and heartbeats cannot pass between them, also giving the appearance of a server failure.

Figure 19. 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 due to 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 compensatable work. 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.

WebSphere Application Server

Warehouse service

Supplier Web service

Stateless session bean

TX_Required

Compensation handler

Business activity scope

Stateless session bean

TX_NotSupported

Retailer service

Stateless session bean

TX_Required

Compensation handler

Manufacturing service
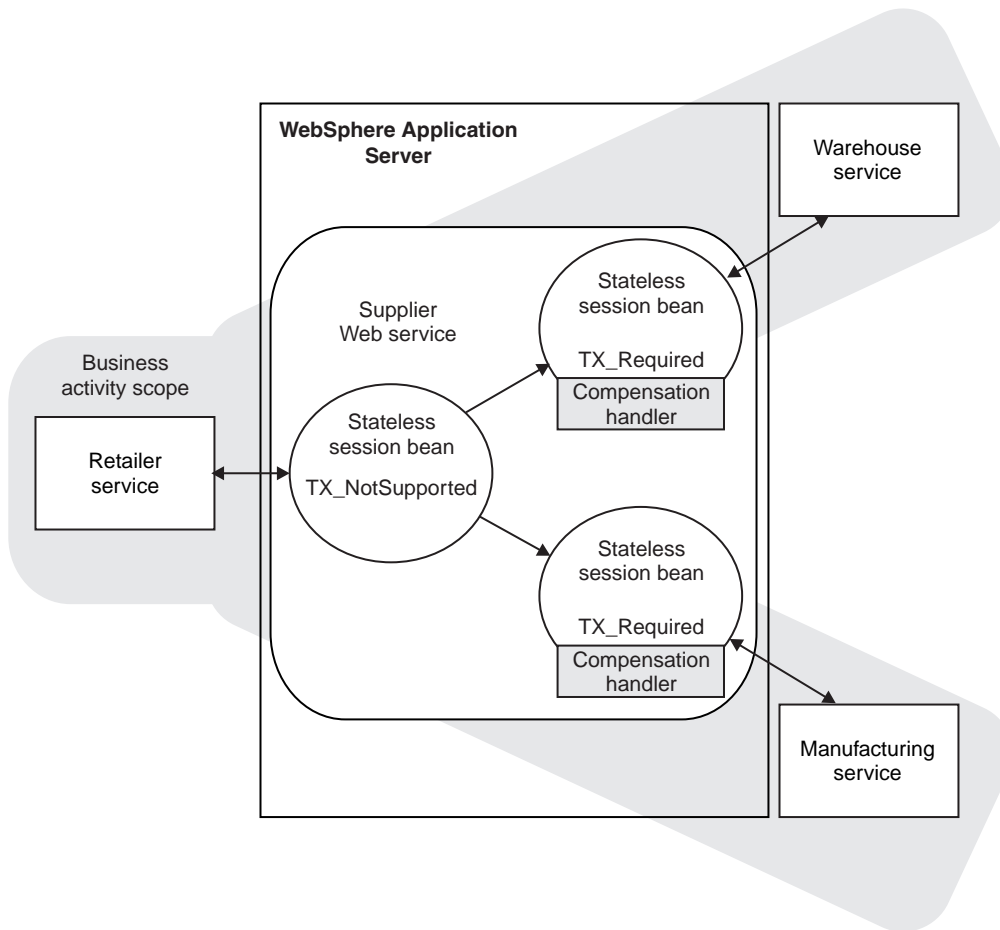
Application design considerations

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 `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 Version 6.1 server. 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.

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.
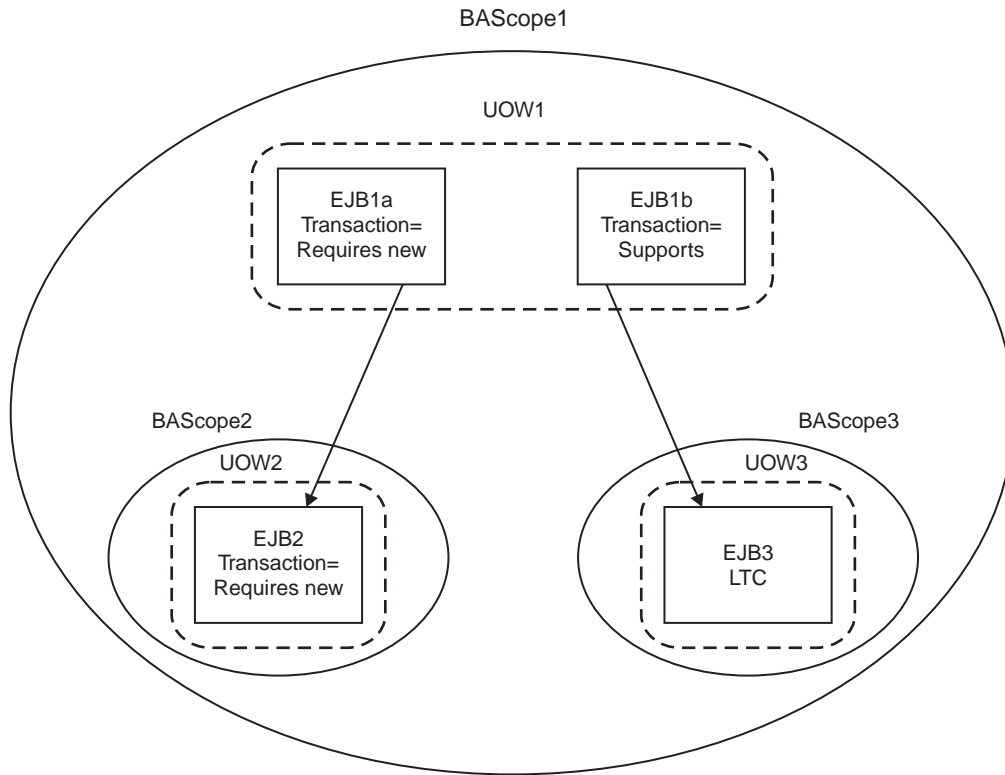
Inner business activity scopes must complete before the outer business activity scope completes. Inner business activity scopes, for example BAScope2, have an association with the outer business activity scope, in this case BAScope1. Each business activity scope is directed to close if its associated 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 39. 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 be inactive initially, 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 "The business activity API" on page 1107.

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 exploits the business activity support" on page 1114.

The effect of application server shutdown on active transactions and later recovery: When an application server shuts down, any active transactions are rolled back. If all transactions are successfully completed in this way, message WTRN0105I is logged, and on the next server restart no recovery activity is needed. If message CWWTR0105I is not logged for an application server shutdown, this does not indicate that there has been a failure, only that recovery activity is required when the server restarts.

A clean shutdown of all application servers should be achieved before the product is uninstalled, to avoid data integrity problems.

Extended JTA support: Extended JTA support provides application programming interfaces additional to the UserTransaction interface that is defined in the JTA as part of the J2EE specification. Specifically, the API extensions provide the following functionality:

- Access to global and local transaction identifiers associated with the thread.

  The global id is based on the tid in CosTransactions::PropagationContext: and the local id identifies the transaction uniquely within the local JVM.

- A transaction synchronization callback that enables any J2EE component to register an interest in transaction completion.

  This can be used by advanced applications to flush updates before transaction completion and clear up state after transaction completion. J2EE (and related) specifications position this function generally as the domain of the J2EE containers.

An application uses a JNDI lookup of java:comp/websphere/ExtendedJTATransaction to get an ExtendedJTATransaction object, which it then uses as follows:

```
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 SynchronizationCallbacks. 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.

The following information provides an overview of the interfaces provided by the Extended JTA support. For more detailed information, see the Javadoc.

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 in a J2EE server, there is no specific J2EE component active when this object is called. So, the object has limited direct access to any J2EE resources. Specifically, it has no access to the java: namespace or to any container-mediated resource. Such an object can cache a reference to a J2EE component (for example, a stateless session bean) that it delegates to. The object would then have all the normal access to J2EE resources and could be used, for example, to acquire a JDBC connection and flush updates to a database during beforeCompletion.

ExtendedJTATransaction interface

A WebSphere programming model extension to the J2EE JTA support. An object implementing this interface is bound, by WebSphere J2EE containers that support this interface, at java:comp/websphere/ExtendedJTATransaction. Access to this object, when called from an EJB container, is not restricted to component-managed transactions.

Support for Web Services protocols:

WebSphere Application Server supports a number of Web Services protocols. These protocols provide standard ways of defining Web service applications, allowing the applications to operate independently of the product, platform or programming language used.

Web Services protocols are defined by the Oasis group. Refer to the Oasis Web site, http://www.oasis-open.org, for specifications and further information on each protocol. Use the sub-topics below to find out more information about the support for each protocol in WebSphere Application Server.
* "Web Services Atomic Transaction support in WebSphere Application Server"
* "Web Services Business Activity support in WebSphere Application Server" on page 1103
* "Web Services transactions, firewalls and intermediary nodes" on page 1103

Web Services Atomic Transaction support in WebSphere Application Server:

The Web Services Atomic Transaction support in WebSphere Application Server provides transactional quality of service to the Web services environment. This enables distributed Web service applications, and the resources they use, to take part in distributed global transactions.

The Web Services Atomic Transaction (WS-AT) support is an implementation of the following specifications on WebSphere Application Server. These specifications define a set of Web services that enable Web service applications to participate in global transactions distributed across the heterogeneous Web service environment.
* Web Services Atomic Transaction (WS-AT), at http://www-106.ibm.com/developerworks/webservices/library/ws-atomtran/

    WS-AT is a specific coordination type that defines protocols for atomic transactions.
* Web Service Coordination (WS-COOR), at http://www-106.ibm.com/developerworks/webservices/library/ws-coor/

    WS-COOR specifies a CoordinationContext and a Registration service with which Participant web services may enlist to take part in the protocols offered by specific coordination types.

The WS-AT support is an interoperability protocol that introduces no new programming interfaces for transactional support. Global transaction demarcation is provided by standard J2EE use of the JTA UserTransaction interface. If a Web service request is made by an application component running under a

global transaction, then a WS-AT CoordinationContext is implicitly propagated to the target Web service, but only if the appropriate application deployment descriptors have been set as described in "Configuring transactional deployment attributes" on page 1111.

If WebSphere Application Server is the system hosting the target endpoint for a Web service request that contains a WS-AT CoordinationContext, then WebSphere automatically establishes a subordinate JTA transaction in the target runtime environment that becomes the transactional context under which the target Web service application executes.

The following figure, Figure 20, shows a transaction context shared between two WebSphere application servers for a Web service request that contains a WS-AT CoordinationContext.
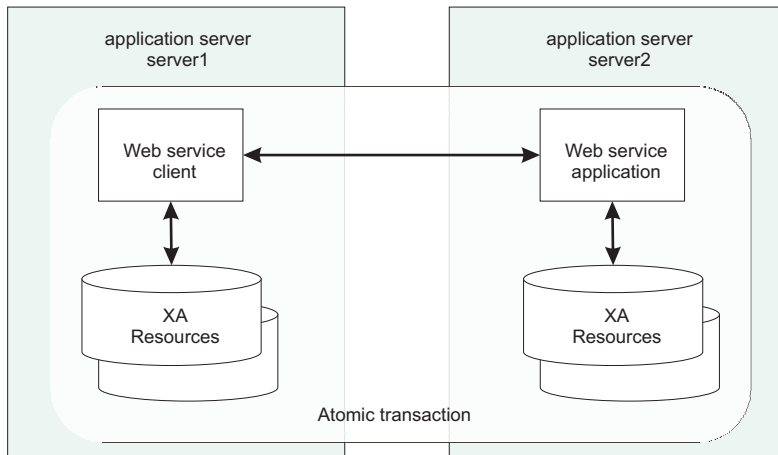
Figure 20. Transaction context shared between two WebSphere application servers.

WS-AT support restrictions

In this version of WebSphere Application Server, WS-AT contexts cannot be started from a non-recoverable client process.

Application design considerations

WS-AT is a two-phase commit transaction protocol and is suitable for short duration transactions only.

Because the purpose of an atomic transaction is to coordinate resource managers that isolate transactional updates by holding transactional locks on resources, it is generally not recommended that WS-AT transactions be distributed across enterprise domains. Inter-enterprise transactions typically require a looser semantic than two-phase commit and, in such scenarios, it can be more appropriate to use a compensating business transaction, for example a Web Services Business Activity or as part of a BPEL process.

WS-AT is most appropriate for distributing transaction context across Web services deployed within a single enterprise. Only request-response message exchange patterns carry transaction context since the originator (application or container) of a transaction needs to be sure that all business tasks executed under that transaction have finished before requesting the completion of a transaction. Web services invoked by a one-way request never run under the transaction of the requesting client.

Application development considerations

There are no specific development tasks required for Web service applications to take advantage of WS-AT. There are some application deployment descriptors that need to be set appropriately, as described in "Configuring transactional deployment attributes" on page 1111.

Application developers do not need to explicitly register WS-AT participants. The WebSphere Application Server runtime takes responsibility for the registration of WS-AT participants, in the same way as the registration of XAResources in the JTA transaction to which the WS-AT transaction is federated. At transaction completion time, all XAResources and WS-AT participants are atomically coordinated by the WebSphere Application Server transaction service.

If a JTA transaction is active on the thread when a Web service Application request is made, the transaction is propagated across on the Web service request and established in the target environment. This is analogous to the distribution of transaction context over IIOP as described in the EJB specification. Any transactional work performed in the target environment becomes part of the same global transaction.

Web Services Business Activity support in WebSphere Application Server:

With Web Services Business Activity (WS-BA) support in WebSphere Application Server, Web services on disparate 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 in the event of an error.

The Web Services Business Activity (WS-BA) support is an implementation of the following specifications in WebSphere Application Server. These specifications define a set of protocols that enable Web service applications to participate in loosely coupled business processes that are distributed across the heterogeneous Web service environment, with the ability to compensate actions if an error occurs. For example, an application that sends an e-mail cannot unsend it following a failure. The application can, however, provide a business-level compensation handler that sends another e-mail advising of the new circumstances. A *business activity* is a group of general tasks that you want to link together so that the tasks have an agreed outcome.

- Web Services Business Activity (WS-BA), at http://www-106.ibm.com/developerworks/library/ specification/ws-tx/#ba

  WS-BA is a specific coordination type that defines protocols for business activities.

- Web Service Coordination (WS-COOR), at http://www-106.ibm.com/developerworks/webservices/library/ ws-coor/

  WS-COOR specifies a CoordinationContext and a registration service with which participant Web services can enlist to take part in the protocols that are offered by specific coordination types.

In addition to supporting the WS-BA interoperability protocol, WebSphere Application Server provides a programming interface for creating business activities and compensation handlers. With this programming interface, you can specify compensation data and check or alter the status of a business activity.

You can also use this compensation ability with applications that are not Web services, as long as these applications involve communication between WebSphere Application Servers only. See the related topics for more information.

Web Services transactions, firewalls and intermediary nodes:

You can configure your system to enable propagation of Web Services Atomic Transactions (WS-AT) message contexts and Web Service Business Activities (WS-BA) message contexts across firewalls or outside the WebSphere Application Server domain. With this configuration you can distribute Web service applications that use WS-AT or WS-BA across disparate systems. The topology that you use can have an effect on the high availability and affinity behavior of the transactions.

The topologies that are available to you are as follows:

Direct connection

No intermediary node exists in this topology. The client communicates directly with a specific WebSphere Application Server. This topology supports high availability for transactions on requests made within a cluster, and affinity of transactions.

Proxy Server for IBM WebSphere Application Server

In this topology, the client communicates with a Proxy Server for IBM WebSphere Application Server, which dynamically routes the client requests to the appropriate server within a WebSphere Application Server cluster.

The proxy server supports high availability, workload management, and affinity of transactions. After you create the proxy server you do not have to perform any extra additional steps to enable these features.
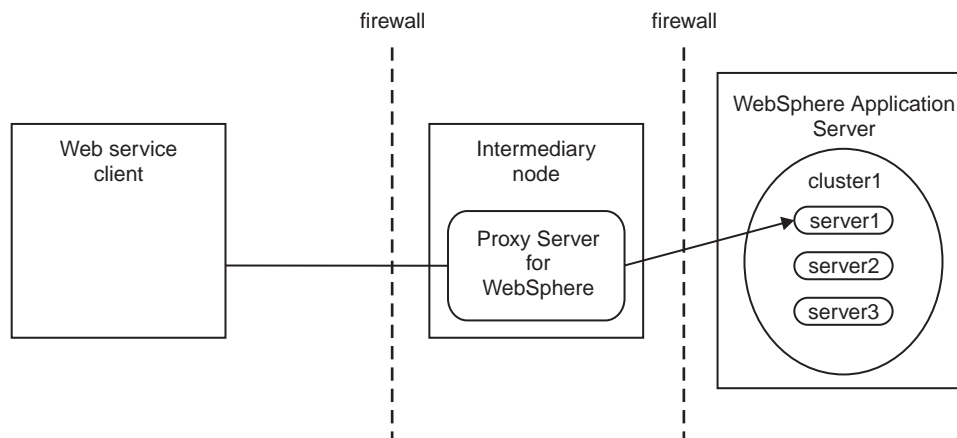
HTTP server, such as IBM HTTP Server

In this topology, the client communicates with an HTTP server, which always routes the client requests to a specific server within a WebSphere Application Server cluster. You configure the HTTP server to specify the WebSphere Application Server to route requests to.

The HTTP server cannot provide either affinity or high availability for transactions. However, transactional integrity is assured because recovery processing occurs after the failed server restarts.

Note: You can still enable high availability on the WebSphere Application Server. Clients accessing this server through an HTTP server cannot have high availability of transactions, however other clients accessing the same server can.

firewall   firewall

```
┌──────────────┐       ┊    ┌──────────────────┐   ┊   ┌────────────────────────┐
│              │       ┊    │   HTTP server    │   ┊   │  WebSphere Application │
│  Web service │       ┊    │                  │   ┊   │         Server         │
│    client    │       ┊    │ ╭──────────────╮ │   ┊   │     ╭───────────────╮  │
│              │message┊    │ │vhost1=server1│ │   ┊   │     │   cluster1    │  │
│              │to vhost2   │ │vhost2=server2│─┼───┊──▶│     │ ╭──────────╮  │  │
│              │───────┊───▶│ │vhost3=server3│ │   ┊   │     │ │ server1  │  │  │
│              │       ┊    │ ╰──────────────╯ │   ┊   │     │ ▶ server2  │  │  │
│              │       ┊    │                  │   ┊   │     │ │ server3  │  │  │
└──────────────┘       ┊    └──────────────────┘   ┊   │     │ ╰──────────╯  │  │
                       ┊                           ┊   │     ╰───────────────╯  │
                       ┊                           ┊   └────────────────────────┘
```
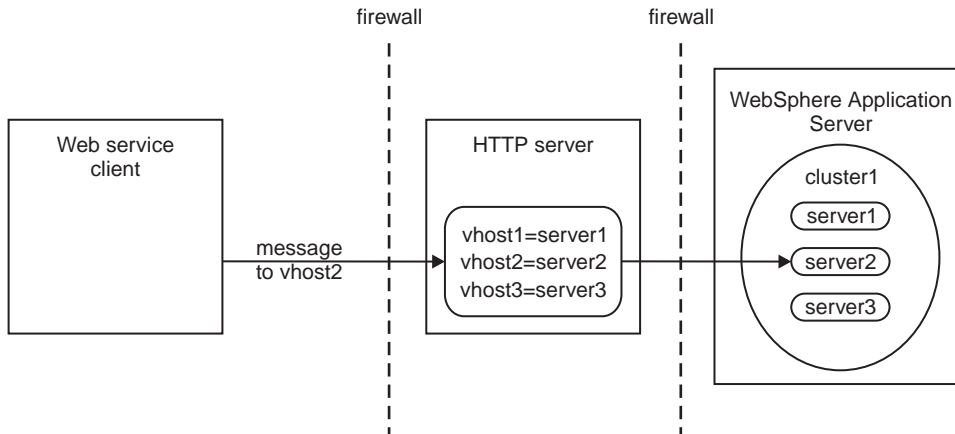
If you have an existing HTTP server, but want the high availability and workload management capabilities of a Proxy Server for WebSphere, create a Proxy Server for IBM WebSphere Application Server and configure the HTTP server to route all requests to the proxy server, as in the following scenario.

HTTP server in conjunction with a Proxy Server for IBM WebSphere Application Server

The client communicates with an HTTP server, which you configure to forward the client's requests to a Proxy Server for IBM WebSphere Application Server. The proxy then dynamically routes the requests to the appropriate server within WebSphere Application Server.

The proxy server provides the client with the high availability and affinity behavior that the HTTP server cannot.

firewall   firewall

```
┌──────────────┐ ┊  ┌──────────────┐ ┊  ┌──────────────┐    ┌────────────────────────┐
│              │ ┊  │              │ ┊  │              │    │  WebSphere Application │
│  Web service │ ┊  │  HTTP Server │ ┊  │ Proxy Server │    │         Server         │
│    client    │ ┊  │              │ ┊  │for WebSphere │    │     ╭───────────────╮  │
│              │─┊─▶│              │─┊─▶│              │    │     │   cluster1    │  │
│              │ ┊  │              │ ┊  │              │─── │     │ ╭──────────╮  │  │
│              │ ┊  │              │ ┊  │              │  ╲ │     │ │ server1  │  │  │
│              │ ┊  │              │ ┊  │              │   ╲│     │ │ server2  │  │  │
│              │ ┊  │              │ ┊  │              │    │  ╲  │ ▶ server3  │  │  │
└──────────────┘ ┊  └──────────────┘ ┊  └──────────────┘    │     │ ╰──────────╯  │  │
                 ┊                   ┊                       │     ╰───────────────╯  │
                 ┊                   ┊                       └────────────────────────┘
```
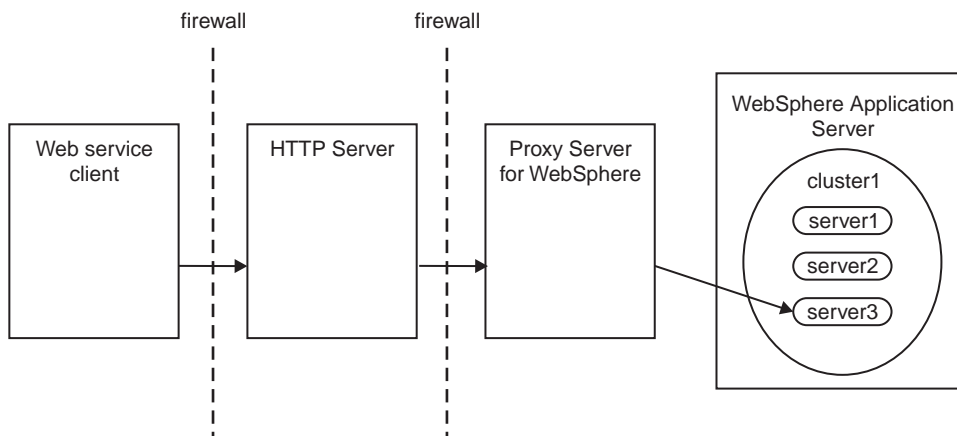
Use of local transactions

Local transaction containment (LTC) support, and its configuration through local transaction extended deployment descriptors, gives IBM WebSphere Application Server application programmers a number of advantages. This topic describes those advantages and how they relate to the settings of the local transaction extended deployment descriptors. This topic also describes points to consider to help you best configure transaction support for some example scenarios that use local transactions.

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 Container Transaction deployment descriptor **Transaction** attribute set to `Not supported` instead of `Required`.

With the extended local transaction support of IBM WebSphere Application Server, applications can perform the same business logic in an unspecific transaction context as they can under a global transaction. An enterprise bean, for example, runs under an unspecified transaction context if it is deployed with a **Transaction** attribute of `Not supported` or `Never`.

The extended local transaction support provides a container-managed, implicit local transaction boundary within which application updates can be committed and their connections cleaned up by the container. Applications can then be designed with a greater degree of independence from deployment concerns. This makes using a **Transaction** attribute of `Supports` much simpler, for example, when the business logic may 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 or not the application runs under a transaction. The application can depend on the close behaving in the same way and not causing 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 under a **Transaction** policy of `Not supported` performs better than if it had been run under a `Required` policy. This benefit is exploited through the **Local Transactions - Resolution-control** extended deployment setting of `ContainerAtBoundary`. With this setting, application interactions with resource providers (such as databases) are managed within implicit RMLTs that are both started and ended by the container. The RMLTs are committed by the container at the configured **Local Transactions - Boundary**; 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.

Use local transactions in a managed environment that guarantees clean-up.

Applications that want to control RMLTs, by starting and ending them explicitly, can use the default **Local Transactions - Resolution-control** extended deployment setting of `Application`. In this case, the container ensures connection cleanup at the boundary of the local transaction context.

J2EE specifications that describe application use of local transactions do so in the manner provided by the default setting of **Local Transactions - Resolution-control**=`Application` and **Local Transactions - Unresolved-action**=`Rollback`. By configuring the **Local Transactions - Unresolved-action** extended deployment setting to `Commit`, then any RMLTs started by the application but not completed when the local transaction containment ends (for example, when the method ends) are committed by the container. This usage applies to both servlets and enterprise beans.

Extend the duration of a local transaction beyond the duration of an EJB component method.

The J2EE specifications restrict the use of RMLTs to single EJB methods. This restriction is because the specifications have no scoping device, beyond a container-imposed method boundary, to which an RMLT can be extended. You can exploit the **Local Transactions - Boundary** extended deployment setting to give the following advantages:

- Significantly extend the use-cases of RMLTs
- Make conversational interactions with one-phase resource managers possible through ActivitySession support.

You can use an ActivitySession to provide a distributed context with a boundary that is longer than a single method. You can extend the use of RMLTs over the longer ActivitySession boundary, which can be controlled by a client. The ActivitySession boundary reduces the need to use distributed transactions where ACID operations on multiple resources are not needed. This benefit is exploited through the **Local Transactions - Boundary** extended deployment setting of `ActivitySession`. Such extended RMLTs can remain under the control of the application or be managed by the container depending on the use of the **Local Transactions - Resolution-control** deployment descriptor setting.

Coordinate multiple one-phase resource managers.

For resource managers that do not support XA transaction coordination, a client can exploit 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 under 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.

To determine how best to configure the transaction support for an application, depending on what you want to do with transactions, consider the following points.

General points

- You want to start and end global transactions explicitly in the application (BMT session beans and servlets only).

  For a session bean, set the **Transaction type** to `Bean` (to use bean-managed transactions) in the component's deployment descriptor. (You do not need to do this for servlets.)

- You want to access only one XA or non-XA resource in a method.

  In the component's deployment descriptor, set **Local Transactions - Resolution-control** to `ContainerAtBoundary`. In the Container transaction deployment descriptor, set **Transaction** to `Supports`.

- You want to access several XA resources atomically across one or more bean methods.

  In the Container transaction deployment descriptor, set **Transaction** to `Required`, `Requires new`, or `Mandatory`.

- You want to access several non-XA resource in a method without having to worry about managing your own local transactions.

  In the component's deployment descriptor, set **Local Transactions - Resolution-control** to `ContainerAtBoundary`. In the Container transaction deployment descriptor, set **Transaction** to `Not supported`.

- You want to access several non-XA resources in a method and want to manage them independently.

  In the component's deployment descriptor, set **Local Transactions - Resolution-control** to `Application` and set **Local Transactions - Unresolved-action** to **Rollback**. In the Container transaction deployment descriptor, set **Transaction** to `Not supported`.

- You want to access one of more non-XA resources across multiple EJB method calls without having to worry about managing your own local transactions.

  In the component's deployment descriptor, set **Local Transactions - Resolution-control** to `ContainerAtBoundary`, **Local Transactions - Boundary** to `ActivitySession`, and **Bean Cache - Activate at** to `ActivitySession`. In the Container transaction deployment descriptor, set **Transaction** to `Not supported` and set **ActivitySession** attribute to `Required`, `Requires new`, or `Mandatory`.

- You want to access several non-XA resources across multiple EJB method calls and want to manage them independently.

  In the component's deployment descriptor, set **Local Transactions - Resolution-control** to `Application`, **Local Transactions - Boundary** to `ActivitySession`, and **Bean Cache - Activate at** to `ActivitySession`. In the Container Transaction deployment descriptor, set **Transaction** to `Not supported` and set **ActivitySession** attribute to `Required`, `Requires new`, or `Mandatory`.

- You want to use a single non-XA resource and one or more XAResources.

  Use the *Last Participant Support*.

The business activity API

Use the business activity API to create business activities and compensation handlers for an application component, and to log data that is required for compensating an activity in the event of a failure.

Overview

The business activity support provides a UserBusinessActivity API, a CompensationHandler interface and two exceptions: RetryCompensationHandlerException and CompensationHandlerFailedException. You can

look up the UserBusinessActivity interface from the application server Java Naming and Directory Interface (JNDI) at `java:comp/websphere/UserBusinessActivity`. For example:

```
InitialContext ctx = new InitialContext();
UserBusinessActivity uba = (UserBusinessActivity) ctx.lookup("java:comp/websphere/UserBusinessActivity");
```

If an application component is running work that might require compensating upon failure, you must provide a compensation handler class that is assembled as part of the deployed application. This new Java class must implement the com.ibm.websphere.wsba.CompensationHandler interface, which supports the close and compensate methods, which take a parameter of a Service Data Object (SDO). During normal application processing, the application can make one or more invocations to the `setCompensationDataImmediate` or `setCompensationDataAtCommit` methods, passing in an SDO representing the current state of the work performed.

When the underlying unit of work (UOW) that the root business activity is associated with completes, all registered compensators are coordinated to complete. During completion, either the compensate or the close method is called on the compensation handler, passing in the most recent compensation data logged by the application component as a parameter. Your compensation handler implementation must be able to understand the data that is stored in the SDO DataObject; using this data, the compensation handler must be able to determine the nature of the work performed by the enterprise bean and compensate or close in an appropriate manner, for example by undoing changes made to database rows in the event of a failure. You associate the compensation handler with an application component by using the assembly tooling, such as Rational Application Developer.

Active and inactive compensation handlers

You implement the CompensationHandler interface for any application component that executes code that might need to be compensated within a business activity scope. CompensationHandler objects are registered implicitly with the business activity scope under which the application runs, whenever the application calls the UserBusinessActivity API to specify compensation data. Compensation handlers can be in one of two states, active or inactive, depending on any transactional UOW under which they are registered. A compensation handler registered within a transactional UOW might be initially inactive until the transaction commits, at which point it becomes active (see below). A compensation handler registered outside a transactional UOW always becomes active immediately.

When a business activity completes, it drives active compensation handlers only. Any inactive compensation handlers that are associated with the business activity are discarded and never driven.

Logging compensation data

The business activity API specifies two methods that allow the application to log compensation data. This data is made available to the compensation handlers during their processing upon completion of the business activity. The application calls one of these methods, depending on whether it expects transactions to be part of the business activity.

`setCompensationDataAtCommit()`

Call this method if the application expects a global transaction on the thread.

- If a global transaction is present on the thread, the CompensationHandler object is initially inactive. If the global transaction fails, it rolls back any transactional work done within its transaction context in an atomic manner, and drives the business activity to compensate other completed UOWs. The compensation handler does not need to be involved. If the global transaction commits successfully, the compensation handler becomes active because it is required to compensate the durable work that is completed by the global transaction, if the overall business activity fails. The setCompensationDataAtCommit method configures the CompensationHandler instance to perform this compensation function.

- If a global transaction is not present when this method is called, the compensation handler becomes active immediately.

For example, using the same business activity instance as in the previous example:

```
DataObject compensationData = doWorkWhichWouldNeedCompensating();
uba.setCompensationDataAtCommit(compensationData);
```

setCompensationDataImmediate()

Call this method if the application does not expect a global transaction on the thread.

The setCompensationDataImmediate method makes a CompensationHandler instance active immediately, regardless of the current UOW context at the time that the method is invoked. The compensation handler is always able to participate during completion of the business activity.

The role of the setCompensationDataImmediate method is to compensate any non-transactional work, in other words, work that can be performed either inside or outside a global transaction, but is not governed by the transaction. For example, sending an e-mail. The compensation handler must be active immediately so if a failure occurs within a business activity, this non-transactional work is always compensated.

Although these two compensation data logging methods, if called within the same enterprise bean, use the same CompensationHandler class, they create two separate instances of the CompensationHandler class at run time. Therefore, the actions of the methods are mutually exclusive; calling one of the methods does not overwrite any work carried out by the other method.

A CompensationHandler instance is not added to a business activity automatically when a business-activity-enabled enterprise bean is invoked. The association is done explicitly through the business activity API, using one of the compensation data logging methods described previously. If such a method is called passing in `null` as a parameter, the CompensationHandler instance is added to the current business activity, and `null` is passed later to either the `compensate` or the close method upon completion of the business activity scope. With this process, you can add the compensation handler to the business activity, and code default behavior within the compensation handler. It is your responsibility to handle the case of `null`s that pass back to your CompensationHandler instance. This scenario happens only if you call a compensation data logging method with `null` as a parameter.

As described previously, the business activity support adds a CompensationHandler instance to the business activity when a compensation data logging method is called for the first time by the enterprise bean using that business activity. At the same time, a snapshot of the J2EE context is taken and logged with the compensation data. Upon completion of the business activity, all the compensation handlers that were added to the business activity are driven to compensate or close. The code that you create within the CompensationHandler class is guaranteed to run wiithin the same J2EE context that was captured in the earlier snapshot.

For details about the methods available in the business activity API, see the WebSphere Application Server application programming interface reference information.

Transaction service exceptions

This topic lists the exceptions that can be thrown by the WebSphere Application Server transaction service. The exceptions are listed in the following groups:
- 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) 1.0.1 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) 1.0.1 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.

UserTransaction interface - methods available

For details about the methods available with the UserTransaction interface, see the WebSphere Application Server application programming interface reference information (Javadoc) or the Java Transaction API (JTA) 1.0.1 Specification.

Developing components to use transactions

These topics provide information about developing WebSphere application components to use transactions

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.

You configure whether EJB components use container- or bean-managed transactions by setting an appropriate value on the Transaction type deployment attribute, as described in Configuring transactional deployment attributes. You can also configure other transactional deployment descriptor attributes.

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 as described in Using component-managed transactions. There are some limitations to the transaction support available to application client components, as described in Client support for transactions.

Configuring transactional deployment attributes

Use this task to configure the transactional deployment descriptor attributes associated with an EJB or Web module, to enable a J2EE application to use transactions.

You can configure the deployment attributes of an application by using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This topic describes the use of the Application Server Toolkit (AST) to configure the deployment attributes of an application. 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 Chapter 7, "Assembling applications," on page 1345.

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 information about starting the AST, refer to the AST infocenter.
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 J2EE Hierarchy view, right-click the component instance, then click **Open With > Deployment Descriptor Editor**. For example:
   - 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 component's deployment descriptor is displayed in the property pane.
4. **[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 Container
   - To use bean-managed transactions, set Bean
5. In the deployment descriptor notebook, select the Bean tab. Under **WebSphere Extensions**, optionally configure **Local Transaction**. 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 **Bean method** 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 they 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 will instruct the RMLTs to commit only in the absence of an un-handled exception. If the application method executing under 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.)

6. Continuing in WebSphere Extensions, configure **Global Transaction**. 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 started by the container on behalf of the enterprise bean. Any value specified overrides, for transactions started on behalf of the component, the default transaction timeout configured on the application server.

Use Web Services Atomic Transaction

[For enterprise beans only.] Specifies whether the application component, if it makes any Web service requests, expects any transaction context to be propagated with the Web service requests in accordance with the WebSphere WS-AtomicTransaction support described in "Web Services Atomic Transaction support in WebSphere Application Server" on page 1101. Unless specified using this attribute, Web service requests do not carry transaction context.

Send Web Services Atomic Transaction on requests

[For web components only.] Specifies whether the application component, if it makes any Web service requests, expects any transaction context to be propagated with the Web service requests in accordance with the WebSphere WS-AtomicTransaction support described in "Web Services Atomic Transaction support in WebSphere Application Server" on page 1101. Unless specified using this attribute, Web service requests do not carry transaction context.

Execute using Web Services Atomic Transaction on incoming requests

[For web components only.] Specifies whether web application components are prepared to run under a received WS-AtomicTransaction context. Unless specified using this attribute, the web application component's container suspends any received transaction context in a similar manner to the EJB container's behavior for an enterprise bean deployed with a **Container transaction type** of `NotSupported`. Setting this attribute enables a web application component

to run under a received WS-AtomicTransaction context in a similar fashion to an enterprise bean deployed with a **Container transaction type** of `Supports`.

7. [For EJB components only] For container-managed transactions, configure how the container must manage the transaction boundaries when delegating a method invocation to an enterprise bean's business method:

   a. In the deployment descriptor notebook, select the Assembly tab. The **Container Transactions** box 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 execute under 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** set in this task, and the **ActivitySession kind** attribute as described in "Setting EJB module ActivitySession deployment attributes" on page 1136. For more detail about the relationship between these two properties, see "Combining transaction and ActivitySession container policies" on page 1125.

8. [For Web service applications that use a SOAP/JMS binding and participates in WS-AtomicTransactions] Ensure that the **Container transaction type** of the message-driven bean named `JMS router MDB` is set, as described in the previous step, to a value of `NotSupported`. Web service applications that use a SOAP/JMS binding include in the assembled EAR a router message-driven bean named `JMS router MDB`. If a Web service uses a SOAP/JMS binding and participates in WS-AtomicTransactions, as described in "Web Services Atomic Transaction support in WebSphere Application Server" on page 1101, then the **Container transaction type** of the `JMS router MDB` must be set, as described in the previous step, to a value of `NotSupported`. Note that there is no equivalent action neccessary for Web service applications that use a SOAP/HTTP binding and participate in WS-AtomicTransactions.

9. [For client application components only] Enable, if required, support for transaction demarcation by the client. In the deployment descriptor notebook, select the option **Allow JTA demarcation**. This directs the client container to bind the JTA UserTransaction interface into JNDI at java:comp/UserTransaction for the client component. There are constraints on the capabilities of the transaction support in the client container described in "Client support for transactions" on page 1087.

10. 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.

11. Verify the archive files. For more information about verifying files using the AST, refer to the AST infocenter.

12. From the popup menu of the project, click **Deploy** to generate EJB deployment code.

13. **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 from the Application Server Toolkit (AST) or Rational Web Developer to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important

    Important: Use **Run On Server** for unit testing only. The Application Server Toolkit (AST) or Rational Web Developer 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 on production servers.

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 Chapter 9, "Deploying and administering applications," on page 1363.

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.

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 ();
        ...
    }
    ...
}
```

Creating an application that exploits the business activity support

To create an application component that uses the business activity support, you must set **Run EJB methods under a Business Activity scope** in the deployment descriptor of the relevant application component, and if required, create and specify a compensation handler for the application to use in the

event of an error. You then build the component into the application and deploy the application onto a server that has the business activity support enabled. The application component can be either an enterprise bean or a Web service that is implemented as an enterprise bean.

For information about editing deployment descriptors, refer topics such as ″Application Deployment Descriptor editor″, in the Application Server Toolkit documentation, accessible from the Contents view.

Perform this task to for an application that runs on a business-activity-enabled sever to exploit the business activity support at run time, and to perform work that might later be compensated by a compensation handler. If the application requires compensation when a business activity scope ends, the application passes data that is required by the compensation process, in the form of a Service Data Object (SDO), to a compensation handler indirectly by using the business activity API.

1. Design the application component that requires the business activity support. In particular, define the application component requirements for compensation and close activities. If the application component requires compensation, define the nature of the data in the SDO that the application component passes to the compensation handler.

2. Using the information from your application design, create the compensation handler for the application component, if required. This handler defines the close and compensation logic that runs upon completion of a business activity scope that has the handler added to it through an application component.

   a. Open your chosen WebSphere Application Server assembly tool.

   b. Create a new Java class that implements the com.ibm.websphere.wsba.CompensationHandler interface.

   c. Implement the close and compensate methods on the new compensation handler object, to perform appropriate actions depending on the SDO data that passes to the handler when it is invoked.

   The compensation handler class is now ready for the application component to reference, and for assembly into an application.

3. Open the application component in the assembly tool.

4. Open the deployment descriptor for the application component in the deployment descriptor viewer.

5. Scroll to the **Compensation** section and select the **Run EJB methods under a Business Activity scope** check box.

6. In the **Compensation handler class** text field, type the fully qualified class name of the compensation handler class that you created earlier.

7. Save the deployment descriptor.

8. Build the application, including both the application component and the compensation handler. If the application is a Web service, the application must be compliant with the Java Specification Request (JSR) 109 standard.

9. Deploy the application onto an application server that is business-activity-enabled.

The application is now business-activity-enabled, and can exploit the business activity support at run time through the business activity API. The application component has a compensation handler associated with it, and can therefore call the setCompensationDataImmediate and setCompensationDataAtCommit methods at run time to add the compensation handler to the business activity scope. For more information about these methods, see "The business activity API" on page 1107. If the unit of work with which the business activity scope is associated fails, the compensation handler performs actions to compensate for the error.

Ensure that the compensation handler class is on the application class path for the WebSphere Application Server runtime environment.

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.

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:
- "Coordinating access to one-phase commit and two-phase commit capable resources within the same transaction"
- "Assembling an application to use one-phase and two-phase commit resources in the same transaction" on page 1117
- "Configuring an application server to log heuristic reporting" on page 1119

Coordinating access to one-phase commit and two-phase commit capable resources within 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 coordinate access to one-phase and two-phase commit capable resources within the same transaction.

Note: If the global transaction is distributed across multiple application servers *that are all running at WebSphere Application Server version 5.1 or later* then you can exploit last participant support to coordinate a one-phase commit capable resource and any number of two-phase commit capable resources within 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 within 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 within the same transaction, you configure the application to 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 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 within the same transaction.

To enable an application to use one-phase and two-phase commit capable resources within the same transaction, you must configure the deployment attributes of the application to accept the increased risk of an heuristic outcome.

You can configure the deployment attributes of an application by using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

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 Chapter 7, "Assembling applications," on page 1345.

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 about starting the AST, refer to the AST infocenter.
2. Create or edit the application EAR file.

   Note: Ensure that you set the target server as WebSphere Application Server version 6.1.

   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 v6.1`
   d. Click **Finish**
3. In the J2EE Hierarchy view, 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. In the property pane, select the Extended Services tab.

5. In the Last Participant Support section, select the **Last participant support** checkbox.

6. 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.

7. Verify the archive files. For more information about verifying files using the AST, refer to the AST infocenter.

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 that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Application Server Toolkit (AST) or Rational Web Developer to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

   Important

   Important: Use **Run On Server** for unit testing only. The Application Server Toolkit (AST) or Rational Web Developer 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 on production servers.

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 Chapter 9, "Deploying and administering applications," on page 1363.

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

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.

Exceptions thrown for transactions involving 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" on page 26.

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
- http://www.jcp.org/jsr/detail/95.jsp
- http://java.sun.com/products/jta/

Other
- http://www-306.ibm.com/software/webservers/appserv/enterprise/
- http://www-306.ibm.com/software/webservers/appserv/was/
- http://publib-b.boulder.ibm.com/Redbooks.nsf/Portals/WebSphere
- http://www-4.ibm.com/software/webservers/appserv/whitepapers.html

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 "Learn about WebSphere applications: Overview and new features" on page 1 for a brief description of each WebSphere extension.

In addition, now 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 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 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. For more information about this feature, see "Application extension registry" on page 97.

ActivitySessions

Using the ActivitySession service
These topics provide information about implementing WebSphere enterprise applications that use ActivitySessions.

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 J2EE 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 J2EE 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.

For more information about implementing WebSphere enterprise applications that use ActivitySessions, see the following topics:
- "The ActivitySession service" on page 1121
  - "ActivitySession and transaction contexts" on page 1124
  - "Using ActivitySessions with HTTP sessions" on page 1122
- "The ActivitySession service application programming interfaces" on page 1131
- "Developing a J2EE application to use ActivitySessions" on page 1134

- "Samples: ActivitySessions" on page 1132
- "Setting EJB module ActivitySession deployment attributes" on page 1136
- "Setting Web module ActivitySession deployment attributes" on page 1138
- Disabling or enabling the ActivitySession service
- Configuring the default ActivitySession timeout for an application server
- 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:
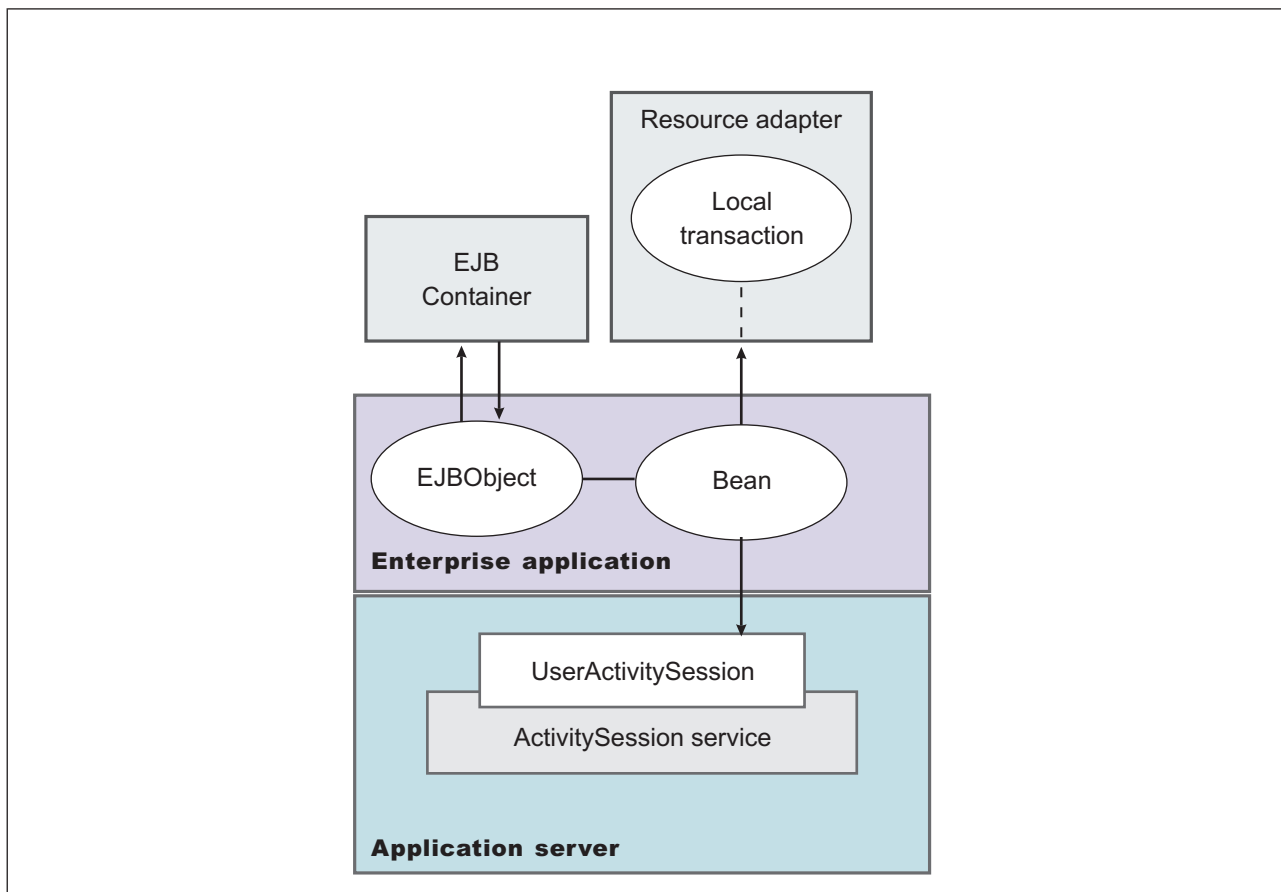
Figure 21. The ActivitySession service. This figure shows the main components of the ActivitySession service within WebSphere Application server. For an overview of these components, see the text that accompanies this figure.

Although the purpose of a global transaction is to coordinate multiple resource managers, global transaction context is often used by J2EE 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 J2EE-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 J2EE 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 a J2EE 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.

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 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:

Web application
invokes servlet

HttpSession

ActivitySession

As1

Servlet starts
HttpSession
(ActivitySession started automatically)

Servlet invalidates
HttpSession
(ActivitySession checkpointed automatically)
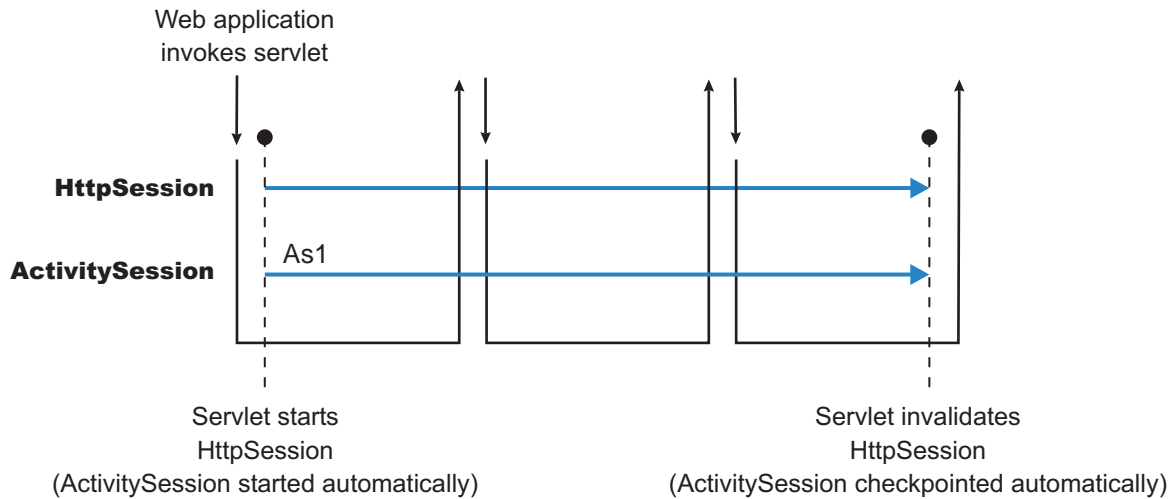
- The Web application starts and ends ActivitySessions.

  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:

Web application
invokes servlet

HttpSession

ActivitySession

As1   As2

Servlet
starts
HttpSession

Servlet starts and
ends ActivitySessions
specifically

Servlet invalidates
HttpSession
(ActivitySession As2
checkpointed automatically)
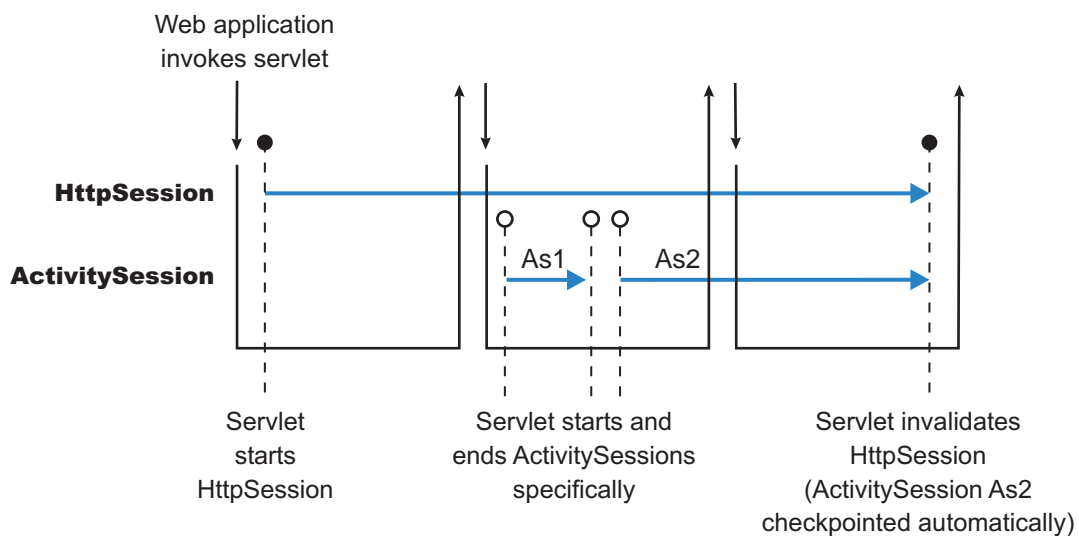
A Web application can invoke servlets configured for either usage model.

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 EJBs 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[5]. 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

---

5. 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

described in the J2EE specifications. The J2EE 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 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-executing global transactions.

Combining transaction and ActivitySession container policies:

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 40. 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 40. 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 40. 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 40. 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 40. 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 40. 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

The ActivitySession service application programming interfaces:

The ActivitySession service consists of an application programming interface available to Web applications, session EJBs, and J2EE 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 J2EE client applications. This UserActivitySession interface defines the set

of ActivitySession operations available to an application component. An implementation of this interface is obtained via a JNDI lookup of the URL ″java:comp/websphere/UserActivitySession″. It is used to begin and end ActivitySessions and to query various attributes of the active ActivitySession associated with the thread.

For more information about the ActivitySession API, see WebSphere Application Server application programming interface reference information (Javadoc).

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.

J2EE client container application 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.

A J2EE client container application and two session beans with different ActivitySession types
This sample consists of a J2EE client container application 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 (Javadoc)

Programming specifications
- J2EE Activity Service for Extended Transactions
- Java Transaction API (JTA) 1.0.1

Other
- WebSphere Business Integration Server Foundation
- Listing of all IBM WebSphere Application Server Redbooks
- Listing of all IBM WebSphere Application Server Whitepapers

Developing a J2EE application to use ActivitySessions

This topic provides an overview of the scenarios for which you would develop a J2EE application to use an ActivitySession.

The following common J2EE application scenarios make use of an ActivitySession:
- Developing a J2EE application to use one or more enterprise beans that are persisted to non-transactional datastores.

  This scenario can be used by an application that needs to coordinate multiple one-phase resource managers; for example, for two or more entity EJBs 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.
- Developing a J2EE application in which an enterprise bean accesses a resource manager multiple times in different business methods.

  This scenario can be used by 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.
- Developing a J2EE client application to use an ActivitySession to scope EJB activation and load-balancing.

  This scenario can be used by 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 J2EE 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 J2EE 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.
- Developing 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.

General considerations:
- An application that is accessed under an ActivitySession context can receive a javax.transaction.InvalidTransactionException RemoteException, thrown by the 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 J2EE 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.

For examples of using ActivitySessions in J2EE applications, see ActivitySessions samples.

Developing an enterprise bean or J2EE client to manage ActivitySessions

Use this task to write the code needed by a session EJB or J2EE client application to manage an ActivitySession, based on the example code extract provided.

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 J2EE client participate directly in ActivitySessions. You then need to write the code needed by the session bean or J2EE client application 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 J2EE client application to manage ActivitySessions, consider the points described in ActivitySessions and transaction contexts.

To write the code needed by a session EJB or J2EE client application 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 J2EE 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.

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.

You can configure the deployment attributes of an application by using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This topic describes the use of the Application Server Toolkit (AST) 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). This task description assumes that you have an 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. For more detail about the fields in the assembly tool, and for associated task help, see the help information provided with the toolkit.

To set the ActivitySession deployment attributes for an enterprise bean, complete the following steps:

1. Start the assembly tool. For more information about starting the AST, refer to the AST infocenter.
2. Create or edit the application EAR file.

   Note: Ensure that you set the target server as WebSphere Application Server version 6.1.

   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 v6.1`
   d. Click **Finish**

3. In the J2EE Hierarchy view of the J2EE 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 1111. For more detail about the relationship between these two properties, see "Combining transaction and ActivitySession container policies" on page 1125.

9. 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.

10. Verify the archive files. For more information about verifying files using the AST, refer to the AST infocenter.

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 that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Application Server Toolkit (AST) or Rational Web Developer to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

Important

Important: Use **Run On Server** for unit testing only. The Application Server Toolkit (AST) or Rational Web Developer 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 on production servers.

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 Chapter 9, "Deploying and administering applications," on page 1363.

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.

You can configure the deployment attributes of an application by using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This topic describes the use of the Application Server Toolkit (AST) to configure the deployment attributes. This task description assumes that you have an 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.

To set the ActivitySession deployment attributes for a Web application, complete the following steps:

1. Start the assembly tool. For more information about starting the AST, refer to the AST infocenter.

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-> WAR file**

b. Click **Next**, then select the WAR file.

c. Click **Finish**

3. In the J2EE Hierarchy view, right-click the Web module, then click **Open With > Deployment Descriptor Editor**. 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 either 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, then the container associates the ActivitySession with the HttpSession.
- If an ActivitySession is started in the absence of an HttpSession, then 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, 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.
- 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 indicate that you want to save changes to the deployment descriptor.

9. Verify the archive files. For more information about verifying files using the AST, refer to the AST infocenter.

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 that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Application Server Toolkit (AST) or Rational Web Developer to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

Important

> **Important:** Use **Run On Server** for unit testing only. The Application Server Toolkit (AST) or Rational Web Developer 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 on production servers.

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.

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 WebSphere Application Server runtime environment. The run time can tailor its support to the exact requirements of that unit of work.

Access intent is currently 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 WebSphere Application Server, 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 J2EE 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. The following are example situations where application profiling is useful
- **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 1141.

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.″

best-practices: 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 WebSphere Application Server 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 WebSphere Application Server 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.

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 Creating a custom access intent policy.

1. Configuring tasks. Declaratively configure tasks as described in the following topics:
   - 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. Creating an application profile.

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 196 describes how to configure a transaction on an EJB module. "Configuring transactional deployment attributes" on page 1111 describes how to define other transaction attributes. "Using the ActivitySession service" on page 1120 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 1141.

Automatic configuration of application profiling:

The Application Server Toolkit (AST) 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 2 Platform, Enterprise Edition (J2EE) 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.

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 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.

Automatically configuring application profiles and tasks:

You can automatically configure application profiling for an application through static analysis.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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 Version 6.0 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. Be sure that the application and its modules successfully compile. To include JavaServer Pages (JSP) files in the analysis, you must precompile the pages. Also, be sure that you have configured all transactional attributes before analyzing.

6. In the Project Explorer view of the J2EE perspective, right-click the **Deployment Descriptor: Enterprise Application Name** under the Enterprise Application, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the Enterprise Application project is displayed in the property pane.

7. In the property pane, select the **Extended Services** tab.

8. Beneath the *Application Profiling* table, select **Auto...**.

9. Select the projects to be analyzed and configured. Select **Next**.

10. To limit the returned results of the analysis, choose *closed world analysis*. Closed world analysis generates application profiles only if a client entry point in a message driven bean (MDB), servlet, JSP, or application client is resolved that begins a transaction and enlists entities. If closed world is not selected, the analysis returns the set of application profiles for all possible transactions represented by the application.

    Note: At this point you can also choose the **Clean** attribute. If you set this attribute, the existing configuration of selected modules is removed and the new configuration is applied. If you do not select this option, the new configuration is merged into the existing configuration.

11. Choose the concurrency for the default configuration of the access intent for generated application profiles.

12. Select **Analyze > Next**.

13. Examine the results of the analysis. Each top-level entry in the table represents a transaction identified by the analysis. The nested entries represent the callers of the transaction, the entities enlisted by the transaction, and the attributes read or modified during the course of the transaction.

14. Select **Finish** to automatically configure the container-managed tasks and application profiles represented by the analysis.

Applying profile-scoped access intent policies to entity beans:

You can configure entities with access intent for an application profile.

1. Start the Application Server Toolkit.

2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.

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 application profile 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 **Extended Access** tab.

7. Select the **application profile** for which you want to specify the access intent.

8. Beneath the **Access Intent for Entities 2.x (Profile Level)** panel, select **Add...**

9. Select the **entities** to configure and click **Next...**.

10. Select the **access intent policy** to apply. Select **Read Ahead Hint** if a read ahead hint is desired.

11. Select **Next**.

12. **Optional:** Specify the **collection scope**

    Transaction

    > This is the default. Collections of entities cannot be used beyond the scope of the transaction in which you create the collection.

    ActivitySession

    > Collections of entities cannot be used beyond the scope of the *ActivitySession* in which you create the collection. The collection can be used in a new transaction if that transaction is nested under the original *ActivitySession*, although you might have to reload the object by querying the underlying data store.

13. **Optional:** Specify the **collection increment**. Specify a valid integer to define the chunks that populate a remote collection. This value only applies to *remote* collections and is ignored by local collections. The default for access types that result in U locks is 1. Otherwise, the default is 25.

14. **Optional:**

15. Specify the **resource manager prefetch increment**. Specify a valid integer to set as the fetch size on the JDBC statement when you execute queries for a bean type. The default is 0.

16. Select **Next**.

17. If you selected *read ahead*, choose the **preload path**.

18. Select **Finish** to apply.

19. Select **OK**.

Creating a custom access intent policy:

You can define a custom access intent policy, which can be configured for 2.x entity beans.

1. Start the Application Server Toolkit.

2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.

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 Version 6.0 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. In the property pane, select the **Extended Access** tab.

7. Beneath the **Defined Access Intent Policies** panel, select **Add**.

8. Specify a **unique name** by which the policy is referenced when applied to entity beans.

9. **Optional:** Specify a **description** of the policy.

10. Specify an **access type**.

11. Specify the **collection scope**.

Transaction

> This is the default. Collections of entities cannot be used beyond the scope of the transaction in which the collection is created.

ActivitySession

> Collections of entities cannot be used beyond the scope of the *ActivitySession* in which the collection is created. The collection can be used in a new transaction if that transaction is nested under the original *ActivitySession*, although you might need to reload the object by querying the underlying data store.

12. Specify the **collection increment**. Specify a valid integer to define the chunks that populate a remote collection. This value only applies to *remote* collections and is ignored by local collections. The default value for access types that result in U locks is 1. Otherwise, the default is 25.

13. Specify the **resource manager prefetch increment**. Specify a valid integer to set as the fetch size on the JDBC statement when executing queries for a bean type. The default value is 0.

Applying access intent policies to entity beans.

Creating 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.

1. Start the Application Server Toolkit.

2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.

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. In the property pane, select the **Extended Access** tab.

7. Beneath the **Application Profiles** table, select **Add...**

8. Select or type the **name** of the task for which this profile applies. You cannot map the task to any other profile within the module. The task name should have been configured already as a task on a web component (container managed task, application managed task), an application client (container managed task, application managed task), or the method of an EJB (container managed task, application managed task).

9. **Optional:** Specify a **description** of the task.

10. Select **Next**.

11. Select the entities that are enlisted in the unit of work represented by the application profile. You can add additional entities as a separate task after the profile has been created.

12. Select **Finish**.

Configuring access intent for application profiles.

Configuring 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.

If a unit of work is not begun, then the configured task is ignored. For more information about using tasks, see "Application profiling tasks" on page 1145 and "Tasks and units of work considerations" on page 1141.

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.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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: Application Client Module Name** under the application client module, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the Application Client project is displayed in the property pane.
6. Select the **Extended Services** tab.
7. Enter the **name** and **description** of the task in the **Container-Managed Task** section. The task name is mapped to application profiles and used by the run time to determine the appropriate access intent to use for enlisted entities. Task names do not have to be unique within an application. However, task names should be shared consciously and conservatively. At run time, all tasks with the same name are treated the same way, regardless of where you configured the task.

   The description is provided for your convenience; it is not used by the runtime environment.
8. Select **OK**.

Configuring container-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 JavaServer Pages (JSP) file with an application profile.

If a unit of work is not begun, then the configured task is ignored. For more information about using tasks, see "Application profiling tasks" on page 1145 and "Tasks and units of work considerations" on page 1141.

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.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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: Web Module Name** under the web module, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the web project is displayed in the property pane.
6. Select the **Servlets** tab.
7. Select the servlet or JSP that you want to change.
8. Expand the WebSphere Programming Model Extensions section.
9. Enter the **name** and **description** of the task in the **Container-Managed Task** section. The task name is mapped to application profiles and used by the run time to determine the appropriate access intent to use for enlisted entities. Task names do not have to be unique within an application. However, task names should be shared consciously and conservatively. At run time, all tasks with the same name are treated the same way, regardless of where you configured the task.

   The description is provided for your convenience; it is not used by the runtime environment.
10. Select **OK**.

Configuring 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 Enterprise JavaBean (EJB) developer, you can configure an enterprise bean's container-managed tasks to associate requests from the bean with application profiles.

Units of work begun during the execution of a method configured with a task are associated with the task name. If the method is executed under an imported transaction, then the configured task is ignored. For more information about using tasks, see "Application profiling tasks" on page 1145 and "Tasks and units of work considerations" on page 1141.

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.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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 **Extended Access** tab.
7. Beneath the **Container-Managed Tasks** table, select **Add...**.
8. Select the **bean** for which you want to configure the task.
9. Select **Next**.
10. Select the **method** for which you want to configure the task. . This method must begin a new unit of work in order 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. If the container begins a new unit of work when the method executes, then it is associated with the configured task name. If the method's implementation programmatically begins a new unit of work, then that unit of work is associated with the configured task name.
11. Select **Next**.
12. Enter the **name** and **description** of the task. The task name is mapped to application profiles and used by the run time to determine the appropriate access intent to use for enlisted entities. Task names do not have to be unique within an application. However, task names should be shared consciously and conservatively. At run time, all tasks with the same name are treated the same way, regardless of where you configured the task.

    The description is provided for your convenience; it is not used by the runtime environment.
13. Select **OK**.

Configuring application-managed tasks for application clients:

For application clients that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession, you must configure application-managed tasks so that an application client can associate requests from the client with application profiles.

If a unit of work is not begun after the task is set, then the set task is ignored. For more information about using tasks, see "Application profiling tasks" on page 1145 and "Tasks and units of work considerations" on page 1141.

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.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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: Application Client Module Name** under the application client module, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the Application Client project is displayed in the property pane.
6. Select the **Extended Services** tab.
7. Beneath the **Application-Managed Tasks** panel, select **Add...**.
8. Specify the name of the **task reference**. The task reference name is used programmatically by the application client. The task reference is the logical representation of the task that is used by the run time environment.
9. Enter the **name** and **description** of the task. The task name is mapped to application profiles and used by the run time to determine the appropriate access intent to use for enlisted entities. Task names do not have to be unique within an application. However, task names should be shared consciously and conservatively. At run time, all tasks with the same name are treated the same way, regardless of where you configured the task.

   The description is provided for your convenience, it is not used by the run time environment.
10. Select **OK**.

Configuring application-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.

If a unit of work is not begun after the task is set, then the set task is ignored. For more information about using tasks, see "Application profiling tasks" on page 1145 and "Tasks and units of work considerations" on page 1141.

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.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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: Web Module Name** under the web module, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the web project is displayed in the property pane.
6. Select the **Servlets** tab.
7. Select the servlet or JSP that you want to change.
8. Expand the WebSphere Programming Model Extensions section.
9. Beneath the **Application-Managed Tasks** table, select **Add...**.
10. Specify the name of the **task reference**. The task reference name is used programmatically by the servlet or JSP. The task reference is the logical representation of the task that is used by the run time environment.
11. Enter the **name** and **description** of the task. The task name is mapped to application profiles and used by the run time to determine the appropriate access intent to use for enlisted entities. Task names do not have to be unique within an application. However, task names should be shared consciously and conservatively. At run time, all tasks with the same name are treated the same way, regardless of where you configured the task.

    The description is provided for your convenience; it is not used by the runtime environment.

12. Select **OK**.

Configuring application-managed tasks for Enterprise JavaBeans:

For Enterprise JavaBeans (EJB) 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.

If a unit of work is not begun after the task is set, then the task is ignored. The task must not be performed for an enterprise bean that uses container-managed transactions or container-managed ActivitySessions. For more information about using tasks, see "Application profiling tasks" on page 1145 and "Tasks and units of work considerations" on page 1141.

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.

1. Start the Application Server Toolkit.
2. **Optional:** Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
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 **Extended Access** tab.
7. Beneath the **Application-Managed Tasks** table, select **Add...**.
8. Select the bean for which you want to configure the task.
9. Type the **name** of the task reference. The task reference name is used programmatically by the bean. The task reference is the logical representation of the task used by the runtime environment.
10. Enter the **name** and **description** of the task. The task name is mapped to application profiles and used by the run time to determine the appropriate access intent to use for enlisted entities. Task names do not have to be unique within an application. However, task names should be shared consciously and conservatively. At run time, all tasks with the same name are treated the same way, regardless of where you configured the task.

    The description is provided for your convenience; it is not used by the runtime environment.

11. Select **OK**.

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.

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 2 Platform Enterprise Edition (J2EE) application, using the J2EE context of the asynchronous bean creator.

Asynchronous beans can improve performance by enabling a J2EE program to decompose operations into parallel tasks. Asynchronous beans support the construction of stateful, active J2EE applications. These applications address a segment of the application space that J2EE 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 J2EE security context of the creator J2EE component. These beans also can run with copies of other J2EE contexts, such as:
* Internationalization context
* Application profiles, which are not supported for J2EE 1.4 applications and deprecated for J2EE 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 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 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 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 J2EE 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 J2EE applications. The administrator specifies the properties of the thread pool and a policy that determines which J2EE 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 J2EE 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 J2EE 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 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 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 J2EE 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 J2EE component metadata

If an asynchronous bean is a J2EE 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 J2EE 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 J2EE 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 J2EE 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 J2EE service context information. The `WorkWithExecutionContext createWorkWithExecutionContext(Work r)` method on the WorkManager interface will create a snapshot of the J2EE 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 J2EE 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 .

Work managers:

A work manager is a thread pool created for J2EE 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 J2EE 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 J2EE 1.4 applications. For more information, see the Programming model section in this article.

When writing a Web or 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 J2EE 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 J2EE 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 J2EE 1.4 environment, the interface does not return a new instance for each JNDI naming lookup, since this specification is not included in the J2EE 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 J2EE contexts

Asynchronous beans can inherit the following J2EE 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 J2EE 1.4 applications. For J2EE 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 J2EE 1.3 applications. For J2EE 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 J2EE 1.4 applications. The scheduled work that runs in a J2EE 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 J2EE 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 J2EE 1.4 environment.
- **Asynchronous beans and CommonJ specification extensions**. The current asynchronous beans Event Source, asynchronous scopes, subsystem monitors and J2EEContext 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 41. Look up work manager

Asynchronous beans	CommonJ

Table 41. Look up work manager (continued)

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 42. Create your work using MyWork

Asynchronous beans	CommonJ

``
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 43. 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 44. Create a timer 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");

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 45. 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 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 46. Set up the timer listener

Asynchronous beans	CommonJ
<pre> public class ABAlarmListener implements
AlarmListener {
     public void fired(Alarm alarm) {
        System.out.println("Alarm fired.
Context =" + alarm.getContext());
     }</pre> | <pre> 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;
    }
 }</pre> |

Example: Using connections with asynchronous beans:

An asynchronous bean method can use the connections that its creating Java 2 Platform Enterprise Edition (J2EE) 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 J2EE 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.

Configure at least one work manager or timer manager using the administrative console.

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.

Your administrator needs to configure at least one work manager using the administrative console.

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 the Application Server Toolkit (AST) or 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 the Application Server Toolkit (AST) or 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.

Your administrator needs to configure at least one timer manager using the administrative console.

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 Application Server Toolkit (AST) or Rational Web 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, such as the Application Server Toolkit (AST) or Rational Web Developer.
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 or Rational Application Developer then 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.

Your administrator needs to configure at least one work manager using the administrative console.

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 the Application Server Toolkit (AST) or Rational Web Developer.

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, such as the Application Server Toolkit (AST) or Rational Web Developer 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.

Your administrator must have configured at least one work manager using the administrative console.

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 J2EE 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 Javadoc.

A component wanting to run work in parallel, or in a different J2EE context, locates a work manager in 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 2 Developer Kit threads because they are not managed threads. Plus, these threads are not affiliated with the J2EE environment, which makes them useless inside an application server. In addition, these threads have no J2EE 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: Work object:

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.

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.

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: Event listener:

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.

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 Javadoc.

   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 Javadoc.

   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.

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 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 2 Enterprise Edition (J2EE) 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 Javadoc. 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.

best-practices: The Java 2 Software Development Kit (SDK) already has a timer mechanism, so why create a new one? The Java 2 SDK is a Java 2 Platform Standard Edition (J2SE) feature that knows nothing about the J2EE environment. Timers fired by the J2SE feature do not run on a managed thread and are therefore unusable inside an application server. These timers also lack a J2EE 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.

Advanced Java 2 Platform Enterprise Edition (J2EE) applications often rely on remote, non-managed, non-J2EE 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 Javadoc.

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 2 Platform Enterprise Edition (J2EE) 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
Use the dynamic cache service to improve application performance by caching the output of servlets, commands, and JavaServer Pages (JSP) files.

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 Enabling 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 1182.
    *   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.

To use the DistributedMap and DistributedObjectCache interfaces for the dynamic cache, see "Using the DistributedMap and DistributedObjectCache interfaces for the dynamic cache" on page 1196.

Dynamic cache:

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.

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.

Example: Configuring the dynamic cache:

This example puts all the steps together for configuring the dynamic cache with the `cachespec.xml` file, showing the use of the cache ID generation rules, dependency IDs, and invalidation rules.

Suppose that a servlet is used to manage a simple news site. This servlet uses the query parameter ″action″ to determine if the request is being used to ″view″ news or ″update″ news (used by the administrator). Another query parameter ″category″ is used to select 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 dynamic cache for this example with the `cachespec.xml` file:
1. Define the <cache-entry> elements necessary to identify the servlet. In this case, the servlet's URI is ″newscontroller″ so this is the cache-entry's <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 is cached 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. Finally, the optional session attribute for the user's layout is included in the cache ID. The cache entry now is :

```
<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. Since a category dependency ID is already defined, define an invalidation rule to invalidate the category when action=update. To incorporate the conditional logic, we will 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 is created and run. The final cache-entry now looks like this:

```
<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>
```

Accessing dynamic cache PMI counters:

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});

Important: Cache instance names are prepended with cache type (″Servlet: ″ or ″Object: ″).

Counter definitions for Servlet Cache

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. MaxInMemoryCache EntryCount	WSDynamicCacheStats.NAME - "Servlet: instance_1"	The maximum number of in-memory cache entries.	5.0 and later

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. InMemoryCache EntryCount	`WSDynamicCacheStats.NAME` `- "Servlet: instance_1"`	The current number of in-memory cache entries	5.0 and later

WSDynamicCacheStats.
HitsIn MemoryCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The number of requests for cacheable objects that are served from memory.	5.0 and later

WSDynamicCacheStats.
HitsOnDiskCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The number of requests for cacheable objects that are served from disk.	5.0 and later

WSDynamicCacheStats.
ExplicitInvalidationCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The number of explicit invalidations.	5.0 and later

WSDynamicCacheStats.
LruInvalidationCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1`
		The number of cache entries that are removed from memory by a Least Recently Used (LRU) algorithm. instance.	5.0 and later

WSDynamicCacheStats.
TimeoutInvalidationCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The number of cache entries that are removed from memory and disk because their timeout has expired.	5.0 and later

WSDynamicCacheStats.
InMemoryAndDisk
CacheEntryCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The current number of used cache entries in memory and disk.	5.0 and later

WSDynamicCacheStats.
RemoteHitCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The number of requests for cacheable objects that are served from other Java virtual machines within the replication domain.	5.0 and later

WSDynamicCacheStats.
MissCount
	`WSDynamicCacheStats.NAME`
	`- "Servlet:`
	`cache_instance_1" -`
	`WSDynamicCacheStats.`
	`TEMPLATE_GROUP`
	`-"Template_1"`
		The number of requests for cacheable objects that were not found in the cache.	5.0 and later

Name of PMI statistics	Path	Description	Version
WSDynamicCacheStats. ClientRequestCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of requests for cacheable objects that are generated by applications running on this application server.	5.0 and later
WSDynamicCacheStats. DistributedRequestCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of requests for cacheable objects that are generated by cooperating caches in this replication domain.	5.0 and later
WSDynamicCacheStats. ExplicitMemory InvalidationCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of explicit invalidations resulting in the removal of an entry from memory.	5.0 and later
WSDynamicCacheStats. ExplicitDisk InvalidationCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of explicit invalidations resulting in the removal of an entry from disk.	5.0 and later
WSDynamicCacheStats. LocalExplicit InvalidationCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of explicit invalidations generated locally, either programmatically or by a cache policy.	5.0 and later
WSDynamicCacheStats. RemoteExplicit InvalidationCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of explicit invalidations received from a cooperating Java virtual machine in this replication domain.	5.0 and later
WSDynamicCacheStats. RemoteCreationCount	`WSDynamicCacheStats.NAME - "Servlet: cache_instance_1" - WSDynamicCacheStats. TEMPLATE_GROUP -"Template_1"`	The number of cache entries that are received from cooperating dynamic caches.	5.0 and later

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

Name of PMI statistics	Path	Description	Version
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
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

Name of PMI statistics	Path	Description	Version

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

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. MaxInMemoryCache EntryCount	`WSDynamicCacheStats.NAME` `- "Object: instance_2"`	The maximum number of in-memory cache entries.	5.0 and later
WSDynamicCacheStats. InMemoryCache EntryCount	`WSDynamicCacheStats.NAME` `- "Object: instance_2"`	The current number of in-memory cache entries	5.0 and later

Name of PMI Statistics	Path	Description	Version

WSDynamicCacheStats.
HitsInMemoryCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.OBJECT_GROUP`
	`- WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of requests for cacheable objects that are served from memory.	5.0 and later

WSDynamicCacheStats.
HitsOnDiskCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of requests for cacheable objects that are served from disk.	5.0 and later

WSDynamicCacheStats.
ExplicitInvalidationCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of explicit invalidations.	5.0 and later

WSDynamicCacheStats.
LruInvalidationCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of cache entries that are removed from memory by a Least Recently Used (LRU) algorithm. instance.	5.0 and later

WSDynamicCacheStats.
TimeoutInvalidation Count
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of cache entries that are removed from memory and disk because their timeout has expired.	5.0 and later

WSDynamicCacheStats.
InMemoryAndDisk
CacheEntryCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The current number of used cache entries in memory and disk.	5.0 and later

WSDynamicCacheStats.
RemoteHitCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of requests for cacheable objects that are served from other Java virtual machines within the replication domain.	5.0 and later

WSDynamicCacheStats.
MissCount
	`WSDynamicCacheStats.NAME`
	`- "Object:`
	`cache_instance_2" -`
	`WSDynamicCacheStats.`
	`OBJECT_GROUP -`
	`WSDynamicCacheStats`
	`OBJECT_COUNTERS`
		The number of requests for cacheable objects that were not found in the cache.	5.0 and later

Name of PMI Statistics	Path	Description	Version
WSDynamicCacheStats. ClientRequestCount	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of requests for cacheable objects that are generated by applications running on this application server.	5.0 and later
WSDynamicCacheStats. DistributedRequest Count	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of requests for cacheable objects that are generated by cooperating caches in this replication domain.	5.0 and later
WSDynamicCacheStats. ExplicitMemory InvalidationCount	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of explicit invalidations resulting in the removal of an entry from memory.	5.0 and later
WSDynamicCacheStats. ExplicitDisk InvalidationCount	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of explicit invalidations resulting in the removal of an entry from disk.	5.0 and later
WSDynamicCacheStats. LocalExplicit InvalidationCount	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of explicit invalidations generated locally, either programmatically or by a cache policy.	5.0 and later
WSDynamicCacheStats. RemoteExplicit InvalidationCount	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of explicit invalidations received from a cooperating Java virtual machine in this replication domain.	5.0 and later
WSDynamicCacheStats. RemoteCreationCount	`WSDynamicCacheStats.NAME` `- "Object:` `cache_instance_2" -` `WSDynamicCacheStats.` `OBJECT_GROUP -` `WSDynamicCacheStats` `OBJECT_COUNTERS`	The number of cache entries that are received from cooperating dynamic caches.	5.0 and later

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
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

Name of PMI statistics	Path	Description	Version
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
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 2 platform, Enterprise Edition (J2EE) applications and system components can cache and share Java objects by storing a reference to the object in the cache.

Enable the dynamic cache service. See Enabling the dynamic cache service for more information.

The DistributedMap and DistributedObjectCache interfaces are simple interfaces for the dynamic cache. Using these interfaces, J2EE 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 that can be set using "Object cache instance settings" on page 1198.

Tip: For more information about the DistributedMap and DistributedObjectCache interfaces, see the API documentation for the com.ibm.websphere.cache package. See "Reference: Generated API documentation" on page 26 for more information.

Important: 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.

There are three methods for configuring and using cache instances.

- **Method 1 - Administrative console** You can create additional cache instances using the administrative console.

    1. In the administrative console, select **Resources > Object cache instances** and create a new object cache instance.

    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 2 - Properties file** You can create cache instances using the `cacheinstances.properties` file and package the file in your Enterprise Archive (EAR) file.

    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`. `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`.

You must 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 `was_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");
```

For more information about the DistributedMap and DistributedObjectCache interfaces, see the API documentation for the `com.ibm.websphere.cache` package.

- **Method 3 - Resource references**

Note: Method three is an extension to method one or method two, listed above. First use either method one or method two.

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-auth>Container</res-auth>
     <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.

Restriction: DistributedMap and DistributedObjectCache do not have authorization or access control associated with the cache entries.

To learn how to share cached objects in a clustered environment, see "Shared cached objects in a clustered environment."

Shared cached objects in a clustered environment:

This section discusses what you should do when sharing cached objects in a clustered environment.

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.

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.

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 2 Platform, Enterprise Edition (J2EE) context from registration time during callbacks. Setting the value of the custom property useListenerContext to true enables the invalidation listener J2EE 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 "Reference: Generated API documentation" on page 26 for the `com.ibm.websphere.cache` package.

Dynamic query

Using EJB query

The EJB query language is used to specify a query over container-managed entity beans. The language is similar to SQL. An EJB query is independent of the bean's mapping to a persistent store.

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()` 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.

WebSphere's EJB query language is compliant with the EJB QL defined in Sun's EJB 2.1 specification and has additional capabilities as listed in the topic Comparison of EJB 2.x 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:
  - **Application Server Toolkit.** When defining an EJB 2.1 entity bean in an EJB deployment descriptor editor, on the **Beans** page click **Add** under **Queries** and, in the Add Finder Descriptor wizard, define a `find` or `ejbSelect` method. See the online **Application Server Toolkit information** for documentation on wizard options.
  - **Rational Application Developer.** When defining an entity bean, specify the `<ejb-ql>` tag for the finder or select method.
  - **Dynamic query service.** Add the executeQuery() method to your application.

See the topic Example: EJB queries.

EJB query language: 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: EJB queries:

Here is an example EJB schema, followed by a set of example queries:

Table 47. 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 48. 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)

Table 48. EmpBean schema (continued)

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 1226. 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 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: An identification variable followed by the navigation operator (.) and a cmp or relationship name is a path expression.

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 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: 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: 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

NOT	
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 may 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: Basic predicates can be of two forms

```
expression-1  comparison-operator  expression-2
expression-3  comparison-operator ( subselect )
```

The subselect must not return more than one value and the subselect can not return a type of an 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.

```
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.

```
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 and 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 can not 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.

```
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.

```
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: To test if a multivalued relationship is empty, 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.

```
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 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: 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: EJB query contains scalar built-in functions, as listed below, for doing type conversions, string manipulation, and for manipulating date-time values.

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: 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 )  }
```

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 a object reference or CMP value. You can use the DISTINCT keyword to eliminate duplicate references.

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:

```
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 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, EXISTS predicate or 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: 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 query is compiled into 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: **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.

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.
- Current releases of UDB DB2 for i5/OS only support a TIMESTAMP value of the format 'yyyy-mm-dd-hh.mm.ss.nnnnnn'. This is not compatible with the standard format supported by the java.sql.Timestamp class, which is 'yyyy-mm-dd-hh mm.ss.nnnnnn'. The TIMESTAMP scalar function should be used to convert a string representation of a java.sql.Timestamp object to a value that can be recognized by DB2 UDB for i5/OS.

Rules for data type manipulation in EJB query: **Rules on 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 EJB query:

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:

```
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 } ]*
```

Comparison of EJB 2.1 specification and WebSphere query language: WebSphere Application Server Version supports the following extensions to the Enterprise JavaBeans 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

Following are common reasons for using the dynamic query service rather than the regular EJB query service (which can be referred to as *deployment query*):

- 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. You can consult the API specifications in "Reference: Generated API documentation" on page 26 (the section for package com.ibm.websphere.ejbquery).

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: Dynamic query remote interface:

When you run a dynamic 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. (See *Class QueryIterator* in "Reference: Generated API documentation" on page 26 for more details.) 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. (See *Class IQueryTuple* in "Reference: Generated API documentation" on page 26 for more details.) 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: Dynamic query local interface:

When you run a dynamic 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. (See *Class QueryIterator* in "Reference: Generated API documentation" on page 26 for more details.) 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: **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 datastore. 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 181 and "Concurrency control" on page 182.

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 705 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 EJB query, the application server must parse the query string into 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);
```

Comparison of the dynamic and deployment EJB query services:

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 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 1239.

- 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 2 Platform, Enterprise Edition (J2EE) 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 1248.

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.

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 2 Platform, Enterprise Edition (J2EE) 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 1248.

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.

The following language versions are available:
- Brazilian Portuguese
- Chinese (Simplified)
- Chinese (Traditional)
- English
- French
- German
- Italian
- Japanese
- Korean
- Spanish

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 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 instructions and examples"
- "Programming specifications"

Programming instructions and examples
- Java internationalization tutorial

  An online tutorial that explains how to use the Java 2 SDK Internationalization API.
- Globalize your On Demand Business

  IBM's portal site for delivering globalized applications.

Programming specifications
- Java 2 SDK, Standard Edition Documentation: Internationalization

  The Java 1.4.2 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 J2EE internationalization service that is currently being 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.

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 2 Platform, Enterprise Edition (J2EE) application.
7. Deploy and manage the application.

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.

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.

Create message catalogs for the language-specific strings.

Creating message catalogs

Perform this task to begin the localization of strings in an application component.

Identify strings that need to be localized.

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."

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
```

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.

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.

When the application is finished, deploy your application. For more information, see "Preparing the localizable-text package for deployment" on page 1247.

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 1238.

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 1243 at the end of this topic. For a usage example, see "Composing complex strings" on page 1245.

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 1246.

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.

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 1241.

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.

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.″

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.

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.

Write code to generate the localized text.

Composing complex strings:

Perform this task to insert variable substrings into a localized string.

Identify strings that need to be localized.

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.

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.

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.

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.

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.

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);
   }
   ...
}
```

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.

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.

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.

Write code to compose the language-specific strings.

1. Make sure that the LocalizableTextEJBDeploy tool is included in the class path.

   transition: 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.

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.

transition: 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.

With the internationalization service, you can manage the distribution of the internationalization information, or *internationalization context*, that is necessary to perform localizations within Java 2 Platform, Enterprise Edition (J2EE) 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 J2EE client programs and stateless session beans in the same manner that you would use conventional J2EE 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 J2EE 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 J2EE 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 1236.

Java 2 Platform, Enterprise Edition (J2EE) 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 J2EE 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 2 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.

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 2 Platform, Enterprise Edition (J2EE) 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" on page 1251
   - "Setting the internationalization type for enterprise beans" on page 1252

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"
- "Configuring container internationalization for enterprise beans" on page 1253

Setting the internationalization type for servlets:

This task sets the internationalization type for a servlet within a Web module.

This topic assumes that you have an assembly tool such as Application Server Toolkit (AST) or Rational Application Developer.

For information about assembly, refer to the online documentation or information center for your assembly tool. This topic points you to AST documentation. The AST information center accompanies the WebSphere Application Server information center.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules, and created or imported a dynamic Web project. Refer to the following topics in AST documentation:

- Starting WebSphere Application Server Toolkit
- Configuring WebSphere Application Server Toolkit
- Creating a dynamic Web project
- Importing Web archive (WAR) files

1. In the J2EE 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**.

The internationalization type setting is assigned to the servlet.

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.

This topic assumes that you have an assembly tool such as Application Server Toolkit (AST) or Rational Application Developer.

For information about assembly, refer to the online documentation or information center for your assembly tool. This topic points you to AST documentation. The AST information center accompanies the WebSphere Application Server information center.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules, and created or imported a dynamic Web project. Refer to the following topics in AST documentation:
- Starting WebSphere Application Server Toolkit
- Configuring WebSphere Application Server Toolkit
- Creating a dynamic Web project
- Importing Web archive (WAR) files

You must also have set the internationalization type of one or more servlets in a Web project to `Container`.

This procedure relates one or more servlets to a container-managed internationalization attribute.
1. In the J2EE 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**.

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.

This topic assumes that you have an assembly tool such as Application Server Toolkit (AST) or Rational Application Developer.

For information about assembly, refer to the online documentation or information center for your assembly tool. This topic points you to AST documentation. The AST information center accompanies the WebSphere Application Server information center.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules, and created or imported an EJB project. Refer to the following topics in AST documentation:
- Starting WebSphere Application Server Toolkit
- Configuring WebSphere Application Server Toolkit
- Creating EJB projects
- Importing EJB JAR files

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 J2EE 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**.

The internationalization type is assigned to the bean.

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.

This topic assumes that you have an assembly tool such as Application Server Toolkit (AST) or Rational Application Developer.

For information about assembly, refer to the online documentation or information center for your assembly tool. This topic points you to AST documentation. The AST information center accompanies the WebSphere Application Server information center.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules, and created or imported an EJB project. Refer to the following topics in AST documentation:
- Starting WebSphere Application Server Toolkit
- Configuring WebSphere Application Server Toolkit
- Creating EJB projects
- Importing EJB JAR files

You must also have one or more enterprise beans set to container-managed internationalization (CMI) by default.

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 J2EE 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**.

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.

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.

The resulting components are said to use *application-managed internationalization* (AMI). For more information about AMI, see "Internationalization context: Management policies" on page 1269.

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.

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.

An application component must first resolve a reference to the CallerInternationalization object and then bind it to the Internationalization interface.

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.

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.

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);
    ...
```

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: 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: 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.
```

1260 Developing and deploying applications

```
      //-------------------------------------------------------------------
      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: 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:

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 J2EE 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.

Tip:

> 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 1257.

Servlets

On every servlet service method (doGet or doPost) invocation, the J2EE 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 J2EE 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 J2EE 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.

Tip:

> 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 1257.

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: 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>
```

Internationalization context: Management policies:

Internationalization policies prescribe how J2EE 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)
- 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 a WebSphere 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 J2EE 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.

Remember: 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.

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.

After you have completed these steps, applications can find the object pool manager by doing a JNDI lookup using the specified JNDI name.

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);

}
```

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. 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 J2EE application tasks to run at a requested time. Schedulers also enable application developers to create their own stateless session EJB components to receive event notifications during a task life cycle, allowing the plugging-in of custom logging utilities or workflow applications.

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	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 article includes instructions for creating and configuring schedulers, creating and configuring a database for schedulers and administering schedulers.

2. Develop and schedule tasks. This article 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 2 Platform Enterprise Edition (J2EE) application client container is not supported.

3. Interoperate with schedulers. This article explains how to manage scheduler in a clustered environment with mixed WebSphere Application Server product versions and mixed platforms.

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 J2EE application.

The `SchedulerCalendars.ear` application is available and provides a default UserCalendar 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 J2EE 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 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 J2EE 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.

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 J2EE server application, such as a servlet or EJB component can use the Scheduler API. Schedulers are accessed by looking them up using a 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 J2EE 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 aUserCalendar 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 J2EE server application, such as a servlet or EJB module. Use a JNDI name or resource reference to access schedulers. Each scheduler is also available using the JMX API, using its associated WASScheduler MBean.

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 J2EE 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
```

The scheduler is now available to use from a J2EE 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.

To create a task to call a method on a TaskHandler session bean, use these steps.

1. Create a new enterprise application with an 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 schedO [$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 $schedO 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 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."
```

A BeanTaskInfo object has been created that contains all of the relevant data to call an EJB method.

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.

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."
```

A MessageTaskInfo object has been created that contains all of the relevant data for a task that sends a JMS message.

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.

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.

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.

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 JNDI name must be the fully-qualified global JNDI name. Using a JavaServer Pages (JSP) file, servlet or 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
```

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.

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 JMS message.

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.

Managing tasks with 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 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.

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.

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 J2EE context:

When a task is created using the Scheduler API create() method, the Java 2 Enterprise Edition (J2EE) thread context of the creator is stored with the scheduled task. When the task runs, the original J2EE 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 J2EE 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.

The TaskInfo.setQOS method allows reducing the quality of service for the task to run at-least-once. When you set the task to TaskInfo.QOS_ATLEASTONCE, the task does not keep a transaction open for the duration of the running task. Because this quality of service is no longer transactional, the task may fire more than one time if the scheduler fails to update the result of the task. This quality of service is ideal for batch jobs. Use QOS_ATLEASTONCE with check points in the business logic, so if the task needs to recover, the business logic can continue at the next check point. See the com.ibm.websphere.scheduler.TaskInfo API documentation for more details.

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 JDBC datasource configured for the scheduler, any 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 article, "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 J2EE 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 J2EE 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 EJB
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 list for more information.

Securing scheduler tasks:

Scheduled tasks are protected using application isolation and administrative roles. This topic describes how to secure scheduler tasks.

If a task is created using a Java 2 Platform, Enterprise Edition (J2EE) 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 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 JNDI name, 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 J2EE 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 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 1293, 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 that can be used to create a task. Several implementations of this class exist, one for each type of task that can be run.

Available TaskInfo implementations include:

BeanTaskInfo

Calls a stateless session bean.

MessageTaskInfo

Sends a 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 the Application Server Toolkit (AST) or 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 the Application Server Toolkit (AST) or 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 the Application Server Toolkit (AST) or 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 2 Platform Enterprise Edition (J2EE) 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.

A module startup bean is a session bean that is loaded when an EJB Jar file starts. Module startup beans enable Java 2 Platform Enterprise Edition (J2EE) 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 2 Platform Enterprise Edition (J2EE) 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.

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. (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.

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

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.

Overview of work area service: 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.

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 the property can be removed or modified.

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. 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. Nested work areas allow applications to define and scope properties for specific tasks without having to make them available to all parts of the application. 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 22. 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.

Work Area 2		
key	**value**	**mode**
key1	A	normal
key2	B	fixed normal
key3	C	read-only
key4	D	fixed read-only

Visible to Work Area 2
key1 = A
key2 = B
key3 = C
key4 = D

Work Area 2.1		
key	**value**	**mode**
key1	X	normal
key2	Y	fixed normal
key5	E	normal
key6	F	fixed

Visible to Work Area 2.1
key1 = X (overridden in 2.1)
key2 = Y (overridden in 2.1)
key3 = C
key4 = D
key5 = E
key6 = F

Figure 23. Redefining existing properties in nested work areas

Distributed work areas: The propagation of work area context operates differently depending on whether a work area partition is defined as bidirectional or not. In either case all work area context propagates to a target object on a remote invocation. However, whether the context propagates from a target object back to the originator depends on whether a partition is defined as bidirectional.

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.

best-practices: 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

Applications interact with the work area service by using the UserWorkArea interface and its implementation. This interface defines all of the methods used to create, manipulate, and complete work areas:

1. Access a partition by either:
   - "Accessing the UserWorkArea partition" on page 1310, 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 a user defined partition can be used in the same way.
2. Beginning a work area.

3. Setting properties in a work area.
4. Using a work area to manage local work.
5. Completing a work area.

An example application, the Work area SimpleSample application, is used throughout this documentation to illustrate these tasks.

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** > **Developer** > **API documentation** > **Application programming interfaces**.

UserWorkArea interface: 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.

Exceptions

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.

Example: WorkArea SimpleSample application: In this 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.

The implementation of this application is discussed in the topic, Developing applications that use work areas.

Accessing the UserWorkArea partition:

The work area service provides a JNDI binding to an implementation of the UserWorkArea interface under the name java:comp/websphere/UserWorkArea. This is the default work area partition, namly the "UserWorkArea" partition. It is created and bound into JNDI naming automatically, as long as it is enabled as defined in Enabling the work area service (UserWorkArea partition). Applications that need to access UserWorkArea partition can perform a lookup on that JNDI name, as shown in the following code example:

```
import com.ibm.websphere.workarea.*;
import javax.naming.*;

public class SimpleSampleServlet {
  ...

  InitialContext jndi = null;
  UserWorkArea userWorkArea = null;
  try {
     jndi = new InitialContext();
     userWorkArea = (UserWorkArea)jndi.lookup(
        "java:comp/websphere/UserWorkArea");
  }
  catch (NamingException e) { ... }
}
```

Rather than using this default work area partition, a user has the option to create their own work area partition using the Work area partition service.

The next step is to use the begin method to create a new work area and associate it with the calling thread, as described in the topic, Beginning a new work area.

Beginning a new work area:

Be sure that your client has a reference to the UserWorkArea interface, as described in the topic Accessing the work area service 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.

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.

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** > **Developer** > **API documentation** > **Application programming interfaces**.

Using a work area

Setting properties in a 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. (See ″Setting property modes″, later in this topic.)

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 "Example: WorkArea SimpleSample application" on page 1309 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

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** > **Developer** > **API documentation** > **Application programming interfaces**.

Using a work area to manage local work:
Be sure that your client has a reference to the UserWorkArea interface, as described in the topic
"Accessing the UserWorkArea partition" on page 1310 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.

In a business application that uses work areas, server objects typically retrieve the work area properties
and use them to guide local work.

1. Retrieving the name of the active work area This step determines whether the calling thread is
   associated with a work area.

2. Overriding work area properties. Server objects can override client work area properties by creating
   their own, nested work area.

3. Retrieving properties from a work area

4. "retrieveAllKeys method" on page 1315
5. Querying the mode of a work area property
6. Deleting a work area property
7. Completing a work area

The server side of the "Example: WorkArea SimpleSample application" on page 1309 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.

Retrieving the name of the active work area:
Applications use the getName method on the UserWorkArea interface to retrieve the name of the current work area. This is the recommended method for determining whether the thread is associated with a 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();
        ...
    }
}
```

For additional information about work area, see the package, com.ibm.websphere.workarea, in the API documentation. The generated API documentation is available in the information center table of contents from the path **Reference** > **Developer** > **API documentation** > **Application programming interfaces**.

Overriding work area properties:

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.

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 Bidirectional propagationfor 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) {
        }
        ...
  }
}
```

Retrieving work area properties:

Properties can be retrieved from a work area by using the get method. This 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");
       ...
  }
}
```

For additional information about work areas, see the package, com.ibm.websphere.workarea, in the API documentation. The generated API documentation is available in the information center table of contents from the path **Reference** > **Developer** > **API documentation** > **Application programming interfaces**.

retrieveAllKeys method:

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** > **Developer** > **API documentation** > **Application programming interfaces**.

Querying the mode of a work area property:
The UserWorkArea interface provides the getMode method for determining the mode of a specific property. This method takes the property's key as an argument and returns the mode as a PropertyModeType object. (See Setting property modes for more information on names of mode types.) 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.

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** > **Developer** > **API documentation** > **Application programming interfaces**.

Deleting a work area property:

The UserWorkArea interface provides the remove method for deleting 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.

For additional information about work area, see the package, com.ibm.websphere.workarea, in the API documentation. The generated API documentation is available in the information center table of contents from the path **Reference** > **Developer** > **API documentation** > **Application programming interfaces**.

Completing a 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) {
      }
   }
}
```

Chapter 5. Rapid deployment of J2EE applications

The rapid deployment tools extends the mechanism for generating deployment artifacts, packaging the application and preparing the application to run only on WebSphere Application Server v6.x.

You can use rapid deployment tools to:

1. Create a new J2EE application quickly without the overhead of using an integrated development environment (IDE).
2. Package J2EE artifacts quickly into an EAR file.
3. Test and deploy J2EE artifacts quickly on a server.

For example, you can place fully composed enterprise applications (EAR files), application modules (WAR files, EJB JAR files), or application artifacts (Java source files, Java class files, images, XML, or HTML) into a configurable location on your file system, referred to as the *monitored* directory. The rapid deployment tool automatically detects added or changed parts of these J2EE artifacts and performs the necessary steps to produce a running application on the server.

There are two ways to configure the monitored directory, each performing separate and distinct tasks. You can either specify the monitored directory as a free-form project or as an automatic installation project.

Free-form project

The rapid deployment tools free you from understanding J2EE application structure, as well as having to update and modify basic deployment descriptor information, such as new servlet or EJB entries. With the free-form approach you can place in a single project directory the individual parts of your application, such as Java source files that represent servlets or enterprise beans, static resources, XML files and other supported application artifacts. The rapid deployment tools then use your artifacts dropped in this free-form project directory to automatically place them in the appropriate J2EE project structure, generate any additional required artifacts to construct a J2EE compliant application and deploy that application on a target server.

Example

Assume that you want to set up a simple Web application on a server using the free-form approach. You have created the Java source files that implement a group of servlets. In addition, you have created the JSP files, images, and static HTML files to define the application. Using the free-form approach, you simply place all of these files together, or one at a time, in the free-form project directory. As J2EE artifacts are placed inside this free-form project, the rapid deployment tools dynamically generate the necessary J2EE project structure and deployment descriptors. For example, the deployment descriptor includes Web deployment descriptor (web.xml). The J2EE project structure includes an enterprise application project, a Web module project, an EJB module project, an EJB client JAR project, and a utility Java project.

The rapid deployment tools detect additions or changes to J2EE artifacts in the free-form project and run a build. As you drop files into the free-form project, the necessary J2EE projects are created, if these projects do not already exist. For example, if you drop a servlet source file, a Web project and an Application project is created if these projects do not exist in your workspace. Then the rapid deployment tool maps the artifacts to their correct location in the appropriate project. In this case, the servlet source file is compiled and its compiled class files are copied to *MyProject*Web\WebContent\WEB-INF\classes. The rapid deployment tools then generate or update any additional required deployment artifacts. In this case, the web.xml for the Web project is updated to include a servlet and servlet mapping entry.

The rapid deployment publishes the updated application on the server. If you choose to use the optional **earExportPath** option that specifies the directory location of the output EAR file, an EAR file is created. If the target server is remote, it will generate an EAR file and transfers the EAR to the remote server.

Benefits

The benefits of using the free-form approach is that you can focus on the structure of your application independent of the requirements of J2EE or other standards. One of the complexities of J2EE development is that it defines the location of artifacts in an application relative to one another. For example, in a Web application, class files must be placed in WEB-INF/classes and JSP files must be placed outside of the WEB-INF folder. This structural requirement must be understood and followed during development in order for the application to operate correctly. The rapid deployment tools removes this structural requirement, allowing the source project to follow your development defined structure and freeing you from having to understand the J2EE defined structures.

Automatic installation project

The rapid deployment tools simplify the process of installing, modifying, and uninstalling applications on a server. Automatic installation relies on a single project directory that listens for J2EE compliant applications (EAR files) or modules (WAR, JAR, or RAR files). The rapid deployment tools then deploy the artifacts to the target server.

If you place EAR files in the automatic installation project, the EAR file is automatically deployed to the server. If you delete EAR files from project, the application is uninstalled from the server. If you place a new copy of the same EAR file in the automatic installation project, then the application is reinstalled. If you place WAR or EJB JAR files in the automatic application installation project, the rapid deployment tool generates the necessary EAR wrapper and then publishes that EAR file on the server. For RAR files, a wrapper is not created. The standalone RAR files are published to the server.

Benefits

The benefit of using the automatic installation project is that it frees you from the complexity of administering your application. For example, using **wsadmin** scripting program or the administrative console to install the application through a multiple-step wizard.

Remember: If you want to go into any of the generated projects, for example, to modify the deployment descriptor, you should not do it here. First switch to a tool designed for such advanced tasks, such as the WebSphere Application Server Toolkit or one of the IBM Rational Software Development Platform.

Constructing a J2EE application from artifacts

The *free-form* configuration allows you to create or drop your J2EE artifacts or module files into a free-form project directory. J2EE artifacts can include the source, annotated-source, or class files of servlets, JavaServer Pages (JSP) files, static Web content, Web filters, Web listeners, enterprise beans, and other generic files. The rapid deployment tools automatically place your artifacts in the appropriate J2EE project structure, generate any additional required artifacts to construct a J2EE compliant application, and deploy that application on a target server.

Steps

To construct a J2EE application from artifacts:

Setting up a rapid deployment environment

You can run the rapid deployment batch tools from a command line to create, configure, and launch rapid deployment projects using the IBM WebSphere v6.x runtime environment. The rapid deployment tools run in a non-graphical command line interface.

Prerequisites

- WebSphere Application Server v6.x installed
- The rapid deployment launch tool is in the *x:*/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:
  - ▶ Windows   wrd.bat
  - ▶ UNIX   wrd.sh
- The rapid deployment configuration tool is in the *x:*/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:
  - ▶ Windows   wrd-config.bat
  - ▶ UNIX   wrd-config.sh

To set up a rapid deployment workspace:

1. Open a command prompt.
2. Specify the location on your file system that will host the rapid deployment activities. This location serves as your workspace root, and all your rapid deployment projects reside here. Define an environment variable called WORKSPACE:

   - ▶ Windows   `set WORKSPACE=<workspace root>`

   - ▶ UNIX   `export WORKSPACE=<workspace root>`

   Configure a rapid deployment workspace for each rapid deployment project, rather than configuring multiple rapid deployment projects inside a single rapid deployment workspace.

   Tip: To list all your environment variables and their values, run the command:

   - ▶ Windows   `set "`

   - ▶ Linux   `export -n`

   - ▶ AIX   ▶ Solaris   ▶ HP-UX   `export`

After the WORKSPACE environment variable is defined, you can continue to configure and launch your rapid deployment workspace.

Creating a free-form project in your workspace

The *free-form* project allows the creation of a J2EE application using a single project directory for development. Free-form development releases you from having to maintain multiple projects as well as having to understand the J2EE application structure.

Prerequisite

- Learn how to work with free-form projects
- Set up a rapid deployment environment.

Steps for this task

1. From the command prompt, change to the *x:*/profiles/*<profileName>*/bin directory, where *x* is the directory WebSphere Application Server is installed.
2. Use one of the following options to configure a new free-form project:
   - To configure a new free-form project, you can complete this by typing the following at the prompt:
     - ▶ Windows   `wrd-config.bat -project "project_name" -style "freeform"`
     - ▶ UNIX   `wrd-config.sh -project "project_name" -style "freeform"`
   - If you want to specify a particular runtime environment to compile your application against, use the optional `runtime` and `runtimePath` parameters. Targeting your runtime will configure which JRE

library to use, based on the runtime location, and configure the project's classpath to contain the WebSphere runtime libraries. You can complete this by typing the following at the prompt:

– `Windows` `wrd-config.bat -project "project_name" -style "freeform" -runtime "was60"` `-runtimePath "x:\WebSphere\AppServer"`

– `UNIX` `wrd-config.sh -project "project_name" -style "freeform" -runtime "was60"` `-runtimePath "x:\WebSphere\AppServer"`

where *x:\WebSphere\AppServer* is the directory WebSphere Application Server is installed.

> **Tip:** The `runtimePath` is not used for targeting the server to publish against but rather specifying the WebSphere libraries you want your application to compile against. If the `runtime` and `runtimePath` are not specified, the default runtime environment used is WebSphere Application Server v6.1.

3. The **Parameter Configuration Settings** dialog box opens. Specify the value for one or more of the following settings:

Option	Description
earExportPath	This is an optional field to specify the directory location of the output EAR file that is created by rapid deployment tools and contains the generated classes required for deployment. For example: c:\temp\export. **Note:** • When you specify the directory for the output EAR file and then run the **wrd-config** command, any existing output EAR file of the same name will be overwritten without warning. In addition, the EAR file gets overwritten as additional J2EE artifacts are added to the free-form project to reflect the new changes.
serverName	The name of the server process you want to publish your application. For example, server1. For WebSphere Application Server Network Deployment, the server name is in the form *<cell name>/<node name>/<server name>*. To publish to a cluster, the server name is in the form *<cell name>/<cluster name>*.
username	(Optional) If security is enabled, specify the user name for current active authentication settings defined in the server configuration.
password	(Optional) If security is enabled, specify the password for current active authentication settings defined in the server configuration.
serverJMXHost	The host name of the machine containing the server you want to make a connection to. For example, localhost. For WebSphere Application Server Network Deployment, type the host name of the Network Deployment Manager.
serverJMXPort	The server administrative port number, also known as the Simple Object Access Protocol (SOAP) connector port. This port is used for making JMX connections with the server. For example, 8880. For WebSphere Application Server Network Deployment, type the SOAP port number of the Network Deployment Manager, default is 8879.

A *projectName*_headless.xml file is generated at the root of your rapid deployment workspace. You can use this file later to configure a free-form project with the same configuration without having to be prompted again for these parameters. Use the optional **configData** parameter to complete this task.

Results

You can see that the initial configuration of the free-form project creates only the free-form project. As artifacts are introduced and modified in the free-form project, rapid deployment tools will dynamically generate the necessary J2EE project structures inside the workspace directory. The following table lists the naming convention of the project folder with its associated project type:

Naming Convention of the Project Folder	Project Type
project_name	The free-form project
*project_name*App	A single enterprise application project
*project_name*EJB	A single EJB module project
*project_name*EJBClient	A single EJB client jar project
*project_name*Utility	A single utility Java project
*project_name*Web	A single Web module project

where *project_name* is the string value provided for the **-project** parameter.

After you have configured the workspace to contain a free-form project and the supporting J2EE projects, you will then launch a rapid deployment session. Launching a rapid deployment session indicates that you want the free-form project to start listening for your J2EE artifacts.

Starting a rapid deployment session

You can launch a rapid deployment session to start listening for application artifacts that are dropped into the folder you configured for either the free-form or automatic installation approach. Use the **wrd** command to launch a rapid deployment session. If you want to monitor rapid deployment activities, you can specify console output.

Prerequisites
- Set up a rapid deployment environment.
- Configure the behavior of the monitored directory as free-form or automatic installation.

Steps
1. From the command prompt, change to the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server is installed.
2. Verify that the server on which you want to deploy your application is running.

   Tip: If there is no server connection, you get the following console output:
   ```
   [06:52:28 PM] Publishing MyProjectApp to server_510658053
   [06:52:28 PM] Failed to make connection to Websphere Application Server.
   ```
3. At the command prompt, type the appropriate command:
   - (Recommended) To monitor rapid deployment activities:
     - ▶ Windows
     - ▶ UNIX `wrd.sh -monitor`
   - To launch a rapid deployment session:
     - ▶ Windows `wrd.bat`
     - ▶ UNIX `wrd.sh`
   - To use batch mode to run a full build of the workspace and then shut down the process:
     - ▶ Windows `wrd.bat -project "project_name" -batch`
     - ▶ UNIX `wrd.sh -project "project_name" -batch`

After you launch the rapid deployment tools, you can create or drop your J2EE artifacts into the monitored directory. The application generated is deployed and updated on the target server.

Dropping J2EE artifacts into a free-form project

You can place application artifacts in a free-form project and let the rapid deployment tools automatically place them in the appropriate J2EE project structure, generate any additional required artifacts to construct a J2EE compliant application, and deploy that application on a target server.

Prerequisites

- Learn how to work with free-form projects
- Set up a rapid deployment environment.
- Create a free-form project in your workspace.

Steps

1. Verify the server you want to deploy your application is running.

   Tip: If there is no server connection, you get the following console output:

   ```
   [06:52:28 PM] Publishing MyProjectApp to server_510658053
   [06:52:28 PM] Failed to make connection to Websphere Application Server.
   ```

2. Launch a rapid deployment session by using the **wrd** command.

3. Use your file management system to place your application artifacts properly into a free-form project. The directory name of the free-form project has the same string value you provided for the **-project** parameter when you created the free-form project.

   Note: When you create or drop in your J2EE artifacts into the free-form project, these resources will automatically be placed in the appropriate location in the J2EE project structure. You might experience approximately five seconds delay for the changes in the files system to be picked up, as these changes are being polled from the workspace.

Tip:

The rapid deployment tools preserve the folder structures created and dropped into a free-form project. When dropping Java source files, place these files in the correct package structure to avoid compilation errors. For example, suppose a Java package statement, package java.src;, is defined in a Java source file, HelloWorld.java. Simply dropping the Java source file, HelloWorld.java, in the file path, *workspace*/MyFreeForm/HelloWorld.java, where *workspace* is the directory where your rapid deployment project resides, results in the following compilation error, displayed in the console output:

```
[06:32:19 PM] [/MyFreeForm/HelloWorld.java] Added
[06:32:20 PM] [/MyFreeForm/bin/HelloWorld.class] copied to project [MyFreeFormUtility]
[06:32:20 PM] 'The declared package does not match the expected package ' in resource 'HelloWorld.java'
```

To correct the compilation error, place the HelloWorld.java source file in the following file path *workspace*/MyFreeForm/java/src/HelloWorld.java

The following topics show how to handle supported artifacts in free-form projects:

Free-form projects

The initial configuration of the free-form project in your workspace creates a free-form project.

Within the free-form project, you can drop or create J2EE artifacts or fully-composed J2EE modules. As artifacts are introduced or modified in the free-form project, the necessary J2EE project structures are dynamically generated in the workspace directory. The J2EE project structures at most contain a Web module project, an EJB module project, an EJB client jar project and a utility Java project; all belonging to

a single enterprise application project. From a J2EE development perspective, the free-form project supports a single J2EE application that contains two modules: a web module and an EJB jar module.

The following table lists the naming convention for dynamically generated project folder with its associate project type in a workspace:

Naming conventions for the project folder	Project type
project_name	The free-form project
*project_name*App	A single enterprise application project
*project_name*EJB	A single EJB module project
*project_name*EJBClient	A single EJB client jar project
*project_name*Utility	A single utility Java project
*project_name*Web	A single Web module project

where *project_name* is the string value provided for the **-project** parameter.

The rapid deployment tools will configure and update the manifest files and project references to maintain the following default settings:
- The EJB module project will have reference and visibility to the EJB client jar project.
- The Web module project will have reference and visibility to the EJB client jar project.
- The Web module project, EJB module project, and EJB client jar project will have reference and visibility to the Utility Project.
- The enterprise application project will contain the Web module project, EJB module project, and EJB client jar project.
- The utility Java project is added as a utility Jar inside the enterprise application project.
- Each of the projects, excluding the enterprise application project, contains an imported_classes directory. The imported_classes folder is set as an exported entry on the classpath of its containing project. As a result, the directory and any contents within it is visible to any projects referencing that project.

The initial configuration of the free-form project creates subdirectories:
- The gen folder is a repository of non-Java artifacts that get generated by the rapid deployment tools.
- The src folder is a subdirectory of the gen folder and is a repository of Java artifacts generated by rapid deployment tools. For example, if you drop an annotated session bean, the remote, home and local interfaces are generated in src directory.
- The bin directory contains the class files that were compiled for any Java source files dropped into the free-form project.

When Java source files are dropped into a free-form project, they are compiled into this project and the generated class files are copied into the target J2EE project's imported_classes folder. The source files are not copied into any of the J2EE project structures. The deployed application contains only class files.

If a classification of a resource changes from one artifact type to another that results in changes to its target J2EE module project, the old mapped locations are purged. For example, if A.java implemented a servlet, its class files map to the Web module project. If the content of A.java changes to implement an enterprise bean, its class files now map to the EJB module project. The dangling class in the Web project is removed.

Any resources that cannot be classified by rapid deployment tools are mapped to the utility Java project, and their folder structure is preserved as created and dropped into the free-form project. For example, if the following resource exists in the free-form project, /MyFreeForm/data/myproperties.props. This file is

mapped to /MyFreeFormUtility/data/myproperties.props, preserving the data folder. In addition, if the class files cannot be classified by rapid deployment as other than a class file, the tool maps these files to the imported classes folder in the utility Java project.

When removing artifacts from the free-form project, the rapid deployment tools remove the artifact from its mapped J2EE project structure location and if applicable, remove the deployment descriptor entries. However, the rapid deployment tools keep any project folders created.

If the free-form project is configured to a local server (on the same machine running the rapid deployment session), the application synchronizes with the server after every change. If the free-form project is configured to a remote server, to limit file transfers over the network for every change, changes are batched together in one minute intervals. A publish is triggered every minute, only when a change occurs.

Adding and deleting Web artifacts from a free-form project

You can add various artifacts to or delete various artifacts from a free-form project. The artifacts can be source code, annotation source files or compiled classes.

Adding and deleting servlets to a free-form project:

You can add or delete servlet source code, annotated source or compiled class files to or from a free-form project.

Prerequisites

The servlet class must be an implementation of the following types:

```
javax.servlet.Servlet
javax.servlet.GenericServlet
javax.servlet.http.HttpServlet
```

Steps

1. To add servlets to a free-form project, use your file management system to properly place your servlet artifacts into the free-form project. The directory name of the free-form project has the same string value you provided for the **-project** parameter used in the wrd-config command.

2. The following activities occur when you drop a servlet into a free-form project:

   a. If necessary, the servlet class is generated and mapped to the imported_classes directory of the Web module project. The rapid deployment tools will then copy the class files to its appropriate J2EE location which is WebContent/WEB-INF/classes folder of the Web module project. The console output can look like this example when adding a servlet Java source file (called *HelloServlet.java*) to a free-form project (called *MyProject*):

   ```
   [06:52:23 PM] [/MyProject/servlet/HelloServlet.java] Added
   [06:52:26 PM] [/MyProject/bin/servlet/HelloServlet.class] copied to project [MyProjectWeb]
   [06:52:27 PM] Servlet added to web.xml: HelloServlet
   [06:52:27 PM] Servlet mapping added.  URL is: [MyProjectWeb/HelloServlet]
   ```

   Tip: If you want to monitor the rapid deployment activity, run your rapid deployment session with console output. You can complete this by running the rapid deployment launch tool as follows:

   - ▶ Windows ◀ `wrd.bat -monitor`

   - ▶ UNIX ◀ `wrd.sh -monitor`

   b. A new servlet entry and servlet mapping entry is created in the Web deployment descriptor (found in the WebContent\WEB-INF\web.xml file in the Web module project). The following is an example of the entries in web.xml:

   ```
   <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>servlet.HelloServlet</servlet-class>
   </servlet>
   ```

```
<servlet-mapping>
 <servlet-name>HelloServlet</servlet-name>
 <url-pattern>/HelloServlet</url-pattern>
</servlet-mapping>
```

    c. The application is synchronized with the server.

3. You can test the servlet running on the server, use the URL displayed in the console. In this example, the console output provided the following URL mapping information to the deployed servlet:

```
[06:52:27 PM] Servlet mapping added.  URL is: [MyProjectWeb/HelloServlet]
```

Type http://*<machine_name>*:*<port>*/MyProjectWeb/HelloServlet in the address bar and press **Enter**. Where *<machine_name>* is the host machine where WebSphere Application Server runs. By default, WebSphere Application Server runs on port 9080.

4. To delete servlets from a free-form project, use your file management system to properly remove your servlet artifacts from the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

5. The following activities occur when you remove a servlet from the free-form project:

    a. The console output can look like this example when removing a servlet Java source file (called *HelloServlet.java*) from a free-form project (called *MyProject*):

```
[07:03:42 PM] [/MyProject/bin/servlet/HelloServlet.class] Deleted
[07:03:42 PM] [/MyProject/servlet/HelloServlet.java] Deleted
[07:03:42 PM] Servlet Mapping removed from web.xml /HelloServlet
[07:03:42 PM] Servlet removed from web.xml: HelloServlet
```

    b. The application is synchronized with the server.

Adding and deleting Web filters to a free-form project:

You can add or delete Web filter source code, annotated source or compiled class files for various J2EE filter types, such as authentication filters, encryption filters, and data compression filters to or from a free-form project.

Prerequisites

The filter class must be an implementation of the following type:

```
javax.servlet.Filter
```

Steps

1. To add Web filters into a free-form project, use your file management system to properly place your filter artifacts into the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

2. The following activities occurs when you drop a filter into the free-form project:

    a. If necessary, the filter class is generated and mapped to the imported_classes directory of the Web module project. The rapid deployment tools will then copy the class files to its appropriate J2EE location which is WebContent/WEB-INF/classes folder of the Web module project. The console output can look like this example when adding a filter Java source file (called *MyFilter.java*) to a free-form project (called *MyProject*):

```
[07:47:17 PM] [/MyProject/MyFilter.java] Added
[07:47:20 PM] [/MyProject/bin/MyFilter.class] copied to project [MyProjectWeb]
[07:47:20 PM] Web Filter added to web.xml MyFilter
[07:47:20 PM] Filter Mapping added to web.xml /MyFilter
```

    Tip: If you want to monitor the rapid deployment activity, run your rapid deployment session with console output. You can complete this by running the rapid deployment launch tool as follows:

        • ▶ Windows wrd.bat -monitor

        • ▶ UNIX wrd.sh -monitor

b. A new filter entry along with its display name, and any initialization parameters, URL mappings, and servlet mappings is created in the web deployment descriptor (found in the WebContent\WEB-INF\web.xml file in the Web module project). The following is an example of the entries in web.xml:

```
<filter>
 <filter-name>MyFilter</filter-name>
 <filter-class>MyFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>MyFilter</filter-name>
 <url-pattern>MyFilter</url-pattern>
</filter-mapping>
```

c. The application is synchronized with the server.

3. To delete Web filters from a free-form project, use your file management system to properly remove your filter artifacts from the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

4. The following activities occurs when you remove a filter from the free-form project:

a. The console output can look like this example when removing a filter Java source file (called *MyFilter.java*) from a free-form project (called *MyProject*):

```
[07:56:10 PM] [/MyProject/MyFilter.java] Deleted
[07:56:10 PM] [/MyProject/bin/MyFilter.class] Deleted
[07:56:10 PM] Web Filter removed from web.xml MyFilter
[07:56:10 PM] Filter Mapping removed from web.xml /MyFilter
```

b. The application is synchronized with the server.

Adding and deleting listeners to a free-form project:

You can add or delete servlet listener source code, annotated source or compiled class files for various J2EE listener types related to servlet context and session events and attributes to or from a free-form project.

Prerequisites

The listener class must be an implementation of the following types:

```
javax.servlet.ServletContextListener
javax.servlet.ServletContextAttributesListener
javax.servlet.http.HttpSessionListener
javax.servlet.http.HttpSessionAttributesListener
```

Steps

1. To add listeners to a free-form project, use your file management system to properly place your listener artifacts into the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

2. The following activities occur when you drop a listener into the free-form project:

a. If necessary, the listener class is generated and mapped to the imported_classes directory of the Web module project. The rapid deployment tools will then copy the class files to its appropriate J2EE location which is WebContent/WEB-INF/classes folder of the Web module project. The console output can look like this example when adding a listener Java source file (called *MyListener.java*) to a free-form project (called *MyProject*):

```
[07:58:31 PM] [/MyProject/MyListener.java] Added
[07:58:31 PM] [/MyProject/bin/MyListener.class] copied to project [MyProjectWeb]
[07:58:31 PM] Listener added to web.xml: MyListener
```

Tip: If you want to monitor the rapid deployment activity, run your rapid deployment session with console output. You can complete this by running the rapid deployment launch tool as follows:

- • ▶ `Windows` `wrd.bat -monitor`

- • ▶ `UNIX` `wrd.sh -monitor`

   b. A new listener entry is created in the Web deployment descriptor (found in the WebContent\WEB-INF\web.xml file in the Web module project). The following is an example of the entries in web.xml:

```
<listener>
 <display-name>MyListener</display-name>
 <listener-class>MyListener</listener-class>
</listener>
```

   c. The application is synchronized with the server.

3. To delete listeners from a free-form project, use your file management system to properly remove your listener artifacts from the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

4. The following activities occur when you remove a listener from the free-form project:

   a. The console output can look like this example when removing a listener Java source file (called *MyListener.java*) from a free-form project (called *MyProject*):

```
[08:00:23 PM] [/MyProject/MyListener.java] Deleted
[08:00:23 PM] [/MyProject/bin/MyListener.class] Deleted
[08:00:23 PM] Listener removed from web.xml: MyListener
```

   b. The application is synchronized with the server.

Adding and deleting JavaServer Pages to a free-form project:

You can add or delete JavaServer Pages (JSP) to or from a free-form project.

Steps

1. To add a JSP file to a free-form project, use your file management system to properly place your JSP artifacts into the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

2. The following activities occur when you drop a JSP into the free-form project:

   a. The JSP is mapped to the Web Content folder in the Web module project. The console output can look like this example when adding a JSP source file (called *MyJSP.jsp*) to a free-form project (called *MyProject*):

```
[08:15:23 PM] [/MyProject/MyJSP.jsp] Added
[08:15:23 PM] [/MyProject/MyJSP.jsp] coppied to project [MyProject]
[08:15:23 PM] JSP entry added to web.xml: MyJSP
[08:15:23 PM] Servlet mapping added.  URL is: [MyProjectWeb/MyJSP]
```

   Tip: If you want to monitor the rapid deployment activity, run your rapid deployment session with console output. You can complete this by running the rapid deployment launch tool as follows:

   - • ▶ `Windows` `wrd.bat -monitor`

   - • ▶ `UNIX` `wrd.sh -monitor`

   b. A new servlet entry and servlet mapping entry is created in the Web deployment descriptor (found in the WebContent\WEB-INF\web.xml file in the Web module project). The following is an example of the entries in web.xml:

```
<servlet>
 <servlet-name>MyJSP</servlet-name>
 <jsp-file>MyJSP.jsp</jsp-file>
</servlet>
<servlet-mapping>
 <servlet-name>MyJSP</servlet-name>
 <url-pattern>/MyJSP</url-pattern>
</servlet-mapping>
```

c. The application is synchronized with the server.

3. You can test the JSP running on the server, use the URL displayed in the console. In this example, the console output provided the following URL mapping information to the deployed JSP:

    ```
    [08:15:23 PM] Servlet mapping added.  URL is: [MyProjectWeb/MyJSP]
    ```

    Type `http://<machine_name>:<port>/MyProjectWeb/MyJSP` in the address bar and press **Enter**. Where *<machine_name>* is the host machine where WebSphere Application Server runs. By default, WebSphere Application Server runs on port 9080.

4. To delete JSP files from a free-form project, use your file management system to properly remove your JSP artifacts from the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

5. The following activities occur when you remove a JSP from the free-form project:

    a. The console output can look like this example when removing a JSP source file (called *MyJSP.jsp*) from a free-form project (called *MyProject*):

    ```
    [08:19:40 PM] [/MyProject/MyJSP.jsp] Deleted
    [08:19:40 PM] Servlet Mapping removed from web.xml /MyJSP
    [08:19:40 PM] JSP entry removed from web.xml MyJSP
    ```

    b. The application is synchronized with the server.

Adding and deleting static Web content to a free-form project:

You can add or delete static Web content to or from a free-form project.

Static content will maintain its directory structure. For example if /page1/index.html is placed in the free-form project, /page1/index.html will be copied into the Web module project, as a result, preserving the page1 folder in the Web module project.

Steps

1. To add static Web contents to a free-form project, use your file management system to properly place your static Web content artifacts into the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

2. The following activities occur when you drop a static Web content to a free-form project:

    a. The static Web content is mapped to the Web Content folder of the Web module project. The console output can look like this example when adding a static Web content file (called *MyStatic.html*) to a free-form project (called *MyProject*):

    ```
    [08:15:23 PM] [/MyProject/MyStatic.html] Added
    [08:15:23 PM] [/MyProject/MyStatic.html] copied to project [MyProjectWeb]
    ```

    Tip: If you want to monitor the rapid deployment activity, run your rapid deployment session with console output. You can complete this by running the rapid deployment launch tool as follows:

    - ▶ Windows `wrd.bat -monitor`

    - ▶ UNIX `wrd.sh -monitor`

    b. The application is synchronized with the server.

3. To delete static Web contents from a free-form project, use your file management system to properly remove your static Web content artifacts from the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

4. The following activities occurs when you remove a static Web content from a free-form project:

    a. The console output can look like this example when removing a static Web content (called *MyStatic.html*) from a free-form project (called *MyProject*):

    ```
    [08:19:40 PM] [/MyProject/MyStatic.html] Deleted
    ```

b. The application is synchronized with the server.

Adding and deleting enterprise beans to a free-form project

You can drop enterprise bean (EJB) artifacts or module files (ejb.jar) into a free-form project. The EJB artifacts can be annotated or non-annotated; they can contain source or compiled classes. The EJB artifacts include the EJB classes or the EJB interfaces (remote, local, home, and primary key class for entity beans).

Prerequisites

- Prior to dropping EJB module files (ejb.jar) into a free-form project, you need to generate the deployment code. The ejbDeploy tools do not automatically run on these already packaged ejb.jar files.

- If you drop an entity bean, by default the rapid deployment tools will generate a bean-managed persistence (BMP) entry in the deployment descriptor. If you want a container-managed persistence (CMP) bean, this needs to be specified using annotated source file.

- A naming convention must be followed for the Java source, compiled class, or annotated source files for each enterprise bean. For example, if an enterprise bean implementation class is named *MySession*Bean.java, follow this convention for any bean interfaces that are defined:

  – The home interface class must be named *MySession*Home.java

  – The remote interface class must be named *MySession*.java

  – The local interface class must be named *MySession*Local.java

  – The local home interface class must be named *MySession*LocalHome.java

  – The primary key class for entity beans can be named anything because the rapid deployment tools logically locates the correct class name by introspecting either the home or local home interface.

  These rules do not imply that both remote and local view types must be created; these are just naming conventions when creating source for a set of bean interfaces.

  A new EJB entry to the deployment descriptor is only added when a minimum set of resources are available in the free-form project. For example, if the bean class is placed in the free-form project, its compiled class maps to the EJB project but no deployment descriptor entry is created until the rapid deployment tools locate either a set of remote view type classes or local view type classes. For entity beans, a primary key class is also required. If the minimum resources are removed, the bean descriptor entry is removed. The examples section of this topic provides a concrete example of these rules.

Steps

1. Use your file management system to place your EJB artifacts properly into the free-form project. The directory name of the free-form project has the same string value you provided for the **-project** parameter used in the wrd-config command.

2. The following activities occur when you drop EJB artifacts into a free-form project:

   - The EJB interface classes are mapped into the imported classes folder of the EJB client JAR project.

   - The EJB classes are mapped into the imported classes folder of the EJB module project.

   - A new enterprise bean entry is created in the deployment descriptor (found in the ejbModule\META-INF\ejb-jar.xml file in the EJB module project).

   - A new enterprise bean binding is created in the bindings with a default JNDI name.

   - The application is synchronized with the server.

3. (Optional) Use the Universal Test Client to test your enterprise beans.

Example

This example shows the activities of the bean entry in the deployment descriptor when EJB resources are added and removed from the free-form project.

Tip: If you want to monitor rapid deployment activities, run your rapid deployment session with console output. Run the rapid deployment launch tool as follows:

- **Windows** `wrd.bat -monitor`

- **UNIX** `wrd.sh -monitor`

1. The console output can look like this when adding an enterprise bean Java source file (called *MyEJBBean.java*), remote interface Java source file (called *MyEJB.java*), and home interface (called *MyEJBHome.java*) to a free-form project (called *MyProject*):

```
[08:47:25 PM] [/MyProject/ejbs/MyEJB.java] Added
[08:47:25 PM] [/MyProject/ejbs/MyEJBBean.java] Added
[08:47:25 PM] [/MyProject/ejbs/MyEJBHome.java] Added
[08:47:30 PM] [/MyProject/bin/ejbs/MyEJB.class] copied to project [MyProjectEJBClient]
[08:47:30 PM] [/MyProject/bin/ejbs/MyEJBBean.class] copied to project [MyProjectEJB]
[08:47:30 PM] [/MyProject/bin/ejbs/MyEJBHome.class] copied to project [MyProjectEJBClient]
[08:47:32 PM] Enterprise bean added to ejb-jar.xml: MyEJB
[08:47:32 PM] Enterprise bean binding for [MyEJB] added to ejb-jar-bnd.xmi.  JNDI Name is: [ejb/MyEJBBean]
```

A new enterprise bean entry is created in the deployment descriptor (found in the ejbModule\META-INF\ejb-jar.xml file in the EJB module project):

```
<session id="Session_1092185252700">
 <ejb-name>MyEJB</ejb-name>
 <home>ejbs.MyEJBHome</home>
 <remote>ejbs.MyEJB</remote>
 <ejb-class>ejbs.MyEJBBean</ejb-class>
 <session-type>Stateless</session-type>
 <transaction-type>Container</transaction-type>
</session>
```

A new enterprise bean binding is created in the bindings with a default JNDI name. The ibm-ejb-bnd.xmi file might contain the following entries:

```
<ejbBindings xmi:id="EnterpriseBeanBinding_1092185252700" jndiName="ejb/MyEJBBean">
 <enterpriseBean xmi:type="ejb:Session" href="META-INF/ejb-jar.xml#Session_1092185252700"/>
</ejbBindings>
```

2. The console output can look like this example when adding a local interface (called *MyEJBLocal.java*) and local home interface (called *MyEJBLocalHome.java*) for the same bean in the same free-form project:

```
[08:52:59 PM] [/MyProject/ejbs/MyEJBLocal.java] Added
[08:52:59 PM] [/MyProject/ejbs/MyEJBLocalHome.java] Added
[08:52:59 PM] [/MyProject/bin/ejbs/MyEJBLocal.class] copied to project [MyProjectEJBClient]
[08:52:59 PM] [/MyProject/bin/ejbs/MyEJBLocalHome.class] copied to project [MyProjectEJBClient]
[08:53:01 PM] Local view type added for ejb: MyEJB
```

An update is made to the existing deployment descriptor (found in the ejbModule\META-INF\ejb-jar.xml file in the EJB module project):

```
<session id="Session_1092185252700">
 <ejb-name>MyEJB</ejb-name>
 <home>ejbs.MyEJBHome</home>
 <remote>ejbs.MyEJB</remote>
 <local-home>ejbs.MyEJBLocalHome</local-home>
 <local>ejbs.MyEJBLocal</local>
 <ejb-class>ejbs.MyEJBBean</ejb-class>
 <session-type>Stateless</session-type>
 <transaction-type>Container</transaction-type>
</session>
```

3. The console output can look like this example when the local interface (called *MyEJBLocal.java*) is removed from the same free-form project:

```
[08:56:27 PM] [/MyProject/bin/ejbs/MyEJBLocal.class] Deleted
[08:56:27 PM] [/MyProject/ejbs/MyEJBLocal.java] Deleted
[08:56:28 PM] Local view type removed ejb: MyEJB
```

Since the local interface was removed from the free-form project and the existing local home interface is not sufficient to define a local view type the bean deployment descriptor (found in the ejbModule\META-INF\ejb-jar.xml file in the EJB module project) is updated to reflect the removal of the local view type:

```
<session id="Session_1092185252700">
 <ejb-name>MyEJB</ejb-name>
 <home>ejbs.MyEJBHome</home>
 <remote>ejbs.MyEJB</remote>
 <ejb-class>ejbs.MyEJBBean</ejb-class>
 <session-type>Stateless</session-type>
 <transaction-type>Container</transaction-type>
</session>
```

4. The console output can look like this example when the remote interface (called *MyEJB.java*) is removed from the same free-form project:

```
[09:00:04 PM] [/MyProject/bin/ejbs/MyEJB.class] Deleted
[09:00:04 PM] [/MyProject/ejbs/MyEJB.java] Deleted
[09:00:09 PM] Enterprise bean removed from ejb-jar.xml: MyEJB
```

Since the remote interface has been removed from the free-form project, the existing home interface is not sufficient to define a remote view type. Both the remote and local view type are not defined, so the bean is removed from the deployment descriptor as well as its bindings.

Deploying Web services using a free-form project

You can add Java beans containing Web service annotation tags to a free-form project. This will generate a WSDL file and Web service deployment descriptors, and deploy the Web service.

Prerequisites: Before you can add the Java bean to the free-form project, the bean must contain the Web service annotation tags that define how the Web service will be deployed. The minimum required tag is `@WebSphere.WebService`, which tells the rapid deployment tools that you want to deploy a Web service. If you want to deploy a stateless session EJB Web service, the Java bean that you add to the free-form project must contain `@ejb.session` with `type=Stateless`. For more information on the available tags and how to add them to your beans, refer to the IBM WebSphere Application Server Toolkit online help.

To deploy a Web service from a Java bean using a free-form project:

1. Create a directory structure within your free-form project that maps exactly to the package name for the bean from which you are deploying the Web service. Note: If your project supports J2EE 1.3 (as compared to J2EE 1.4), the bean and its package structure must be contained in the `gen/src` folder or the free-form project. If it is located somewhere else the tools will create an SEI with the same name as the bean which in turn will result in a conflict.

2. Use your file management system to properly place your Java bean into the free-form project. The directory name of the free-form project has the same string value you had provided for the **-project** parameter used in the wrd-config command.

3. The following activities occur when you drop a bean containing the `@WebSphere.WebService` tag to a free-form project:

   a. The bean searched for annotation tags and is then compiled.

   b. The bean is analyzed and a Java2WSDL model is created. If any invalid or malformed tags are encountered the deployment process will abort at this point.

   c. The Service Endpoint Interface (SEI) code is generated and compiled.

   d. If you are deploying a stateless session EJB Web service, at this point a Web project is created or selected to act as the router project.

   e. Java2WSDL creates the WSDL file for the Web service.

   f. The Web service deployment descriptors, JAX-RPC mapping file, serializers, deserializers, helper and holder code are created. These files are then compiled.

g. If you are deploying a stateless session EJB Web service, at this point the EJB is deployed and the deployment files are compiled.

The Web service endpoint can be accessed with the following URL: `http://<hostname>:<WC_defaulthost>/<free-form project name>Web/services/<port-component-name>`, and the `<port-component-name>` element can be found in the generated `webservices.xml` in the Web project.

For example, if your free-form project is called `MyProject`, the port-component-name is `MyService`. If you have published the application to `myhost` using the default ports, then the endpoint URL will be: `http://myhost:9080/MyProjectWeb/services/MyService`

Tip: If you want to monitor the rapid deployment activity, run your rapid deployment session with console output. You can complete this by running the rapid deployment launch tool as follows:

- ▶ `Windows` `wrd.bat -monitor`
- ▶ `UNIX` `wrd.sh -monitor`

Restriction: There is no support for deleting Web services deployed by a free-form project at this time.

Automatically installing applications on WebSphere Application Server v6.x

The *automatic installation* configuration creates a single monitored directory that listens for fully composed EAR files or application module such as WAR, EJB JAR files, or standalone resource adapter archive (RAR) files.

If you place EAR files inside this monitored directory, the EAR file is automatically deployed to the server. If you delete the EAR file from the monitored directory, that application is uninstalled from the server. If you place WAR or EJB jar files in the monitored directory, the rapid deployment tool generates the necessary EAR wrapper and then publishes that EAR file on the server. For RAR files, a wrapper is not created; the standalone RAR files are published to the server.

Restrictions
- The rapid deployment tools do not automatically configure server-specific functions for an application. You need to create and manage the server-specific configurations through the WebSphere administrative console. Examples of server-specific configurations include setting up ports, security role mappings or data sources.
- If you did not generate deployment code for the enterprise beans or enterprise application containing enterprise beans, the rapid deployment tool generates the deployment code with the default backend option set to DB2UDB_V82. If you want to set a different backend option, run the EJB deployment tools (ejbdeploy.bat) on the EAR file before dropping it into the automatic installation project.

To automate the installation of applications on WebSphere Application Server v6.x:

Setting up a rapid deployment environment

You can run the rapid deployment batch tools from a command line to create, configure, and launch rapid deployment projects using the IBM WebSphere v6.x runtime environment. The rapid deployment tools run in a non-graphical command line interface.

Prerequisites
- WebSphere Application Server v6.x installed
- The rapid deployment launch tool is in the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:

- – | Windows | wrd.bat
- – | UNIX | wrd.sh
- The rapid deployment configuration tool is in the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:
  - – | Windows | wrd-config.bat
  - – | UNIX | wrd-config.sh

To set up a rapid deployment workspace:

1. Open a command prompt.
2. Specify the location on your file system that will host the rapid deployment activities. This location serves as your workspace root, and all your rapid deployment projects reside here. Define an environment variable called WORKSPACE:

   - | Windows | `set WORKSPACE=<workspace root>`
   - | UNIX | `export WORKSPACE=<workspace root>`

   Configure a rapid deployment workspace for each rapid deployment project, rather than configuring multiple rapid deployment projects inside a single rapid deployment workspace.

   Tip: To list all your environment variables and their values, run the command:

   - | Windows | `set "`
   - | Linux | `export -n`
   - | AIX | | Solaris | | HP-UX | `export`

After the WORKSPACE environment variable is defined, you can continue to configure and launch your rapid deployment workspace.

Creating an automatic installation project in your workspace

The *automatic installation* project automates the installation, modification, and uninstallation of J2EE compliant applications (EAR files) or modules (WAR, JAR, or RAR files) on a server. Automatic application installation releases you from the complexity of administrating your application on a server.

The initial configuration of the automatic installation project in your workspace creates an automatic installation project. After you have configured the workspace to contain an automatic installation project, you can launch a rapid deployment session. Launching a rapid deployment session indicates that you want the project to start listening for your EAR or module files. If you place an EAR file in the automatic application installation project, the EAR file is automatically deployed to the server. If you delete EAR files from the project, the application is uninstalled from the server. If you place a new copy of the same EAR file in the automatic application installation project, the application is either reinstalled or the server restarts to pick up the modifications. If you place module files in the project, a supporting EAR wrapper is generated, and then that EAR file is deployed to the server.

Prerequisite

- Set up a rapid deployment environment.

To configure a new automatic installation workspace:

1. From the command prompt, change to the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server is installed.
2. Type the following at the prompt:

   - | Windows | `WRD-config.bat -project "project_name" -style "autoappinstall"`

- > UNIX WRD-config.sh -project *"project_name"* -style *"autoappinstall"*

3. In the **Parameter Configuration Settings** dialog box, specify the value for one or more of the following settings:

Option	Description
serverName	The name of the server process you want to publish your application. For example, server1. For WebSphere Application Server Network Deployment, the server name is in the form *<cell name>/<node name>/<server name>*. To publish to a cluster, the server name is in the form *<cell name>/<cluster name>*
username	(Optional) If security is enabled, specify the user name for current active authentication settings defined in the server configuration.
password	(Optional) If security is enabled, specify the password for current active authentication settings defined in the server configuration.
serverJMXHost	The host name of the machine containing the server you want to make a connection to. For example, localhost. For WebSphere Application Server Network Deployment, type the host name of the Network Deployment Manager.
serverJMXPort	The server administrative port number, also known as the Simple Object Access Protocal (SOAP) connector port. This port is used for making JMX connections with the server. For example, 8880. For WebSphere Application Server Network Deployment, type the SOAP port number of the Network Deployment Manager, default is 8879.

The automatic installation project is created with the naming convention *project_name*, where *project_name* is the string value provided for the **-project** parameter.

Starting a rapid deployment session

You can launch a rapid deployment session to start listening for application artifacts that are dropped into the folder you configured for either the free-form or automatic installation approach. Use the **wrd** command to launch a rapid deployment session. If you want to monitor rapid deployment activities, you can specify console output.

Prerequisites
- Set up a rapid deployment environment.
- Configure the behavior of the monitored directory as free-form or automatic installation.

Steps
1. From the command prompt, change to the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server is installed.
2. Verify that the server on which you want to deploy your application is running.

   Tip: If there is no server connection, you get the following console output:

   ```
   [06:52:28 PM] Publishing MyProjectApp to server_510658053
   [06:52:28 PM] Failed to make connection to Websphere Application Server.
   ```
3. At the command prompt, type the appropriate command:
   - (Recommended) To monitor rapid deployment activities:
     - > Windows

- – `UNIX` `wrd.sh -monitor`
- To launch a rapid deployment session:
  - – `Windows` `wrd.bat`
  - – `UNIX` `wrd.sh`
- To use batch mode to run a full build of the workspace and then shut down the process:
  - – `Windows` `wrd.bat -project "project_name" -batch`
  - – `UNIX` `wrd.sh -project "project_name" -batch`

After you launch the rapid deployment tools, you can create or drop your J2EE artifacts into the monitored directory. The application generated is deployed and updated on the target server.

Managing EAR or module files in the automatic installation project

You can place J2EE compliant applications (EAR files) or application modules such as WAR files, EJB JAR files, or standalone RAR files in an automatic application installation project and let rapid deployment tools deploy that application on to a target server.

From a J2EE development perspective, the automatic application installation project supports a single J2EE application and at most one of each of the application modules (WAR, EJB jars, or stand-alone RAR file).

Prerequisites

- Set up a rapid deployment environment.
- Created an automatic application installation project in your workspace.
- Verify that the server where you want to deploy your application is running.
- Launch a rapid deployment session by using the wrd command.

Use your file management system to place your EAR or module files properly in the automatic installation project. The directory name of the automatic installation project has the same string value you provided for the **-project** parameter used in the wrd-config command.

The following actions are supported when using the automatic installation projects:

Dropping EAR files

You can place EAR files in the automatic application installation project. The EAR file is automatically deployed to the server by expanding into the installedApps folder of the local WebSphere Application Server.

Restriction: If you did not generate deployment code for the enterprise beans contained in the enterprise application, the rapid deployment tool generates the deployment code with the default backend option set to DB2UDB_V82 during the installation of your application on the server. If you want to set a different backend option, run the EJB deployment tools (ejbdeploy.bat) on the EAR file before dropping it into the automatic installation project.

Deleting EAR files

When you delete EAR files from the automatic installation project, the application is uninstalled from the server.

Modifying EAR files

You can place a modified version of the same EAR file in the automatic installation project and the application is reinstalled.

Dropping module files

You can place WAR or EJB jar files in the automatic installation project. A supporting EAR wrapper is generated in the **gen** subdirectory, and then that EAR file is deployed to the server by expanding into the **installedApps** folder of the local WebSphere Application Server. For RAR files, a wrapper is not created. The standalone RAR files are published to the server.

Restriction: If you did not generate deployment code for the enterprise beans, the rapid deployment tool will generate the deployment code with the default backend option set to DB2UDB_V82. If you want to set a different backend option, run the EJB deployment tools (ejbdeploy.bat) on the EAR file prior to dropping it into the automatic installation project.

Deleting module files

When you delete module files from the automatic application installation project, the application is uninstalled from the server. In addition, any supporting EAR wrapper that was generated in the **gen** subdirectory is removed.

Modifying module files

You can place a modified version of the same module file in an automatic application installation project, then the application is reinstalled. In addition, any supporting EAR wrapper that was generated in the **gen** subdirectory is modified.

References

The rapid deployment configuration tool reference

You can use the rapid deployment configuration tool (wrd-config.bat or wrd-config.sh) from a command line to configure rapid deployment projects, such as free-form or automatic installation projects. The **wrd-config** command is run in a non-graphical command line interface.

Requirements

- WebSphere Application Server v6.x installed. The **wrd-config** command supports only WebSphere Application Server v6.x as the target server.
- The rapid deployment launch tool, is in the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:

  - Windows   wrd.bat

  - UNIX   wrd.sh

- The rapid deployment configuration tool, is in the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:

  - Windows   wrd-config.bat

  - UNIX   wrd-config.sh

.

Before using the wrd-config command you need to set up a workspace environment variable. For more information, see the related task about setting up a rapid deployment environment. After you have set up your command line, you can use the following syntax and arguments to run the tool.

Syntax

`Windows`   `wrd-config.bat -project <"project_name"> -style <"freeform"|"autoappinstall">` `[<optional parameters>]`

`UNIX`   `wrd-config.sh -project <"project_name"> -style <"freeform"|"autoappinstall">` `[<optional parameters>]`

Mandatory parameters:

-project <"*project_name*">
> The name of the rapid deployment project that you want to create. The project name needs to be unique in your workspace.

-style <"`freeform`"|"`autoappinstall`">
> The deployment approach - either the free-form or automatic application installation project is used as the deployment style.
>
> The free-form project ("`freeform`") allows you to create or drop in your J2EE artifacts (such as servlet source, JSP, static Web content, EJB classes or source files, Java class files, and all other generic files) into the free-form project. These resources are automatically placed in the appropriate location in the J2EE project structure.
>
> The automatic installation project ("`autoappinstall`") creates a single project that listens for fully composed EAR or module files. If EAR files are placed inside this project, the EAR file is automatically deployed to the server. If the EAR file is deleted, then that application is uninstalled from the server. If you place WAR or EJB jar files in a automatic installation project, the rapid deployment tool generates the necessary EAR wrapper and then publishes that EAR file on the server. For RAR files, a wrapper is not created. The standalone RAR files are published to the server.

Optional parameters:

-workspace
> Specifies the location on your file system that will host the rapid deployment activities.

-rebuild
> Cleans and rebuilds the contents of the rapid deployment project.

-configure
> Opens an interactive console session to modify any available parameters.

-runtime <"`was51`"|"`was60`" | "`was61`">
> Targeting your runtime will configure which JRE library to use, based on the runtime location, and configure the project's classpath to contain the WebSphere runtime libraries. Specify as an identifier if either WebSphere Application Server v5.1 ("`was51`"), WebSphere Application Server v6.0 ("`was60`"), or WebSphere Application Server v6.1 ("`was61`") is used as the target runtime environment for rapid deployment processing. If this parameter is not specified, the default setting is WebSphere Application Server v6.1. This parameter is only used with **-runtimePath** parameter.

-runtimePath <"*x:\WebSphere\AppServer*">
> Where *x:\WebSphere\AppServer* is the directory where WebSphere Application Server is installed. This runtimePath parameter specify the directory path to where the target runtime is installed and is used in conjunction with **-runtime** parameter.

-j2eeVersion <"`1.3`"|"`1.4`">
> Specifies either J2EE version 1.3 or 1.4 is used for development and deployment.

-configPath "*x:\filename.xml*"
> The destination file path for the configuration file, where *x* is the temporary directory. This file

persists the configuration data to an XML file, which can later be used to drive other rapid deployment configuration sessions. If this path is not specified, the default location is in the root of the rapid deployment workspace.

-configData *"x:\filename.xml"*

The path of an existing XML configuration file that is used to drive the configuration session. If this path is not specified, the default location of the XML configuration file is in the root of the rapid deployment workspace with the naming convention *projectName_*headlessconfig.xml, where *projectName* is the value specified in the **-project** parameter.

-listStyles

The available deployment styles and their descriptions.

-listServers

The available runtime server targets.

-properties

Lists the properties for a given deployment project. This parameter is only used with **-project**.

-buildMode

Specify to disable all console outputs. You find this useful for silent builds.

-usage

Displays the optional and required parameters for this command.

Examples of running the WRD-config command

To create a new free-form project:

```
WRD-config.bat -project "MyProject" -style "freeform" -runtime "was61" -runtimePath "c:\WebSphere\AppServer"
```

To create a new automatic application installation project:

```
WRD-config.bat -project "MyProject" -style "autoappinstall"
```

To persist configuration data to an XML file:WRD-config.bat -project *"MyProject"* -style *"freeform"* -runtime *"was61"* -runtimePath *"c:\WebSphere\AppServer"* -configPath *"c:\configData\myHeadlessConfig.xml"*

To create a new rapid deployment project using an existing XML configuration file:

```
WRD-config.bat -configData "c:\configData\myHeadlessConfig.xml"
```

To clean and rebuild an existing rapid deployment project:

```
WRD-config.bat -project "MyProject" -rebuild
```

To modify available deployment parameters:

```
WRD-config.bat -project "MyProject" -configure
```

To query properties of an existing rapid deployment project:

```
WRD-config.bat -project "MyProject" -properties
```

To query available rapid deployment styles and runtime targets:

```
WRD-config.bat -listStyles -listServers
```

The rapid deployment launch tool reference

You can use the rapid deployment launch tool (wrd.bat or wrd.sh) to specify that you want the tool to start listening for application artifacts that are dropped into the monitored project. The **wrd** command is run in a non-graphical command line interface.

Requirements

- WebSphere Application Server v6.x installed. The **wrd** command support only WebSphere Application Server v6.x as the target server.
- The rapid deployment launch tool is in the *x*:/profiles/*<profileName>*/bin directory, where *x* is the directory where WebSphere Application Server v6.x is installed:

    – ▶ Windows wrd.bat

    – ▶ UNIX wrd.sh

- The rapid deployment configuration tool is in the *x*:/profiles/*<profileName>*/bin directory where *x* is the directory where WebSphere Application Server v6.x is installed:

    – ▶ Windows wrd-config.bat

    – ▶ UNIX wrd-config.sh

Before using the **wrd-config** command you need to set up a workspace environment variable. For more information, see the related task about setting up a rapid deployment environment. After you have set up your command line, you can use the following syntax and arguments to run the tool.

Syntax

▶ Windows `wrd.bat [<optional parameters>]`

▶ UNIX `wrd.sh [<optional parameters>]`

Optional parameters

-workspace
> Specifies the location on your file system that will host the rapid deployment activities.

-monitor
> Enables console feedback from the rapid deployment tool.

-project <*"project_name"*>
> The name of the rapid deployment project that you want to target to run in batch mode. This **-project** parameter is only used in conjunction with **-batch** parameter.

-batch Batch mode runs a full build on the projects generated in the rapid deployment workspace and then shutdowns the process. A project, specified by the **-project**, is required when using the batch mode parameter to generate the rapid deployment projects that will be compiled during the full build.

-usage
> Displays the optional parameters for this command.

Chapter 6. Debugging applications

To debug your application, you must use a development environment like Application Server Toolkit or Rational Application Developer to create a Java project. You must then import the program that you want to debug into the project. By following the steps below, you can import the WebSphere Application Server examples into a Java project.

Two debugging styles are available:
- **Step-by-step** debugging mode prompts you whenever the server calls a method on a Web object. A dialog lets you step into the method or skip it. In the dialog, you can turn off step-by-step mode when you are finished using it.
- **Breakpoints** debugging mode lets you debug specific parts of programs. Add breakpoints to the part of the code that you must debug and run the program until one of the breakpoints is encountered.

Breakpoints actually work with both styles of debugging. Step-by-step mode just lets you see which Web objects are being called without having to set up breakpoints ahead of time.

You do not need to import an entire program into your project. However, if you do not import all of your program into the project, some of the source might not compile. You can still debug the project. Most features of the debugger work, including breakpoints, stepping, and viewing and modifying variables. You must import any source that you want to set breakpoints in.

The inspect and display features in the source view do not work if the source has build errors. These features let you select an expression in the source view and evaluate it.

1. Create a Java Project by opening the New Project dialog.
2. Select **Java** from the left side of the dialog and **Java Project** in the right side of the dialog.
3. Click **Next** and specify a name for the project, for example, WASExamples.
4. Click **Finish** to create the project.
5. Select the new project, choose **File > Import > File System**, then **Next** to open the import file system dialog.
6. Browse the directory for files.

   Go to the following directory: *app_server_root*/installedApps/*node_name*/DefaultApplication.ear/ DefaultWebApplication.war.
7. Select DefaultWebApplication.war in the left side of the Import dialog and then click **Finish**. This imports the JavaServer Pages files and Java source for the examples into your project.
8. Add any JAR files needed to build to the Java Build Path.

   Select **Properties** from the right-click menu. Choose the Java Build Path node and then select the Libraries tab. Click **Add External JARs** to add the following JAR files:
   - *app_server_root*/installedApps/*node_name*/DefaultApplication.ear/Increment.jar.

     When you have added this JAR file, select it and use the **Attach Source** function to attach the Increment.jar file because it contains both the source and class files.
   - *app_server_root*/lib/j2ee.jar
   - *app_server_root*/lib/pagelist.jar
   - *app_server_root*/lib/webcontainer.jar

   Click **OK** when you have added all of the JARs.
9. You can set some breakpoints in the source at this time if you like, however, it is not necessary as step-by-step mode will prompt you whenever the server calls a method on a Web object.
   Step-by-step mode is explained in more detail below.
10. To start debugging, you need to start the WebSphere Application Server in debug mode and make note of the JVM debug port. The default value of the JVM debug port is 7777.

11. When the server is started, switch to the debug perspective by selecting **Window > Open Perspective > Debug**. You can also enable the debug launch in the Java Perspective by choosing **Window > Customize Perspective** and selecting the **Debug** and **Launch** checkboxes in the **Other** category.

12. Select the workbench toolbar **Debug** pushbutton and then select **WebSphere Application Server Debug** from the list of launch configurations. Click the **New** pushbutton to create a new configuration.

13. Give your configuration a name and select the project to debug (your new WASExamples project). Change the port number if you did not start the server on the default port (7777).

14. Click **Debug** to start debugging.

15. Load one of the examples in your browser. For example: `http://your.server.name:9080/hitcount`

To learn more about debugging, launch the Application Server Toolkit, select **Help > Help Contents** and choose the **Debugger Guide bookshelf** entry. To learn about known limitations and problems that are associated with the Application Server Toolkit, see the Application Server Toolkit release notes. 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 Must gather documents page for information to gather to send to IBM Support.

Debugging components in the Application Server Toolkit

The Application Server Toolkit, included with the WebSphere Application Server on a separately-installable CD, includes debugging functionality that is built on the Eclipse workbench. Documentation for the Application Server Toolkit is provided with that product. To learn more about the debug components, launch the Application Server Toolkit, select **Help > Help Contents** and choose the **Debugger Guide bookshelf** entry.

The Application Server Toolkit includes the following:

The WebSphere Application Server debug adapter
which allows you to debug Web objects that are running on WebSphere Application Server and that you have launched in a browser. These objects include enterprise beans, JavaServer Pages files, and servlets.

The JavaScript debug adapter
which enables server-side JavaScript debugging.

The Compiled language debugger
which allows you to detect and diagnose errors in compiled-language applications.

The Java development tools (JDT) debugger
which allows you to debug Java code.

All of the debug components in the Application Server Toolkit can be used for debugging locally and for remote debugging. To learn more about the debug components, launch the Application Server Toolkit, select **Help > Help Contents** and choose the **Debugger Guide bookshelf** entry.

To learn more about Log and Trace Analyzer, launch the Application Server Toolkit, and select **Help > Help Contents**. To learn about known limitations and problems that are associated with the Application Server Toolkit, see the Application Server Toolkit release notes.

Chapter 7. Assembling applications

Application assembly consists of creating Java 2 Platform, Enterprise Edition (J2EE) modules that can be deployed onto application servers. The modules are created from code artifacts such as Web application archives (WAR files), resource adapter archives (RAR files), enterprise bean (EJB) JAR files, and application client archives (JAR files). 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.

This topic assumes that you have developed code artifacts that you want to deploy onto an application server and have unit tested the code artifacts in your favorite integrated development environment. Code artifacts that you might assemble into deployable J2EE modules include the following:
- Enterprise beans
- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (*connector*) implementations
- Application clients
- Session Initiation Protocol (SIP) modules (SAR files)
- Other supporting classes and files

Before you can assemble your code artifacts into deployable J2EE modules, you must install or get access to a supported assembly tool. WebSphere Application Server supports two tools that you can use to develop, assemble, and deploy J2EE modules:
- Application Server Toolkit (AST)
- Rational Application Developer

You assemble code artifacts into J2EE modules in order to deploy the code artifacts onto an application server. When you assemble code artifacts, you package and configure the code artifacts into deployable J2EE applications and modules, edit deployment descriptors, and map databases as needed. Unless you assemble your code artifacts into J2EE modules, you cannot run them successfully on an application server.

This topic describes how to assemble J2EE code artifacts into deployable modules using an assembly tool. Alternatively, you can use a WebSphere rapid deployment tool to quickly assemble and deploy J2EE code artifacts. Refer to Rapid deployment of J2EE applications for details.

1. Start an assembly tool.
2. **Optional:** Read the online documentation for the assembly tool.
   - Click **Help > Help Contents >** *product_name* **information**, for example **Help > Help Contents > Application Server Toolkit information**. The displayed documentation provides extensive information on assembling modules.
   - Click **Help > Cheat Sheets >** *tutorial_name* **> OK**. The displayed tutorial provides steps with illustrations.
   - Press **F1** to access information specific to an AST or Rational Application Developer view or window.
   - Visit the **Application Server Toolkit** information center that accompanies this WebSphere Application Server information center. Also, refer to articles on **Rapid deployment of J2EE applications** in this information center.
   - See the topic "Assembling applications: Resources for learning" on page 1348 for additional sources.
3. Configure the assembly tool for work on J2EE modules.
4. Migrate J2EE projects or code artifacts created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool.

   To migrate files, use the J2EE Migration wizard or import the files to AST or Rational Application Developer.

5. Create an enterprise application project to which you can add archive files. You can create an enterprise application project separately or when you create archive files such as the following:

   - Create a Web project.
   - Create an enterprise bean (EJB) project.
   - Create an application client.
   - Create a resource adapter (connector) project.

6. Edit the deployment descriptors as needed. You can edit deployment descriptors for enterprise application, Web, application client, and enterprise bean (EJB) modules.

   Topics on deployment descriptor editors such as Application Deployment Descriptor editor in AST documentation provide extensive information on editing deployment descriptors.

7. **Optional:** Generate enterprise bean (EJB) to relational database (RDB) mappings for EJB modules.

8. Verify the archive files.

9. Generate code for deployment for Web services-enabled modules or for enterprise applications that use Web service modules.

After assembling your applications, use a systems management tool to deploy the EAR or WAR files onto the application server. "Ways to install applications or modules" on page 1366 lists systems management tools available for deploying J2EE modules on an application server. The systems management tool follows the security and deployment instructions defined in the deployment descriptor, and enables you to modify bindings specified within an assembly tool. The tool locates the required external resources that the application uses, such as enterprise beans and databases.

To deploy EJB projects to a target server, right-click the EJB project in the Project Explorer view and click **Deploy**.

Package your application so that the `.ear` file contains necessary modules only. Modules can include metadata for the modules such as information on deployment descriptors, bindings, and IBM extensions.

Use the administrative console at installation to complete the security instructions defined in the deployment descriptor and to locate required external resources, such as enterprise beans and databases. You can add configuration properties and redefine binding properties defined in an assembly tool.

After installation, you can view module deployment descriptors using the console.

Application assembly and J2EE applications

Application assembly is the process of creating an enterprise archive (EAR) file containing all files related to an application, as well as an XML deployment descriptor for the application. This configuration and packaging prepares the application for deployment onto an application server.

EAR files are comprised of the following archives:
- Enterprise bean JAR files (known as EJB modules)
- Web archive (WAR) files (known as Web modules)
- Application client JAR files (known as client modules)
- Resource adapter archive (RAR) files (known as resource adapter modules)
- SAR files (known as Session Initiation Protocol (SIP) modules)

Ensure that modules are contained in an EAR file so that they can be deployed onto the server. The exceptions are WAR modules, which you can deploy individually. Although WAR modules can contain regular JAR files, they cannot contain the other module types described previously.

The assembly process includes the following actions:
- Selecting all of the files to include in the module.

- Creating a deployment descriptor containing instructions for module deployment on the application server.

  As you configure properties using an assembly tool, the tool generates the deployment descriptor for you. While the Application Server Toolkit (AST) or Rational Application Developer graphical interface is recommended, you can also edit descriptors directly in your favorite XML editor.
- Packaging modules into a single EAR file, which contains one or more files in a compressed format.

As part of the assembly process, you might also set environment-specific binding information. These bindings are defaults for an administrator to use when installing the application through the administrative console. Further, you might define IBM extensions to the J2EE specification, such as to allow servlets to be served by class name. To ensure portability to other application servers, these extensions are saved in an XML file that is separate from the standard J2EE deployment descriptor.

Assembly tools

WebSphere Application Server supports two tools that you can use to develop, assemble, and deploy J2EE modules: Application Server Toolkit (AST) and Rational Application Developer. These tools are referred to in this information center as the *assembly tools*.

The AST is available in your WebSphere Application Server CD-ROM package. Rational Application Developer is available only on a trial basis in the WebSphere Application Server CD-ROM package.

The assembly feature of the AST and Rational Application Developer products runs on Windows and Linux Intel platforms. Users of WebSphere Application Server on other platforms must assemble their modules using an assembly tool installed on Windows or Linux Intel platforms. To install an assembly tool, follow instructions available with the tool.

Although this information center refers to the AST and Rational Application Developer products as the *assembly tools*, you can use the products to do more than assemble modules. Rational Application Developer is an integrated development environment that provides development, testing, assembly and deployment capabilities. However, topics on application assembly in this information center focus on assembling J2EE modules using the J2EE Perspective of the assembly tools. Each assembly tool provides extensive online documentation; the topics on application assembly in this information center supplement that documentation. The **Application Server Toolkit** information center is available with this information center.

Generating code for Web service deployment

Before deploying Web services-enabled modules or any enterprise application archive (EAR) files that contain Web services-enabled module onto an application server, you must generate deployment code for the application.

This article assumes you have assembled a module enabled with Web services, added it to an application, saved the application, and verified the application. It also assumes that you have started and configured an assembly tool.

You can use an assembly tool to generate deployment code for the Web services-enabled module or for the EAR file that contains the Web services-enabled module.

1. If you have turned automatic validation off, manually validate any modules that use Web services with the JSR109 Web services validator before generating deployment code for them. If validating your module results in compilation errors or validation errors, fix the errors before generating deployment code. However, if validating your module results in warning or information messages, you can generate deployment code.

2. In the Project Explorer view of the assembly tool, right-click on the Web services-enabled module (WAR, enterprise bean JAR, or application client JAR file) for which you want to generate code for deployment.

3. Click **Deploy**. Alternatively, you can generate deployment code for Web services-enabled modules using the deployment tool for Web services (`wsdeploy`) from a command prompt.

4. If messages indicate that automatic file overwriting is not enabled, click **Yes to All** so the generated files are added to the module.

5. If errors such as *Unbound classpath variable: WAS_50_PLUGINDIR* appear in the Tasks list, change the Java build path libraries properties to define that variable to be the WebSphere Application Server installation directory.

Code is generated into the folder where your Web services-enable module is located. Problems with the generation of code result in a window that displays error messages.

Install the Java 2 Platform, Enterprise Edition (J2EE) application on your server machine. You can install the application onto a server using the administrative console. Before installing the application, you might need to set class paths.

Assembling applications: Resources for learning

Additional information and guidance on assembling applications is available on various Internet sites.

Use the following links to find relevant supplemental information about the application assembly and using an assembly tool. 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.

Refer to "Web resources for learning" on page 15 for links to information applicable to WebSphere Application Server generally, such as lists of IBM technical papers, Redbooks and samples.

View links to additional information about:
- "Programming instructions and examples"
- "Programming specifications"
- "Administration" on page 1349

Programming instructions and examples
- Rational developer community
- *IBM WebSphere Developer Technical Journal:* Using Rational Developer to create a simple Web service and use it in a Web application
- The J2EE^{TM} Tutorial
- Java 2 Enterprise Edition: Books index

Programming specifications
- J2EE 1.4 specification
- EJB specifications
- Servlet specifications
- Connector RAR files

Administration

- WebSphere Version 6 Web Services Handbook Development and Deployment
- WebSphere Application Server V6 Migration Guide
- WebSphere Version 6 Web Services Handbook Development and Deployment
- Listing of all IBM WebSphere Application Server Redbooks

Chapter 8. Class loading

Class loaders are part of the Java virtual machine (JVM) code and are responsible for finding and loading class files. Class loaders enable applications that are deployed on servers to access repositories of available classes and resources. Application developers and deployers must consider the location of class and resource files, and the class loaders used to access those files, to make the files available to deployed applications. Class loaders affect the packaging of applications and the runtime behavior of packaged applications of deployed applications.

This topic describes how to configure class loaders for application files or modules that are installed on an application server.

To better understand class loaders in WebSphere Application Server, read "Class loaders." The topic "Class loading: Resources for learning" on page 1360 refers to additional sources.

Configure class loaders for application files or modules that are installed on an application server using the administrative console. You configure class loaders to ensure that deployed application files and modules can access the classes and resources that they need to run successfully.

1. If an installed application module uses a resource, create a resource provider that specifies the directory name of the resource drivers.

   Do not specify the resource Java archive (JAR) file names. All JAR files in the specified directory are added into the class path of the WebSphere Application Server extensions class loader. If a resource driver requires a native library (`.dll` or `.so` file), specify the name of the directory that contains the library in the native path of the resource configuration.

2. Specify class-loader values for an application server.

3. Specify class-loader values for an installed enterprise application.

4. Specify the class-loader mode for an installed Web module.

5. If your deployed application uses shared library files, associate the shared library files with your application. Use a library reference to associate a shared library file with your application.

   a. If you have not done so already, define a shared library instance for each library file that your applications need.

   b. Define a library reference instance for each shared library that your application uses.

After configuring class loaders, ensure that your application performs as desired. To diagnose and fix problems with class loaders, refer to Troubleshooting class loaders.

Class loaders

Class loaders find and load class files. Class loaders enable applications that are deployed on servers to access repositories of available classes and resources. Application developers and deployers must consider the location of class and resource files, and the class loaders used to access those files, to make the files available to deployed applications.

This topic provides the following information about class loaders in WebSphere Application Server:
- "Class loaders used and the order of use"
- "Class-loader isolation policies" on page 1353
- "Class-loader modes" on page 1355

Class loaders used and the order of use

The runtime environment of WebSphere Application Server uses the following class loaders to find and load new classes for an application in the following order:

1351

1. The bootstrap, extensions, and CLASSPATH class loaders created by the Java virtual machine

   The bootstrap class loader uses the boot class path (typically classes in `jre/lib`) to find and load classes. The extensions class loader uses the system property java.ext.dirs (typically `jre/lib/ext`) to find and load classes. The CLASSPATH class loader uses the CLASSPATH environment variable to find and load classes.

   The CLASSPATH class loader loads the Java 2 Platform, Enterprise Edition (J2EE) application programming interfaces (APIs) provided by the WebSphere Application Server product in the `j2ee.jar` file. Because this class loader loads the J2EE APIs, you can add libraries that depend on the J2EE APIs to the class path system property to extend a server class path. However, a preferred method of extending a server class path is to add a shared library.

2. A WebSphere extensions class loader

   The WebSphere extensions class loader loads the WebSphere Application Server classes that are required at run time. The extensions class loader uses a ws.ext.dirs system property to determine the path that is used to load classes. Each directory in the `ws.ext.dirs` class path and every Java archive (JAR) file or ZIP file in these directories is added to the class path used by this class loader.

   The WebSphere extensions class loader also loads resource provider classes into a server if an application module installed on the server refers to a resource that is associated with the provider and if the provider specifies the directory name of the resource drivers.

3. One or more application module class loaders that load elements of enterprise applications running in the server

   The application elements can be Web modules, enterprise bean (EJB) modules, resource adapter archives (RAR files), and dependency JAR files. Application class loaders follow J2EE class-loading rules to load classes and JAR files from an enterprise application. WebSphere Application Server enables you to associate shared libraries with an application.

4. Zero or more Web module class loaders

   By default, Web module class loaders load the contents of the `WEB-INF/classes` and `WEB-INF/lib` directories. Web module class loaders are children of application class loaders. You can specify that an application class loader load the contents of a Web module rather than the Web module class loader.

Each class loader is a child of the previous class loader. That is, the application module class loaders are children of the WebSphere extensions class loader, which is a child of the CLASSPATH Java class loader. Whenever a class needs to be loaded, the class loader usually delegates the request to its parent class loader. If none of the parent class loaders can find the class, the original class loader attempts to load the class. Requests can only go to a parent class loader; they cannot go to a child class loader. If the WebSphere extensions class loader is requested to find a class in a J2EE module, it cannot go to the application module class loader to find that class and a ClassNotFoundException error occurs. After a class is loaded by a class loader, any new classes that it tries to load reuse the same class loader or go up the precedence list until the class is found.

Class-loader isolation policies

The number and function of the application module class loaders depend on the class-loader policies that are specified in the server configuration. Class loaders provide multiple options for isolating applications and modules to enable different application packaging schemes to run on an application server.

Two class-loader policies control the isolation of applications and modules:

Class-loader policy	Description
Application	Application class loaders load EJB modules, dependency JAR files, embedded resource adapters, and application-scoped shared libraries. Depending on the application class-loader policy, an application class loader can be shared by multiple applications (Single) or unique for each application (Multiple). The application class-loader policy controls the isolation of applications that are running in the system. When set to `Single`, applications are not isolated. When set to `Multiple`, applications are isolated from each other.
WAR	By default, Web module class loaders load the contents of the `WEB-INF/classes` and `WEB-INF/lib` directories. The application class loader is the parent of the Web module class loader. You can change the default behavior by changing the Web application archive (WAR) class-loader policy of the application. The WAR class-loader policy controls the isolation of Web modules. If this policy is set to `Application`, then the Web module contents also are loaded by the application class loader (in addition to the EJB files, RAR files, dependency JAR files, and shared libraries). If the policy is set to `Module`, then each Web module receives its own class loader whose parent is the application class loader. **Tip:** The console and the underlying `deployment.xml` file use different names for WAR class-loader policy values. In the console, the WAR class-loader policy values are `Application` or `Module`. However, in the underlying `deployment.xml` file where the policy is set, the WAR class-loader policy values are `Single` instead of `Application`, or `Multiple` instead of `Module`. `Application` is the same as `Single`, and `Module` is the same as `Multiple`.

Note: WebSphere Application Server class loaders never load application client modules.

For each application server in the system, you can set the application class-loader policy to `Single` or `Multiple`. When the application class-loader policy is set to `Single`, then a single application class loader loads all EJB modules, dependency JAR files, and shared libraries in the system. When the application class-loader policy is set to `Multiple`, then each application receives its own class loader that is used for loading the EJB modules, dependency JAR files, and shared libraries for that application.

An application class loader loads classes from Web modules if the application's WAR class-loader policy is set to `Application`. If the application's WAR class-loader policy is set to `Module`, then each WAR module receives its own class loader.

The following example shows that when the application class-loader policy is set to `Single`, a single application class loader loads all of the EJB modules, dependency JAR files, and shared libraries of all applications on the server. The single application class loader can also load Web modules if an application has its WAR class-loader policy set to `Application`. Applications that have a WAR class-loader policy set to `Module` use a separate class loader for Web modules.

```
Server's application class-loader policy: Single
Application's WAR class-loader policy: Module
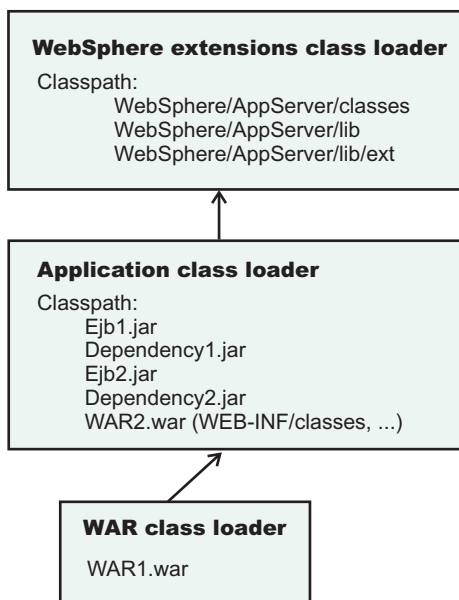
Application 1
 Module:  EJB1.jar
 Module: WAR1.war
   MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = Module
```

```
Application 2
 Module:   EJB2.jar
  MANIFEST Class-Path: Dependency2.jar
 Module: WAR2.war
  WAR Classloader Policy = Application
```

WebSphere extensions class loader

Classpath:
        WebSphere/AppServer/classes
        WebSphere/AppServer/lib
        WebSphere/AppServer/lib/ext

Application class loader

Classpath:
    Ejb1.jar
    Dependency1.jar
    Ejb2.jar
    Dependency2.jar
    WAR2.war (WEB-INF/classes, ...)

WAR class loader

  WAR1.war

The following example shows that when the application class-loader policy of an application server is set
to `Multiple`, each application on the server has its own class loader. An application class loader also loads
its Web modules if the application WAR class-loader policy is set to `Application`. If the policy is set to
`Module`, then a Web module uses its own class loader.

```
Server's application class-loader policy: Multiple
Application's WAR class-loader policy: Module
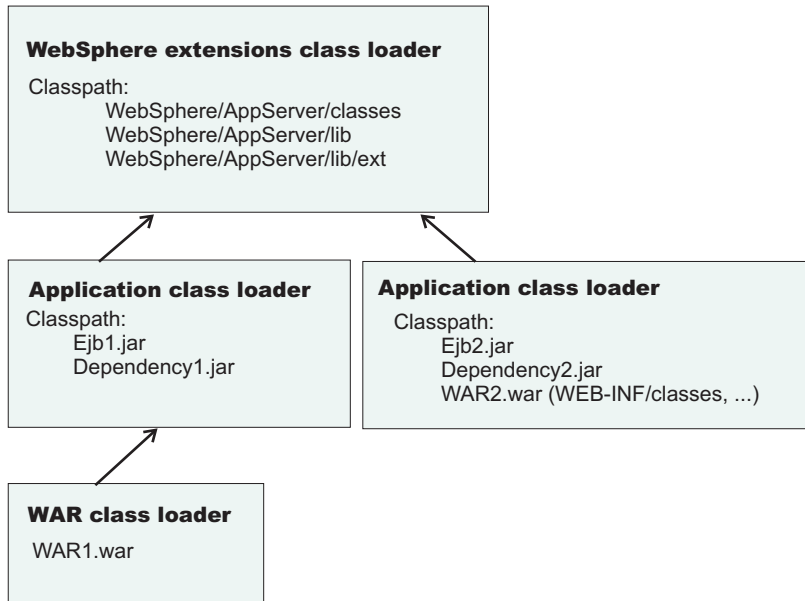
Application 1
 Module:   EJB1.jar
 Module: WAR1.war
  MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = Module
Application 2
 Module:   EJB2.jar
  MANIFEST Class-Path: Dependency2.jar
 Module: WAR2.war
  WAR Classloader Policy = Application
```

Class-loader modes

The class-loader delegation mode, also known as the *class loader order*, determines whether a class loader delegates the loading of classes to the parent class loader. The following values for class-loader mode are supported:

Class-loader mode	Description
Parent first Also known as **Classes loaded with parent class loader first**.	The `Parent first` or `Classes loaded with parent class loader first` class-loader mode 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. This value is the default for the class-loader policy and for standard JVM class loaders.
Parent last Also known as **Classes loaded with application class loader first**.	The `Parent last` or `Classes loaded with application class loader first` class-loader mode causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.

The following settings determine the mode of a class loader:
- If the application class-loader policy of an application server is `Single`, the server-level mode value defines the mode for an application class loader.
- If the application class-loader policy of an application server is `Multiple`, the application-level mode value defines the mode for an application class loader.
- If the WAR class-loader policy of an application is `Module`, the module-level mode value defines the mode for a WAR class loader.

Configuring class loaders of a server

You can configure the application class loaders for an application server. Class loaders enable applications that are deployed on the application server to access repositories of available classes and resources.

This topic assumes that an administrator created an application server on a WebSphere Application Server product.

Configure the class loaders of an application server to set class-loader policy and mode values which affect all applications that are deployed on the server. Use the administrative console to configure the class loaders.

1. Click **Servers > Application Servers >** *server_name* to access the settings page for an application server.

2. Specify the application class-loader policy for the application server. The application class-loader policy controls the isolation of applications that run in the system (on the server). An application class loader groups enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Dependency JAR files are JAR files that contain code which can be used by both enterprise beans and servlets. The application class-loader policy controls whether an application class loader can be shared by multiple applications or is unique for each application. Use the settings page for the application server to specify the application class-loader policy for the server:

Option	Description
Single	Applications are not isolated from each other. Uses a single application class loader to load all of the EJB modules, shared libraries, and dependency JAR files in the system.
Multiple	Applications are isolated from each other. Gives each application its own class loader to load the EJB modules, shared libraries, and dependency JAR files of that application.

3. Specify the application class-loader mode for the application server. The application class loading mode specifies the class-loader mode when the application class-loader policy is `Single`. On the settings page for the application server, select either of the following values:

Option	Description
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. `Parent first` is the default value for class loading mode.
Parent last	Causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.

4. Specify the class-loader mode for the class loader.
   a. On the settings page for the application server, click **Java and Process Management > Class loader** to access the Class loader page.
   b. On the Class loader page, click **New** to access the settings page for a class loader.
   c. On the settings page for a class loader, specify the class loader order. The `Classes loaded with parent class loader first` value 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. The `Classes loaded with application class loader first` value causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent.
   d. Click **OK**.

   An identifier is assigned to a class-loader instance. The instance is added to the collection of class loaders shown on the Class loader page.

Save the changes to the administrative configuration.

Class loader collection

Use this page to manage class-loader instances on an application server. A class loader determines whether an application class loader or a parent class loader finds and loads Java class files for an application.

To view this administrative console page, click **Servers > Application servers >** *server_name* **> Java and Process Management > Class loader**.

Class loader ID

Provides a string that is unique to the server identifying the class-loader instance. The product assigns the identifier.

Class loader order

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for development kit class loaders and WebSphere Application Server class loaders is `Classes loaded with parent class loader first`. By specifying `Classes loaded with application class loader first`, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

Class loader settings

Use this page to configure a class loader for applications that reside on an application server.

To view this administrative console page, click **Servers > Application servers >** *server_name* **> Java and Process Management > Class loader >** *class_loader_ID*.

Class loader ID

Provides a string that is unique to the server identifying the class-loader instance. The product assigns the identifier.

Data type   String

Class loader order

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for development kit class loaders and WebSphere Application Server class loaders is `Classes loaded with parent class loader first`. By specifying `Classes loaded with application class loader first`, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

The options are `Classes loaded with parent class loader first` and `Classes loaded with application class loader first`. The default is to search in the parent class loader before searching in the application class loader to load a class.

For your application to use the default configuration of Jakarta Commons Logging in WebSphere Application Server, set this application class loader order to `Classes loaded with parent class loader first`. For your application to override the default configuration of Jakarta Commons Logging in WebSphere Application Server, your application must provide the configuration in a form supported by Jakarta Commons Logging and this class loader order must be set to `Classes loaded with application class loader first`. Also, to override the default configuration, set the class loader order for each Web module in your application so that the correct logger factory loads.

Data type   String

Configuring application class loaders

You can set values that control the class-loading behavior of an installed enterprise application. Class loaders enable an application to access repositories of available classes and resources.

This topic assumes that you installed an application on an application server.

Configure the class loaders of an enterprise application to set class-loader policy and mode values for this application.

An application class loader groups enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Dependency JAR files are JAR files that contain code which can be used by both enterprise beans and servlets.

An application class loader is the parent of a Web application archive (WAR) class loader. By default, a Web module has its own WAR class loader to load the contents of the Web module. The WAR class-loader policy value of an application class loader determines whether the WAR class loader or the application class loader is used to load the contents of the Web module.

Use the administrative console to configure the class loaders.

1. Click **Applications > Enterprise Applications >** *application_name* **> Class loading and update detection** to access the settings page for an application class loader.
2. Specify whether to reload application classes when the application or its files are updated.

   By default, class reloading is not enabled. Select **Reload classes when application files are updated** to choose to reload application classes. You might specify different values for EJB modules and for Web modules such as servlets and JavaServer Pages (JSP) files.
3. Specify the number of seconds to scan the application's file system for updated files.

   The value specified for **Polling interval for updated files** takes effect only if class reloading is enabled. The default is the value of the reloading interval attribute in the IBM extension (`META-INF/ibm-application-ext.xmi`) file of the enterprise application (EAR file). You might specify different values for EJB modules and for Web modules such as servlets and JSP files.

   To enable reloading, specify an integer value that is greater than zero (for example, 1 to 2147483647).

   To disable reloading, specify zero (0).
4. Specify the class loader order for the application.

   The application class loader order specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The default is to search in the parent class loader before searching in the application class loader to load a class.

   Select either of the following values for **Class loader order**:

Option	Description
Classes loaded with parent class loader first	Causes the class loader to search in the parent class loader first to load a class. This value is the standard for Development Kit class loaders and WebSphere Application Server class loaders.

Option	Description
Classes loaded with application class loader first	Causes the class loader to search in the application class loader first to load a class. By specifying `Classes loaded with application class loader first`, your application can override classes contained in the parent class loader. **Attention:** Specifying the `Classes loaded with application class loader first` value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes.

5. Specify whether to use a single or multiple class loaders to load Web application archives (WAR files) of your application.

   By default, Web modules have their own WAR class loader to load the contents of the `WEB-INF/classes` and `WEB-INF/lib` directories. The default WAR class loader value is `Class loader for each WAR file in application`, which uses a separate class loader to load each WAR file. Setting the value to `Single class loader for application` causes the application class loader to load the Web module contents as well as the EJB modules, shared libraries, RAR files, and dependency JAR files associated to the application. The application class loader is the parent of the WAR class loader.

   Select either of the following values for **WAR class loader policy**:

Option	Description
Class loader for each WAR file in application	Uses a different class loader for each WAR file.
Single class loader for application	Uses a single class loader to load all of the WAR files in your application.

6. Click **OK**.

Save the changes to the administrative configuration.

Configuring Web module class loaders

You can set values that control the class-loading behavior of an installed Web module.

This topic assumes that you installed a Web module on an application server.

Configure the class loader order value of an installed Web module. By default, a Web module has its own Web application archive (WAR) class loader to load the contents of the Web module, which are in the `WEB-INF/classes` and `WEB-INF/lib` directories.

An application class loader is the parent of a WAR class loader. The WAR class-loader policy value of an application class loader determines whether the WAR class loader or the application class loader is used to load the contents of the Web module. The default WAR class loader policy value is `Class loader for each WAR file in application`. If the policy is set to `Class loader for each WAR file in application`, then each Web module receives its own class loader whose parent is the application class loader. If the policy is set to `Single class loader for application` on the settings page of an application class loader, then the application class loader loads the Web module contents as well as the enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Thus, the configuration of the parent application class loader affects the WAR class loader.

Use the administrative console to configure the application and WAR class loaders.

1. If you have not done so already, configure the application class loader.

Settings such as **Reload classes when application files are updated**, **Polling interval for updated files** and **WAR class loader policy** can affect Web module class loading.

If **WAR class loader policy** is set to `Class loader for each WAR file in application`, then the Web module receives its own class loader and the WAR class-loader policy of the Web module defines the mode for a WAR class loader. If the policy is set to `Single class loader for application`, then the application class loader loads the Web module contents.

2. Specify the class loader order for the installed Web module.

   The Web module class-loader mode specifies whether the class loader searches in the parent application class loader or in the WAR class loader first to load a class. The default is to search in the parent application class loader before searching in the WAR class loader to load a class.

   Select either of the following values for **Class loader order**:

Option	Description
Classes loaded with parent class loader first	Causes the class loader to search in the parent application class loader first to load a class. This is the standard for Development Kit class loaders and WebSphere Application Server class loaders. **Tip:** If classes and resources needed by the Web module cannot be accessed by the application class loader, but can be accessed by the WAR class loader, specify `Classes loaded with application class loader first`. If the application class loader cannot find a class, the class loader delegates the request to find the class to its parent, the WebSphere Application Server extensions class loader. If the WebSphere Application Server extensions class loader cannot find the class, the class loader delegates the request to its parent, the bootstrap, extensions, and CLASSPATH class loaders created by the Java virtual machine. Requests can only go to a parent class loader; they cannot go to a child class loader. Thus, if `Classes loaded with parent class loader first` is specified, the WAR class loader never receives a request to load a class.
Classes loaded with application class loader first	Causes the class loader to search in the WAR class loader first to load a class. By specifying `Classes loaded with application class loader first`, your WAR class loader can override classes contained in the parent application class loader. **Attention:** Specifying the `Classes loaded with application class loader first` value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes.

3. Click **OK**.

Save the changes to the administrative configuration.

Class loading: Resources for learning

Additional information and guidance on class loading is available on various Internet sites.

Use the following links to find relevant supplemental information about class loaders. 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.

Refer to "Web resources for learning" on page 15 for links to information applicable to WebSphere Application Server generally, such as lists of IBM technical papers, Redbooks and samples.

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 that is needed to resolve this problem. Before opening a PMR, see the IBM Support page.

View links to additional information about:
- "Programming model and decisions"
- "Programming instructions and examples"
- "Programming specifications"

Programming model and decisions
- Demystifying class loading problems, Part 1: An introduction to class loading and debugging tools - Learn how class loading works and how your JVM can help you sort out class loading problems (*developerWorks*, November 2005)
- Demystifying class loading problems, Part 2: Basic class loading exceptions - An in-depth look at some simple class loading quirks and conundrums (*developerWorks*, December 2005)
- Demystifying class loading problems, Part 3: Tackling more unusual class loading problems - Understand class loading and quash subtle exceptions (*developerWorks*, December 2005)
- J2EE Class Loading Demystified (*developerWorks*, August 2002)
- Java programming dynamics, Part 1: Classes and class loading - A look at classes and what goes on as they're loaded by a JVM (*developerWorks*, April 2003)

Programming instructions and examples
- WebSphere Application Server V6 System Management & Configuration Handbook
- *IBM WebSphere Developer Technical Journal:* Co-hosting multiple versions of J2EE applications
- Chapter 24 - J2EE Packaging and Deployment excerpted from Professional Java Server Programming J2EE 1.3 Edition

Programming specifications
- J2EE^{TM} Platform Specification
- J2EE^{TM} Extension Mechanism Architecture

Chapter 9. Deploying and administering applications

Deploying an application file consists of installing the application file on a server configured to hold installable modules.

Before installing an enterprise application or other installable module on an application server, you must develop the module, assemble the module, and configure the target server or cluster. Before choosing a deployment target for the module, ensure that the target version is compatible with your module.

During installation, you can configure the module enough to enable it to run on the server. After installation, you can configure the module further, start or stop the application, and otherwise manage its activity.

The topics in this section describe how to deploy and administer applications or modules using the administrative console. You can also use scripting or administrative programs (JMX).
* Install application files on an application server.
* Edit the administrative configuration for an application.
* **Optional:** View the deployment descriptor for an application or module.
* Start and stop the application.
* Export applications.
* Export DDL files.
* Update an application or module.
* Uninstall applications.
* Remove a file from an application or module.

If a changed application or module is deployed on a cluster, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Use **Rollout Update** if the application is deployed on one or more clusters spread across multiple nodes. This action reduces the amount of time that any single cluster member is unavailable to serve requests to the smallest interval possible. Pending IIOP transactions will complete before a cluster member stops; in-flight HTTP and JMS transactions might be lost while the cluster member is stopping. For an application server without clusters, use **Update** and then save and synchronize the node instead. For a standalone application server, simply update and save.

After making changes to administrative configurations of your applications in the administrative console, ensure that you save the changes.

Enterprise (J2EE) applications

Enterprise applications (or J2EE applications) are applications that conform to the Java 2 Platform, Enterprise Edition, specification.

Enterprise applications can consist of the following:
* Zero or more EJB modules (packaged in JAR files)
* Zero or more Web modules (packaged in WAR files)
* Zero or more connector modules (packaged in RAR files)
* Zero or more Session Initiation Protocol (SIP) modules (packaged in SAR files)
* Zero or more application client modules
* Additional JAR files containing dependent classes or other components required by the application

- Any combination of the above

A J2EE application is represented by, and packaged in, an enterprise archive (EAR) file.

System applications

A *system application* is a J2EE enterprise application that is central to a WebSphere Application Server product.

Examples of system applications include *isclite*, *managementEJB* and *filetransfer*.

Because a system application is an important part of a WebSphere Application Server product, a system application is deployed when the product is installed and is updated only through a product fix or upgrade. For some system applications, such as *filetransfer*, users cannot change the metadata for the system application, unless the metadata assigns users and groups for security purposes. For these applications, non-security related metadata such as its J2EE bindings or J2EE extensions must be updated through a product fix or upgrade.

System applications are not shown in the list of installed applications on the console Enterprise Applications page, or through wsadmin and Java application programming interfaces, to prevent users from accidentally stopping, updating or removing the system applications.

Note that J2EE Samples are not system applications even though they are provided as part of a WebSphere Application Server product. Similarly, applications that support changes to their metadata are not system applications.

Installing application files

As part of deploying an application, you install application files on a server configured to hold installable modules.

Before you can install your application files on an application server, you must configure the target application server. As part of configuring the server, determine whether your application files can be installed to your deployment targets.

Also, before you install the files, assemble modules as needed.

Installable modules include enterprise archive (EAR), enterprise bean (EJB), Web archive (WAR), Session Initiation Protocol (SIP) module (SAR), resource adapter (connector or RAR), and application client modules. Application client files can be installed in a WebSphere Application Server configuration but cannot be run on a server. Complete the following steps to install your files.

1. Determine which method to use to install your application files. WebSphere Application Server provides several ways to install modules.
2. Install the application files using
   - Administrative console
   - wsadmin scripts
   - Java administrative programs that use JMX APIs
   - Java programs that define a J2EE DeploymentManager object in accordance with J2EE Deployment API Specification (JSR-88)
3. Start the deployed application files using
   - Administrative console
   - wsadmin startApplication
   - Java programs that use ApplicationManager or AppManagement MBeans

- Java programs that define a J2EE DeploymentManager object in accordance with J2EE Deployment API Specification (JSR-88)

Save the changes to your administrative configuration.

When saving the configuration, synchronize the configuration with the nodes where the application is expected to run.

Next, test the application. For example, point a Web browser at the URL for a deployed application (typically http://*hostname*:9060/*Web_module_name*, where *hostname* is your valid Web server and 9060 is the default port number) and examine the performance of the application. If the application does not perform as desired, edit the application configuration, then save and test it again.

Installable module versions

The contents of a module affect whether you can install the module on a WebSphere Application Server Version 6.0 and later (6.x) deployment target, or if you must install the module on a Version 5.0 and later (5.x) deployment target.

Installable application modules

You can install an application, enterprise bean (EJB) module, Session Initiation Protocol (SIP) module (SAR), or Web module developed for a Version 5.x product on a 5.x or 6.x deployment target, provided the module:
- Does not support Java 2 Platform, Enterprise Edition (J2EE) 1.4;
- Does not call any 6.x runtime application programming interfaces (APIs); and
- Does not use any 6.x product features.

If the module supports J2EE 1.4, then you must install the module on a 6.x deployment target. If the module calls a 6.1.x API or uses a 6.1.x feature, then you must install the module on a 6.1.x deployment target. Modules that call a 6.0.x API or use a 6.0.x feature can be installed on a 6.0.x or 6.1.x deployment target.

Selecting options such as **Precompile JavaServer Pages files**, **Use binary configuration**, **Deploy Web services** or **Deploy enterprise beans** during application installation indicates that the application uses 6.1.x product features. You cannot deploy such applications on a 5.x or 6.0.x deployment target. You must deploy such applications on a 6.1.x deployment target.

Similarly, you must deploy an application that uses J2EE 1.4 features such as Java Authorization Contract for Containers (JACC) provided by an application server on a 6.x deployment target.

Installable RAR files

You can install a standalone resource adapter (connector) module, or RAR file, developed for a Version 5.x product to a 5.x or 6.x deployment target, provided the module does not call any 6.x runtime APIs. If the module calls a 6.x API, then you must install the module on a 6.x deployment target.

Deployment targets

A *5.x deployment target* is a server or a cluster with at least one member on a WebSphere Application Server Version 5 product.

A *6.x deployment target* is a server or cluster with all members on a WebSphere Application Server Version 6 product.

Table 49. Compatible deployment target versions for 5.x and 6.x modules

Module type	Module Java support	Module calls 6.x runtime APIs or uses 6.x features?	Client versions that can install module	Deployment target versions
Application, EJB, Web, or client	J2EE 1.3	No	5.x or 6.x	5.x or 6.x
Application, EJB, Web, or client	J2EE 1.3	Yes	6.x	6.x Modules that call 6.1.x runtime APIs or use 6.1.x features must be installed on a 6.1.x deployment target. Modules that call 6.0.x runtime APIs or use 6.0.x features can be installed on any 6.x deployment target.
Application, EJB, SAR, Web, or client	J2EE 1.4	Yes or No	6.x	6.x
Resource adapter	JCA 1.0	No	5.x or 6.x	5.x or 6.x
Resource adapter	JCA 1.0	Yes	6.x	6.x Modules that call 6.1.x runtime APIs must be installed on a 6.1.x deployment target. Modules that call 6.0.x runtime APIs can be installed on any 6.x deployment target.
Resource adapter	JCA 1.5	Yes or No	6.x	6.x Modules that call 6.1.x runtime APIs must be installed on a 6.1.x deployment target. Modules that call 6.0.x runtime APIs can be installed on any 6.x deployment target.

Ways to install applications or modules

The product provides several ways to install application files.

Installable files include enterprise archive (EAR), enterprise bean (EJB), Web archive (WAR), Session Initiation Protocol (SIP) module (SAR), resource adapter (connector or RAR), and application client modules. They can be installed on a server or cluster. Application client files can be installed in a WebSphere Application Server configuration but cannot be run on a server.

Table 50. Ways to install application files

Option	Method	Modules	Comments	Starting after install
Administrative console install wizard See "Installing application files with the console" on page 1368.	Click **Applications > Install New Application** in the console navigation tree and follow instructions in the wizard.	All EAR, EJB, WAR, SAR, RAR, and application client files	Provides one of the easier ways to install application files. See "Preparing for application installation settings" on page 1373 for guidance. For applications that do not require changes to the default bindings, select **Show me all installation options and parameters**, select **Generate default bindings**, click the Summary step, and then click **Finish**.	Click **Start** on the Enterprise Applications page accessed by clicking **Applications > Enterprise Applications** in the console navigation tree.
wsadmin scripts	Invoke AdminApp object *install* commands in a script or at a command prompt.	All EAR, EJB, WAR, SAR, RAR, and application client files	Getting started with scripting provides an overview of wsadmin.	• Invoke the AdminApp *startApplication* command. • Invoke the *startApplication* method on an ApplicationManager MBean using AdminControl.
Java application programming interfaces	Install programs by completing the steps in Installing an application through programming.	All EAR files	Use MBeans to install the application. Managing applications through programming provides an overview of Java MBean programming.	Start the application by calling the *startApplication* method on a proxy.
WebSphere rapid deployment Refer to articles under Rapid deployment of J2EE applications in this information center.	Briefly, do the following: 1. Update your J2EE application files. 2. Set up the rapid deployment environment. 3. Create a free-form project. 4. Launch a rapid deployment session. 5. Drop your updated application files into the free-form project.	All J2EE modules, including EAR files and standalone EJB, WAR, SAR, RAR, and application client files	WebSphere rapid deployment offers the following advantages: • You do not need to assemble your J2EE application files prior to deployment. • You do not need to use other installation tools mentioned in this table to deploy the files.	Use any of the above options to start the application. Clicking **Start** on the Enterprise Applications page is the easiest option.

Table 50. Ways to install application files (continued)

Option	Method	Modules	Comments	Starting after install
Java programs	Code programs that use J2EE DeploymentManager (JSR-88) methods.	All J2EE modules, including EAR files and standalone EJB, WAR, SAR, RAR, and application client files	• Uses J2EE Application Deployment Specification (JSR-88). • Can customize modules using DConfigBeans.	Call the J2EE DeploymentManager (JSR-88) method *start* in a program to start the deployed modules when the module's running environment initializes.

Installing application files with the console

Installing 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*.

Before installing enterprise application files, ensure that you are installing your application files onto a compatible deployment target. If the deployment target is not compatible, select a different target.

To install new enterprise application files to a WebSphere Application Server configuration, you can use the administrative console, the wsadmin tool, Java MBean programs, or Java programs that call J2EE DeploymentManager (JSR-88) methods. This topic describes how to use the administrative console to install an application, EJB component, Session Initiation Protocol (SIP) module (SAR), or Web module.

Important: After you start performing the steps below, click **Cancel** to exit if you decide not to install the application. Do not simply move to another administrative console page without first clicking **Cancel** on an application installation page.

1. Click **Applications > Install New Application** in the console navigation tree.
2. On the first Preparing for application installation page:
   a. Specify the full path name of the source enterprise application file (`.ear` file otherwise known as an *EAR file*). The EAR file that you are installing can be either on the client machine (the machine that runs the Web browser) or on the server machine (the machine to which the client is connected). If you specify an EAR file on the client machine, then the administrative console uploads the EAR file to the machine on which the console is running and proceeds with application installation. You can also specify a standalone Web application archive (WAR), SAR or Java archive (JAR) file for installation.
   b. If you are installing a standalone WAR or SAR file, specify the context root.
   c. Select whether to view all installation options.
      Prompt me only when additional information is required
         Displays the module mapping step as well as any steps that require you to specify needed information to install the application successfully.
      Show me all installation options and parameters
         Displays all installation options. To use **Generate default bindings**, which supplies default values for incomplete bindings, select this option.
   d. Click **Next**.
3. If you selected **Show me all installation options and parameters**, for the second Preparing for application installation page:
   a. Select whether to generate default bindings. Using the default bindings causes any incomplete bindings in the application to be filled in with default values. Existing bindings are not altered. You can customize default values used in generating default bindings. For example, you can specify a Java Naming and Directory Interface (JNDI) prefix for EJB files in EJB modules, default data

source and connection factory settings for EJB modules, virtual host for Web modules, and so on. "Preparing for application installation settings" on page 1373 describes available customizations and provides sample bindings.

b. Click **Next**. If security warnings are displayed, click **Continue**. The Install New Application pages are displayed. If you chose to generate default bindings, you can proceed to the Summary step. "Example: Installing an EAR file using the default bindings" on page 1388 provides sample steps.

4. Specify values for installation options as needed.

You can click on a step number to move directly to that panel instead of clicking **Next**.

Panel	Description
Select installation options	On the **Select installation options** panel, provide values for the settings specific to WebSphere Application Server. Default values are used if you do not specify a value.
Map modules to servers	On the **Map modules to servers** panel, specify deployment targets where you want to install the modules contained in your application. Modules can be installed on the same deployment target or dispersed among several deployment targets. Each module must be mapped to a target server. A deployment target can be an application server, cluster of application servers or Web server.
Provide options to compile JSPs	If the **Precompile JavaServer Pages files** setting is enabled on the **Select installation options** panel and your application uses JavaServer Pages (JSP) files, then you can specify JSP compiler options on the **Provide options to compile JSPs** panel.
Provide JNDI names for beans	If your application uses EJB modules, on the **Provide JNDI names for beans** panel, specify a JNDI name for each enterprise bean in every EJB module. You must specify a JNDI name for every enterprise bean defined in the application. For example, for the EJB module `MyBean.jar`, specify `MyBean`.
Map default data sources for modules containing 1.x entity beans	If your application uses EJB modules that contain Container Managed Persistence (CMP) beans that are based on the EJB 1.x specification, for **Map default data sources for modules containing 1.x entity beans**, specify a JNDI name for the default data source for the EJB modules. The default data source for the EJB modules is optional if data sources are specified for individual CMP beans.
Map data sources for all 1.x CMP beans	If your application has CMP beans that are based on the EJB 1.x specification, for **Map data sources for all 1.x CMP beans**, specify a JNDI name for data sources to be used for each of the 1.x CMP beans. The data source attribute is optional for individual CMP beans if a default data source is specified for the EJB module that contains CMP beans. If neither a default data source for the EJB module nor a data source for individual CMP beans are specified, then a validation error displays after you click **Finish** and the installation is cancelled.
Map EJB references to beans	If your application defines EJB references, for Map EJB references to beans, specify JNDI names for enterprise beans that represent the logical names specified in EJB references. Each EJB reference defined in the application must be bound to an EJB file before clicking **Finish** on the Summary panel.
Map resource references to resources	If your application defines resource references, for **Map resource references to resources**, specify JNDI names for the resources that represent the logical names defined in resource references. You can optionally specify login configuration name and authentication properties for the resource. After specifying authentication properties, click **OK** to save the values and return to the mapping step. Each resource reference defined in the application must be bound to a resource defined in your WebSphere Application Server configuration before clicking on **Finish** on the Summary panel.

Panel	Description
Map virtual hosts for Web modules	If your application uses Web modules, for **Map virtual hosts for Web modules**, select a virtual host from the list that should map to a Web module defined in the application. The port number specified in the virtual host definition is used in the URL that is used to access artifacts such as servlets and JSP files in the Web module. Each Web module must have a virtual host to which it maps. Not specifying all needed virtual hosts will result in a validation error displaying after you click **Finish** on the Summary panel.
Map security roles to users or groups	If the application has security roles defined in its deployment descriptor then, for **Map security roles to users or groups**, specify users and groups that are mapped to each of the security roles. Select **Role** to select all of the roles or select individual roles. For each role, you can specify whether predefined users such as **Everyone** or **All authenticated users** are mapped to it. To select specific users or groups from the user registry: 1. Select a role and click **Lookup users** or **Lookup groups**. 2. On the Lookup users or groups panel displayed, enter search criteria to extract a list of users or groups from the user registry. 3. Select individual users or groups from the results displayed. 4. Click **OK** to map the selected users or groups to the role selected on the **Map security roles to users or groups** panel.
Map RunAs roles to users	If the application has Run As roles defined in its deployment descriptor, for **Map RunAs roles to users**, specify the Run As user name and password for every Run As role. Run As roles are used by enterprise beans that must run as a particular role while interacting with another enterprise bean. Select **Role** to select all of the roles or select individual roles. After selecting a role, enter values for the user name, password, and verify password and click **Apply**.
Ensure all unprotected 1.x methods have the correct level of protection	If your application contains EJB 1.x CMP beans that do not have method permissions defined for some of the EJB methods, for **Ensure all unprotected 1.x methods have the correct level of protection**, specify if you want to leave such methods unprotected or assign protection with deny all access.
Bind listeners for message-driven beans	If your application contains message driven enterprise beans, for **Bind listeners for message-driven beans**, provide a listener port name or an activation specification JNDI name for every message driven bean.
Map default data sources for modules containing 2.x entity beans	If your application uses EJB modules that contain CMP beans that are based on the EJB 2.x specification, for **Map default data sources for modules containing 2.x entity beans**, specify a JNDI name for the default data source and the type of resource authorization to be used for the default data source for the EJB modules. You can optionally specify a login configuration name and authentication properties for the data source. When creating authentication properties, you must click **OK** to save the values and return to the mapping step. The default data source for EJB modules is optional if data sources are specified for individual CMP beans.
Map data sources for all 2.x CMP beans	If your application has CMP beans that are based on the EJB 2.x specification, on the **Map data sources for all 2.x CMP beans** panel, for each of the 2.x CMP beans specify a JNDI name and the type of resource authorization for data sources to be used. You can optionally specify a login configuration name and authentication properties for the data source. When creating authentication properties, you must click **OK** to save the values and return to the mapping step. The data source attribute is optional for individual CMP beans if a default data source is specified for the EJB module that contains CMP beans. If neither a default data source for the EJB module nor a data source for individual CMP beans are specified, then a validation error is displayed after you click **Finish** and installation is cancelled.

Panel	Description
Ensure all unprotected 2.x methods have the correct level of protection	If your application contains EJB 2.x CMP beans that do not have method permissions defined in the deployment descriptors for some of the EJB methods, on the **Ensure all unprotected 2.x methods have the correct level of protection** panel, specify whether you want to assign a specific role to the unprotected methods, add the methods to the exclude list, or mark them as unchecked. Methods added to the exclude list are marked as uncallable. For methods marked unchecked no authorization check is performed prior to their invocation.
Provide options to perform the EJB Deploy	If the **Deploy enterprise beans** setting is enabled on the **Select installation options** panel, then you can specify options for the EJB deployment tool on the **Provide options to perform the EJB Deploy** panel. On this panel, you can specify extra class paths, RMIC options, database types, and database schema names to be used while running the EJB deployment tool.
Map shared libraries	On the **Shared library references** and **Shared library mapping** panels, specify shared library files for your application or Web modules to use. A defined shared library must exist to associate your application or module to the library file.
Provide JSP reloading options for Web modules	If your application uses Web modules, for **Provide JSP reloading options for Web modules**, configure the class reloading of JavaServer Pages (JSP) files.
Map context roots for Web modules	If your application uses Web modules, for **Map context roots for Web modules**, specify a context root for each Web module in the application.
Initialize parameters for servlets	If your application uses Web modules, for **Initialize parameters for servlets**, specify or override initial parameters that are passed to the init method of Web module servlet filters.
Map environment entries for Web modules	If your application uses Web modules, for **Map environment entries for Web modules**, configure the environment entries of Web modules such as servlets and JSP files.
Map resource environment entry references to resources	If your application contains resource environment references, for **Map resource environment entry references to resources**, specify JNDI names of resources that map to the logical names defined in resource environment references. If each resource environment reference does not have a resource associated with it, after you click **Finish** a validation error is displayed.
Correct use of system identity	If your application defines **Run-As Identity** as *System Identity*, for **Correct use of system identity**, you can optionally change it to *Run-As role* and specify a user name and password for the Run As role specified. Selecting *System Identity* implies that the invocation is done using the WebSphere Application Server security server ID and should be used with caution as this ID has more privileges.
Correct isolation levels for all resource references	If your application has resource references that map to resources that have an Oracle database doing backend processing, for **Correct isolation levels for all resource references**, specify or correct the isolation level to be used for such resources when used by the application. Oracle databases support ReadCommitted and Serializable isolation levels only.
Bind message destination references to administered objects	If your application uses message driven beans, for **Bind message destination references to administered objects**, specify the JNDI name of the J2C administered object to bind the message destination reference to the message driven beans. **Attention:** If multiple message destination references are linked to the same message destination, only one JNDI name is collected. When a message destination reference links to the same message destination as a message driven bean and the destination JNDI name has been collected already, the destination JNDI name for the message destination reference is not collected.
Provide JNDI names for JCA objects	If your application contains an embedded .rar file, for **Provide JNDI names for JCA objects**, specify the name and JNDI name of each J2C connection factory, J2C administered object and J2C activation specification.

Panel	Description
Bind J2C activationspecs to destination JNDI names	If your application contains an embedded `.rar` file, its activationSpec property has the value `Destination`, and its introspected type is `javax.jms.Destination`, for **Bind J2C activationspecs to destination JNDI names**, specify the jndiName value for each activation bound to it.
Select current backend ID	If your application has EJB modules for which deployment code has been generated for multiple backend databases using an assembly tool, for **Select current backend ID**, specify the backend ID representing the backend database to be used when the EJB module runs. For information on backend databases, refer to EJB deployment tool. This step is not shown if the **Deploy enterprise beans** setting is enabled on the **Select installation options** panel and if a database type other than `None` is specified on the **Provide options to perform the EJB Deploy** panel.
Provide options to perform the Web services deployment	If the **Deploy Web services** setting is enabled on the **Select installation options** panel and your application uses Web services, then you can specify `wsdeploy` command options on the **Provide options to perform the Web services deployment** panel. For information on this panel, refer to descriptions of the `wsdeploy -cp` and `-jardir` options.
Configure remote request dispatch properties	If you are using a remote request dispatcher, you need to first configure it for use. 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. Refer to "Remote request dispatch property settings" on page 90 for information on where to configure these properties in the administration console.

5. On the Summary panel, verify the cell, node, and server onto which the application modules will install:
   a. Beside **Cell/Node/Server**, click **Click here**.
   b. Verify the settings.
   c. Return to the Summary panel.
   d. Click **Finish**.

Several messages are displayed, indicating whether your application file is installing successfully.

If **Validate input off/warn/fail** on the **Select installation options** panel is set to **warn**, the default, several validation warnings might be displayed. If the setting is **fail**, the validation warnings might cause errors.

If you receive an OutOfMemory exception and the source application file does not install, your system might not have enough memory or your application might have too many modules in it to install successfully onto the server. If lack of system memory is not the cause of the exception, package your application again so the `.ear` file has fewer modules. If lack of system memory and the number of modules are not the cause of the exception, check the options you specified on the Java Virtual Machine page of the application server running the administrative console. Then, try installing the application file again.

▶ Windows ◀ During installation certain application files are extracted in the directory represented by the configuration session and, when the configuration is saved, these files are saved in the WebSphere Application Server configuration repository. On Windows machines, there is a limit of 256 characters for file paths. Therefore, the application installation might fail if the path for application files in the configuration session or in the configuration repository exceeds the limit of 256 characters. You might see FileNotFound exceptions with *path name too long* in the message. To overcome such problems, make application names and module URI names shorter in length, which helps reduce the file path length. Then, try installing the application file again.

After the application file installs successfully, do the following:

1. Save the changes to your configuration.

   The application is registered with the administrative configuration and application files are copied to the target directory, which is *app_server_root*/installedApps/*cell_name* by default or the directory that you designate.

   For a Network Deployment installation, files are copied to remote nodes when the configuration on the deployment manager synchronizes with the configuration on individual nodes.

2. Start the application.

3. Test the application. For example, point a Web browser at the URL for the deployed application and examine the performance of the application. If necessary, edit the application configuration.

Preparing for application installation settings

Use this page to install an application (EAR file) or module (JAR, SAR or WAR file).

To view this administrative console page, click **Applications > Install New Application**.

Follow the steps on this page to install an application or module. You must complete, at minimum, the first step; you must complete some or all of the later steps, depending on whether you are installing an application, EJB module, SIP module or Web module.

Path to the new application:

Specifies the fully qualified path to the .ear, .jar, .sar, or .war file for the enterprise application.

Use **Local file system** if the browser and application files are on the same machine (whether or not the server is on that machine, too).

Use **Remote file system** if the application file resides on any node in the current cell context.You can browse the entire file system of a node if the node agent or deployment manager is running on that selected node. Only .ear, .jar, .sar, or .war files are shown during the browsing.

During application installation, application files typically are uploaded from a client machine running the browser to the server machine running the administrative console, where they are deployed. In such cases, use the Web browser running the administrative console to select EAR, WAR, SAR or JAR modules to upload to the server machine.

In some cases, however, the application files reside on the file system of any of the nodes in a cell. To have the application server install these files, use the **Remote file system** option.

Also use the **Remote file system** option to specify an application file already residing on the machine running the application server. For example, the field value might be *profile_root*/installableApps/ test.ear. If you are installing a standalone WAR module, then specify the context root as well.

After the application file is transferred, the **Remote file system** value shows the path of the temporary location on the deployment manager or server machine.

Context root:

Specifies the context root of the Web application (WAR)or a Session Initiation Protocol (SIP) module (SAR).

This field is used only to install a standalone WAR or SAR file. The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is http://host:port/gettingstarted/MySession.

How do you want to install the application?:

Specifies whether to show only installation options that require you to supply information or to show all installation options.

The **Prompt me only when additional information is required** option enables you to install your application more easily because you do not need to examine all available installation options.

However, to use the **Generate default bindings** option, which might be the quickest and easiest option for installing your application, you must select the **Show me all installation options and parameters** and then select **Generate default bindings** on the next panel.

Generate default bindings:

Specifies whether to generate default bindings. If you place a check mark in the check box, then any incomplete bindings in the application are filled in with default values. Existing bindings are not altered.

By choosing this option, you can directly jump to the Summary step and install the application if none of the steps have a red asterisk (*) next to them. A red asterisk denotes that the step has incomplete data and requires a valid value. On the Summary panel, verify the cell, node and server on which the application is installed.

You must select **Show me all installation options and parameters** to view this option.

Bindings are generated as follows:
- EJB JNDI names are generated of the form *prefix*/*ejb-name*. The default prefix is `ejb`, but can be overridden. The *ejb-name* is as specified in the deployment descriptors `<ejb-name>` tag.
- EJB references are bound as follows: If an `<ejb-link>` is found, it is honored. Otherwise, if a unique enterprise bean is found with a matching home (or local home) interface as the referenced bean, the reference is resolved automatically.
- Resource reference bindings are derived from the `<res-ref-name>` tag. Note that this action assumes that the `java:comp/env` name is the same as the resource global JNDI name.
- Connection factory bindings (for EJB 2.0 JAR files) are generated based on the JNDI name and authorization information provided. This action results in default connection factory settings for each EJB 2.0 JAR file in the application being installed. No bean-level connection factory bindings are generated.
- Data source bindings (for EJB 1.1 JAR files) are generated based on the JNDI name, data source user name password options. This results in default data source settings for each EJB JAR file. No bean-level data source bindings are generated.
- For EJB2.1 or EJB2.0 message-driven beans deployed as JCA 1.5-compliant resources, the JNDI names corresponding to activationSpec instances are generated in the form eis/*MDB_ejb-name*. Message Destination references are bound as follows: if a `<message-destination-link>` is found then the JNDI name is set to ejs/*message-destination-link*Name. Otherwise the JNDI name is set to eis/*message-destination-ref*Name.
- For EJB 2.0 message-driven beans deployed against a listener ports, the listener ports are derived from the MDB `<ejb-name>` tag with the string `Port` appended.
- For `.war` files, the virtual host is set as `default_host` unless otherwise specified.

The default strategy suffices for most applications or at least for most bindings in most applications. However, it does not work if:
- You want to explicitly control the global JNDI names of one or more EJB files.
- You need tighter control of data source bindings for container-managed persistence (CMP) beans. That is, you have multiple data sources and need more than one global data source.
- You must map resource references to global resource JNDI names that are different from the `java:comp/env` name.

In such cases, you can change the behavior with an XML document (a custom strategy). Use the **Specific bindings file** field to specify a custom strategy and see the field's help for examples.

Prefixes:

Specifies prefixes to use for generated JNDI names.

You must select **Show me all installation options and parameters** to view prefix options.

Prefixes is similar to the scripting option `-defaultbinding.ejbjndi.prefix`.

Override:

Specifies whether generated bindings are to override existing bindings.

If **Override existing bindings** is selected, the existing bindings are overridden by the generated ones.

You must select **Show me all installation options and parameters** to view override options.

Override is similar to the scripting option `-defaultbinding.force`.

EJB 1.1 CMP bindings:

Specifies the default data source JNDI name.

If the **Default bindings for EJB 1.1 CMPs** radio button is selected, specify the JNDI name for the default data source to be used with the container-managed persistence (CMP) 1.1 beans. Also specify the user ID and password for this default data source.

You must select **Show me all installation options and parameters** to view EJB CMP binding options.

Default bindings for EJB 1.1 CMPs is similar to the scripting option `-defaultbinding.datasource.jndi`.

Data source bindings for 2.0 CMP beans:

Specifies the default data source JNDI name for 2.0 CMP beans.

If **Default data source for 2.0 CMP beans** is selected, specify the JNDI name for the default data source to be used with the 2.0 CMP beans. Also specify the resource authorization.

You must select **Show me all installation options and parameters** to view data source binding options.

Default data source for 2.0 CMP beans is similar to the scripting option `-defaultbinding.cf.jndi`.

Virtual host:

Specifies the virtual host for the Web module.

You must select **Show me all installation options and parameters** to view virtual host options.

Virtual host is similar to the scripting option `-defaultbinding.virtual.host`.

Specific bindings file:

Specifies a bindings file that overrides the default binding.

You must select **Show me all installation options and parameters** to view this option.

Specific bindings file is similar to the scripting option `-defaultbinding.strategy.file`.

Change the behavior of the default binding with an XML document (a custom strategy). Custom strategies extend the default strategy so you only need to customize those areas where the default strategy is insufficient. Thus, you only need to describe how you want to change the bindings generated by the default strategy; you do not have to define bindings for the entire application.

Brief examples of how to override various aspects of the default bindings generator follow:

Controlling an EJB JNDI name

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>helloEjb.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
         <ejb-name>HelloEjb</ejb-name>
         <jndi-name>com/acme/ejb/HelloHome</jndi-name>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Note: Ensure that the setting for `<ejb-name>` matches the `ejb-name` entry in the EJB JAR deployment descriptor. Here the setting is `<ejb-name>HelloEjb</ejb-name>`.

Setting the connection factory binding for an EJB JAR file

```
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>yourEjb20.jar</jar-name>
      <connection-factory>
        <jndi-name>eis/jdbc/YourData_CMP</jndi-name>
        <res-auth>Container</res-auth>
      </connection-factory>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Setting the connection factory binding for an EJB file

```
<?xml version="1.0">
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>yourEjb20.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourCmp20</ejb-name>
          <connection-factory>
           <jndi-name>eis/jdbc/YourData_CMP</jndi-name>
           <res-auth>PerConnFact</res-auth>
          </connection-factory>
        </ejb-binding>
```

```
        </ejb-bindings>
      </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Restriction: Ensure that the setting for `<ejb-name>` matches the `ejb-name` tag in the deployment descriptor. Here the setting is `<ejb-name>YourCmp20</ejb-name>`.

Setting the message destination reference JNDI for a specific enterprise bean

Example XML extract in a custom strategy file for setting message-destination-refs for a specific enterprise bean.

```
<?xml version="1.0">
 <!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
 <dfltbndngs>
  <module-bindings>
   <ejb-jar-binding>
     <jar-name>yourEjb21.jar</jar-name>
     <ejb-bindings>
      <ejb-binding>
        <ejb-name>YourSession21</ejb-name>
        <message-destination-ref-bindings>
         <message-destination-ref-binding>
           <message-destination-ref-name>jdbc/MyDataSrc</message-destination-ref-name>
           <jndi-name>eis/somAO</jndi-name>
         </message-destination-ref-binding>
        </message-destination-ref-bindings>
      </ejb-binding>
     </ejb-bindings>
   </ejb-jar-binding>
  </module-bindings>
 </dfltbndngs>
```

Restriction: Ensure that the setting for `<ejb-name>` matches the `ejb-name` tag in the deployment descriptor. Here the setting is `<ejb-name>YourSession21</ejb-name>`. Also ensure that the setting for `<message-destination-ref-name>` matches the `message-destination-ref-name` tag in the deployment descriptor. Here the setting is `<message-destination-ref-name>jdbc/MyDataSrc</message-destination-ref-name>`.

Overriding a resource reference binding from a WAR, EJB JAR file, or J2EE client JAR file

Example code for overriding a resource reference binding from a WAR file follows. Use similar code to override a resource reference binding from an enterprise bean (EJB) JAR file or a J2EE client JAR file.

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <war-binding>
      <jar-name>hello.war</jar-name>
      <resource-ref-bindings>
        <resource-ref-binding>
          <resource-ref-name>jdbc/MyDataSrc</resource-ref-name>
          <jndi-name>war/override/dataSource</jndi-name>
        </resource-ref-binding>
      </resource-ref-bindings>
    </war-binding>
  </module-bindings>
</dfltbndngs>
```

Restriction: Ensure that the setting for `<resource-ref-name>` matches the `resource-ref` tag in the deployment descriptor. Here the setting is `<resource-ref-name>jdbc/MyDataSrc</resource-ref-name>`.

Overriding the JNDI name for a message-driven bean deployed as a JCA 1.5-compliant resource

Example XML extract in a custom strategy file for overriding the JMS activationSpec JNDI name for an
EJB 2.1 or EJB 2.0 message-driven bean deployed as a JCA 1.5-compliant resource.

```xml
<?xml version="1.0"?>
 <!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
  <dfltbndngs>
  <module-bindings>
   <ejb-jar-binding>
     <jar-name>YourEjbJar.jar</jar-name>
     <ejb-bindings>
      <ejb-binding>
        <ejb-name>YourMDB</ejb-name>
        <activationspec-jndi-name>activationSpecJNDI</activationspec-jndi-name>
      </ejb-binding>
     </ejb-bindings>
   </ejb-jar-binding>
  </module-bindings>
 </dfltbndngs>
```

Overriding the JMS listener port name for an EJB 2.0 message-driven bean

Example XML extract in a custom strategy file for overriding the JMS listener port name for an EJB 2.0
message-driven bean deployed against a listener port.

```xml
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>YourEjbJar.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourMDB</ejb-name>
          <listener-port>yourMdbListPort</listener-port>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Overriding an EJB reference binding from an EJB JAR, WAR file, or EJB file

Example code for overriding an EJB reference binding from an EJB JAR file follows. Use similar code to
override an EJB reference binding from a WAR file or an EJB file.

```xml
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>YourEjbJar.jar</jar-name>
      <ejb-ref-bindings>
        <ejb-ref-binding>
          <ejb-ref-name>YourEjb</ejb-ref-name>
          <jndi-name>YourEjb/JNDI</jndi-name>
        </ejb-ref-binding>
      </ejb-ref-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Select installation options settings

Use this panel to specify options for the installation of an application onto a WebSphere Application Server deployment target. Default values for the options are used if you do not specify a value. After application installation, you can specify values for many of these options from an enterprise application settings page.

To view this administrative console panel, click **Applications > Install New Application** and then specify values as needed for your application on the Preparing for application installation pages. The Select installation options panel is the same for the application installation and update wizards.

Precompile JavaServer Pages files:

Specify whether to precompile JavaServer Pages (JSP) files as a part of installation. The default is not to precompile JSP files.

For this option, install only onto a 6.1 deployment target.

If you select **Precompile JavaServer Pages files** and try installing your application onto an earlier deployment target such as version 5.x, the installation is rejected. You can deploy applications to only those targets that have same WebSphere version as the deployment manager. If applications are targeted to servers that have an earlier version than the deployment manager, then you cannot deploy to those targets.

Data type	Boolean
Default	False

Directory to install application:

Specifies the directory to which the enterprise application (EAR) file will be installed.

The default value is the value of `APP_INSTALL_ROOT`/*cell_name*, where the `APP_INSTALL_ROOT` variable is *app_server_root*/`installedApps`; for example, *app_server_root*/`installedApps`/*cell_name*.

Important: If an installation directory is not specified when an application is installed on a single-server configuration, the application is installed in `APP_INSTALL_ROOT`/*cell_name*. When the server is made a part of a Network Deployment configuration (using the addNode utility), the cell name of the new configuration becomes the cell name of the deployment manager node. If the `-includeapps` option is used for the addNode utility, then the applications that are installed prior to the addNode operation still use the installation directory `APP_INSTALL_ROOT`/*cell_name*. However, an application that is installed after the server is added to the network configuration uses the default installation directory `APP_INSTALL_ROOT`/*network_cell_name*. To move the application to the `APP_INSTALL_ROOT`/*network_cell_name* location upon running the addNode operation, explicitly specify the installation directory as `${APP_INSTALL_ROOT}`/`${CELL)` during installation. In such a case, the application files can always be found under `APP_INSTALL_ROOT`/*current_cell_name*.

You can specify an absolute path or use a pathmap variable such as `${MY_APPS}`. You can use a pathmap variable in any installation.

A pathmap variable is particularly needed when installing an application on a cluster with members on heterogeneous nodes because, in such cases, there might not be a single way to specify an absolute path. A WebSphere Application Server variable `${CELL}` that denotes the current cell name can also be in the pathmap variable; for example, `${MY_APP}`/`${CELL}`. You can define WebSphere Application Server variables on the WebSphere Variables console page, accessed by clicking **Environment > WebSphere Variables**.

This **Directory to install application** field is the same as the **Location (full path)** setting on an Application binaries page.

Data type	String
Units	Full path name

Distribute application:

Specifies whether the product expands application binaries in the installation location during installation and deletes application binaries during uninstallation. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified. The binaries are deleted when you uninstall and save changes to the configuration, and, on the Network Deployment product, synchronize changes.

If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

Important: If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you.

This **Distribute application** field is the same as the **Enable binary distribution, expansion and cleanup post uninstallation** setting on an Application binaries page.

Data type	Boolean
Default	true

Use binary configuration:

Specifies whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the `deployment.xml` file (default), or those located in the enterprise application resource (EAR) file. Select this setting for applications installed on 6.x deployment targets only. This setting is not valid for applications installed on 5.x deployment targets.

This **Use binary configuration** field is the same as the **Use configuration information in binary** setting on an Application binaries page.

Data type	Boolean
Default	false

Deploy enterprise beans:

Specifies whether the EJBDeploy tool runs during application installation.

The tool generates code needed to run enterprise bean (EJB) files. You must enable this setting in the following situations:
* The EAR file was assembled using an assembly tool such as Rational Application Developer, Rational Web Developer or Application Server Toolkit (AST) and the EJBDeploy tool was not run during assembly.
* The EAR file was not assembled using an assembly tool such as Rational Application Developer, Rational Web Developer or AST.
* The EAR file was assembled using versions of the Application Assembly Tool (AAT) previous to Version 5.

For this option, install only onto a 6.1 deployment target.

If you select **Deploy enterprise beans** and try installing your application onto an earlier deployment target such as version 5.x, the installation is rejected. You can deploy applications to only those targets that have same WebSphere version as the deployment manager. If applications are targeted to servers that have an earlier version than the deployment manager, then you cannot deploy to those targets.

Also, if you select **Deploy enterprise beans** and specify a database type on the **Provide options to perform the EJB Deploy** panel, previously defined backend IDs for all of the EJB modules are overwritten by the chosen database type. To enable backend IDs for individual EJB modules, set the database type to ″″ (null) on the **Provide options to perform the EJB Deploy** panel.

The default database type is `DB2UDB_V81`.

Enabling this setting might cause the installation program to run for several minutes.

Data type   Boolean
Default   true

Application name:

Specifies a logical name for the application. An application name must be unique within a cell and cannot contain an unallowed character.

An application name cannot begin with a period (**.**), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

Unallowed characters		
/ forward slash	**$** dollar sign	**'** single quote mark
**** backslash	**=** equal sign	**″** double quote mark
* asterisk	**%** percent sign	**|** vertical bar
, comma	**+** plus sign	**<** left angle bracket
: colon	**@** at sign	**>** right angle bracket
; semi-colon	**#** hash mark	**&** ampersand (and sign)
? question mark	**]]>** No specific name exists for this character combination	

This **Application name** field is the same as the **Name** setting on an Enterprise application settings page.

Data type   String

Create MBeans for resources:

Specifies whether to create MBeans for resources such as servlets or JSP files within an application when the application starts. The default is to create MBeans.

This field is the same as the **Create MBeans for resources** setting on a Startup behavior page.

Data type   Boolean
Default   true

Enable class reloading:

Specifies whether the WebSphere Application Server run time detects changes to application classes when the application is running. If this setting is enabled and if application classes are changed, then the application is stopped and restarted to reload updated classes.

The default is not to enable class reloading.

This **Enable class reloading** field is the same as the **Reload classes when application files are updated** setting on an Class loading and update detection page.

Data type	Boolean
Default	false

Reload interval in seconds:

Specifies the number of seconds to scan the application's file system for updated files. The default is the value of the reloading interval attribute in the IBM extension (`META-INF/ibm-application-ext.xmi`) file of the EAR file.

The reloading interval attribute takes effect only if class reloading is enabled.

To enable reloading, specify a value greater than zero (for example, 1 to 2147483647). To disable reloading, specify zero (0). The range is from 0 to 2147483647.

This **Reload interval in seconds** field is the same as the **Polling interval for updated files** setting on an Class loading and update detection page.

Data type	Integer
Units	Seconds
Default	3

Deploy Web services:

Specifies whether the Web services deploy tool `wsdeploy` runs during application installation.

The tool generates code needed to run applications using Web services. The default is not to run the `wsdeploy` tool. You must enable this setting if the EAR file contains modules using Web services and has not previously had the `wsdeploy` tool run on it, either from the **Deploy** menu choice of an assembly tool or from a command line.

For this option, install only onto a 6.1 deployment target.

If you select **Deploy Web services** and try installing your application onto an earlier deployment target such as version 5.x, the installation is rejected. You can deploy applications to only those targets that have same WebSphere version as the deployment manager. If applications are targeted to servers that have an earlier version than the deployment manager, then you cannot deploy to those targets.

Data type	Boolean
Default	false

Validate input off/warn/fail:

Specifies whether WebSphere Application Server examines the application references specified during application installation or updating and, if validation is enabled, warns you of incorrect references or fails the operation.

An application typically refers to resources using data sources for container managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the application is defined in the scope of the deployment target of that application.

Select **off** for no resource validation, **warn** for warning messages about incorrect resource references, or **fail** to stop operations that fail as a result of incorrect resource references.

This **Validate input off/warn/fail** field is the same as the **Application reference validation** setting on an Enterprise Application settings page.

Data type   String
Default   warn

Process embedded configuration:

Specifies whether the embedded configuration should be processed. An embedded configuration consists of files such as `resource.xml` and `variables.xml`. When selected or `true`, the embedded configuration is loaded to the application scope from the `.ear` file. If the `.ear` file does not contain an embedded configuration, the default is `false`. If the `.ear` file contains an embedded configuration, the default is `true`.

Data type   Boolean
Default   false

File permission:

Specifies access permissions for application binaries for installed applications that are expanded to the directory specified.

The **Distribute application** option must be enabled to specify file permissions.

You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the drop-down list. Drop-down list selections overwrite file permissions set in the text field.

You can set one or more of the following file permission strings in the drop-down list. Selecting multiple options combines the file permission strings.

Drop-down list option	File permission string set
Allow all files to be read but not written to	`.*=755`
Allow executables to execute	`.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755`
Allow HTML and image files to be read by everyone	`.*\.htm=755#.*\.html=755#.*\.gif=755#.*\.jpg=755`

Instead of using the drop-down list to specify file permissions, you can specify a file permission string in the text field. File permissions use a string that has the following format:

file_name_pattern=permission#file_name_pattern=permission

where *file_name_pattern* is a regular expression file name filter (for example, `.*\\.jsp` for all JSP files), *permission* provides the file access control lists (ACLs), and *#* is the separator between multiple entries of *file_name_pattern* and *permission*. If *#* is a character in a *file_name_pattern* string, use `\#` instead.

If multiple file name patterns and file permissions in the string match a uniform resource identifier (URI) within the application, then the product uses the most stringent applicable file permission for the file. For example, if the file permission string is `.*\\.jsp=775#a.*\\.jsp=754`, then the `abc.jsp` file has file permission 754.

Tip: Using regular expressions for file matching pattern compares an entire string URI against the specified file permission pattern. You must provide more precise matching patterns using regular expressions as defined by Java programming API. For example, suppose the following directory and file URIs are processed during a file permission operation:

1	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war
2	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
3	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF/MANIFEST.MF
4	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/WEB-INF/classes/MyClass.class
5	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/mydir/MyClass2.class
6	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF

The file pattern matching results are:
- `MyWarModule.war` does not match any of the URIs
- `.*MyWarModule.war.*` matches all URIs
- `.*MyWarModule.war$` matches only URI 1
- `.*\\.jsp=755` matches only URI 2
- `.*META-INF.*` matches URIs 3 and 6
- `.*MyWarModule.war/.*/.*\.class` matches URIs 4 and 5

If you specify a directory name pattern for **File permissions**, then the directory permission is set based on the value specified. Otherwise, the **File permissions** value set on the directory is the same as its parent. For example, suppose you have the following file and directory structure:

`/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp`

and you specify the following file pattern string:

`.*MyApp.ear$=755#.*\.jsp=644`

The file pattern matching results are:
- Directory `MyApp.ear` is set to 755
- Directory `MyWarModule.war` is set to 755
- Directory `MyWarModule.war` is set to 755

Important: Regardless of the operation system, always use a forward slash (/) as a file path separator in file patterns.

▶ Windows ◀ You cannot unset read permission on a file on Windows platforms. With POSIX style permission bits, the bit for denoting `readable` on a file is 4, `writable` is 2, and `executable` is 1. Thus, permission of a file on a Windows platform is either 5 or 7. Also, in POSIX style there are `user`, `group` and `world` permissions. You can only set the `user` permission for a file on Windows platforms. The `group` and `world` permission bits are ignored.

Access permissions specified here are at the application level. You can also specify access permissions for application binaries in the node level configuration. The node level file permissions specify the

maximum (most lenient) permissions that can be given to application binaries. Access permissions specified here at application level can only be the same as or more restrictive than those specified at the node level.

This setting is the same as the **File permissions** field on the Application binaries page.

Data type	String

Application build identifier:

Specifies an uneditable string that identifies the build version of the application.

This **Application build identifier** field is the same as the **Application build level** field on the Application binaries page.

Data type	String

Allow dispatching includes to remote resources:

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

This field is the same as the **Allow dispatching includes to remote resources** field on the Remote request dispatch properties page.

Data type	Boolean
Default	true

Allow servicing includes from remote resources:

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

This field is the same as the **Allow servicing includes from remote resources** field on the Remote request dispatch properties page.

Data type	Boolean
Default	true

Provide options to perform the EJB Deploy settings

Use this panel to specify options for the enterprise bean (EJB) deployment tool. The tool generates code needed to run enterprise bean files. You can specify extra class paths, Remote Method Invocation compiler (RMIC) options, database types, and database schema names to be used while running the EJB deployment tool.

This administrative console panel is a step in the application installation and update wizards. To view this panel, you must select **Deploy enterprise beans** on the **Select installation options** panel. Thus, to view this panel, click **Applications > Install New Application >** *application_path* **> Show me all installation options and parameters > Next > Next > Deploy enterprise beans > Next > Step: Provide options to perform the EJB Deploy**.

You can specify the EJB deployment tool options on this panel only when installing or updating an application that contains EJB modules.

The options that you specify set parameter values for the `ejbdeploy` command. The tool, and thus the `ejbdeploy` command, is run on the enterprise archive (EAR) file during installation after you click **Finish** on the **Summary** panel of the wizard.

Deploy EJB option - Class path:

Specifies the class path of one or more zipped or Java archive (JAR) files on which the Java archive (JAR) or EAR file being installed depends.

To specify the class paths of multiple zipped and JAR files, the zipped and JAR file names must be fully qualified, separated by semicolons, and enclosed in double quotation marks. For example:

`path\myJar1.jar;path\myJar2.jar;path\myJar3.jar`

Deploy EJB option - Class path is the same as the `ejbdeploy` command parameter `-cp class_path`.

Data type	String
Default	null

Deploy EJB option - RMIC:

Specifies whether the EJB deployment tool passes RMIC options to the Remote Method Invocation compiler. Refer to RMI Tools documentation for information on the options.

Separate options by a space and enclose them in double quotation marks. For example:

`"-nowarn -verbose"`

Deploy EJB option - RMIC is the same as the `ejbdeploy` command parameter `-rmic "options"`.

Data type	String
Default	null

Deploy EJB option - Database type:

Specifies the name of the database vendor, which is used to determine database column types, mapping information, `Table.sql`, and other information. Select a database type or the empty choice from the drop-down list. The list contains the names of valid database vendors. Selecting the empty choice sets the database type to ″″ (null).

If you specify a database type, previously defined backend IDs for all of the EJB modules are overwritten by the chosen database type. To enable backend IDs for individual EJB modules, select the empty choice to set the database type to null.

The backend IDs SQL92 (1992 SQL Standard) and SQL99 (1999 SQL Standard) are deprecated. Although the `SQL92` and `SQL99` backend IDs are available in the list, they are deprecated.

Deploy EJB option - Database type is the same as the `ejbdeploy` command parameter `-dbvendor name`.

Data type	String
Default	DB2UDB_V82

Deploy EJB option - Database schema:

Specifies the name of the schema that you want to create.

The EJB deployment tool saves database information in the schema document in the JAR or EAR file, which means that the options do not need to be specified again. It also means that when a JAR or EAR is generated, the correct database must be defined at that point because it cannot be changed later.

If the name of the schema contains any spaces, the entire name must be enclosed in double quotes. For example:

`"my schema"`

Deploy EJB option - Database schema is the same as the `ejbdeploy` command parameter `-dbschema` `"name"`.

Data type	String
Default	null

Bind listeners for message-driven beans settings

Use this panel to specify bindings for message-driven beans in your application or module.

To view this administrative console panel, click **Applications > Enterprise Applications >** *application_name* **> Message Driven Bean listener bindings**. This panel is the same as the **Bind listeners for message-driven beans** panel on the application installation and update wizards.

Each message-driven bean must be bound to a listener port name or to an activation specification Java Naming and Directory Interface (JNDI) name.

Provide a listener port name if your application uses any of the following Java Message Service (JMS) providers:
*   Version 5 default messaging
*   WebSphere MQ
*   Generic

Provide an activation specification JNDI name if your application's resources are configured using the default messaging provider or any generic J2C resource adapter that supports inbound messaging.

Not providing valid listener port names or activation specification JNDI names results in the following errors:
*   If neither a listener port name or an activation specification JNDI name is specified for a message driven bean, then a validation error is displayed after you click **Finish** on the **Summary** panel.
*   If the module containing the message-driven bean is deployed on a 5.x deployment target and a listener port is not specified, then a validation error is displayed after you click **Next**.
*   If multiple message driven beans are linked to the same destination, specify the same destination JNDI name for each message driven bean. If you specify different destination JNDI names, a validation error is displayed and all JNDI specifications after the first one are ignored.

To apply binding changes to multiple mappings:
1. In the list of mappings, select the **Select** check box beside each EJB module that you want mapped to a particular binding.
2. Expand **Apply Multiple Mappings**.
3. Specify a listener port name or select a target resource JNDI name for an activation specification.
4. If you are defining a binding for an activation specification, optionally specify the following:
    Destination JNDI name
    > For resource adapters that support JMS, specify javax.jms.Destinations so the resource adapter can service messages from the JMS destination. A destination JNDI name set as part of application deployment take precedence over properties set on an activation specification administrative object.

ActivationSpec authentication alias

Specify an authentication alias that is used to access the user name and password that are set on the configured J2C activation specification. Authentication alias properties set as part of application deployment take precedence over properties set on an activation specification administrative object.

5. Click **Apply**.

6. Click **OK**.

EJB module:

Specifies the name of the module that contains the enterprise bean.

EJB:

Specifies name of an enterprise bean in the application.

URI:

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

Messaging type:

Specifies the type of message-driven bean.

Bindings:

Specifies a listener port name or an activation specification JNDI name for the message-driven bean. When a message-driven enterprise bean is bound to an activation specification JNDI name you can also specify the destination JNDI name and the authentication alias.

Bindings specify JNDI names for the referenceable and referenced artifacts in an application. An example JNDI name for a listener port to be used by a `Store` application might be `StoreMdbListener`. The binding definition is stored in IBM bindings files such as `ibm-ejb-jar-bnd.xmi`.

Example: Installing an EAR file using the default bindings

If application bindings were not specified for all enterprise beans or resources in an application during application development or assembly, you can select to generate default bindings. After application installation, you can modify the bindings as needed using the administrative console.

An example of a simple `.ear` file installation using the default bindings follows:

1. Go to the Preparing for application install pages.

   Click **Applications > Install New Application** in the console navigation tree.

2. For **Path to the new application**, specify the full path name of the `.ear` file.

   For this example, the base file name is `my_appl.ear` and the file resides on a server at `C:\sample_apps`.
   a. Select **Remote file system** and click **Browse**.
   b. On the Browse Remote Filesystems page, select the node that runs on the server which holds `my_appl.ear` and the EAR file name.

      Click **C:**, **sample_apps**, **my_appl.ear**, and then **OK**.

3. For **How do you want to install the application**, select **Show me all installation options and parameters**.

4. Click **Next**.

5. On the second Preparing for application installation page, select **Generate default bindings** and click **Next**.

   Using the default bindings causes any incomplete bindings in the application to be filled in with default values. Existing bindings are not changed. By choosing this option, you can skip many of the steps on the Install New Application page and go directly to the Summary step.

6. If application security warnings are displayed, read the warnings and click **Continue**.

7. On the Install New Application page, click step 2, **Manage modules**, and verify the cell, node, and server onto which the application files will install.

   a. On the **Manage modules** panel, select the server onto which the application files will install from the **Clusters and Servers** list, click **Module** to select all of the application modules, and click **Next**.

      On the **Manage modules** panel, you can map modules to other servers such as Web servers. If you want a Web server to serve the application, use the **Ctrl** key to select an application server or cluster and the Web server together in order to have the plug-in configuration file `plugin-cfg.xml` for that Web server generated based on the applications which are routed through it.

8. On the Install New Application page, click the step number beside **Summary**, the last step.

9. On the Summary panel, click **Finish**.

Examine the application installation progress messages. If the application installs successfully, save your administrative configuration. You can now see the name of your application in the list of deployed applications on the Enterprise Applications page accessed by clicking **Applications > Enterprise Applications** in the console navigation tree.

If the application does not install successfully, read the messages to identify why the installation failed. Correct problems with the application as needed and try installing the application again.

Installing J2EE modules with JSR-88

You can install Java 2 Platform, Enterprise Edition (J2EE) modules on an application server provided by a WebSphere Application Server product using the J2EE Deployment API Specification (JSR-88).

JSR-88 defines standard application programming interfaces (APIs) to enable deployment of J2EE applications and stand-alone modules to J2EE product platforms. The J2EE Deployment Specification Version 1.1 is available at http://java.sun.com/j2ee/tools/deployment/reference/docs/index.html as part of the J2EE 1.4 Application Server Developer Release.

Read about JSR-88 and APIs used to manage applications at http://java.sun.com/j2ee/tools/deployment/.

JSR-88 defines a contract between a tool provider and a platform that enables tools from multiple vendors to configure, deploy and manage applications on any J2EE product platform. The tool provider typically supplies software tools and an integrated development environment (IDE) for developing and assembly of J2EE application modules. The J2EE platform provides application management functions that deploy, undeploy, start, stop, and otherwise manage J2EE applications.

WebSphere Application Server is a J2EE 1.4 specification-compliant platform that implements the JSR-88 APIs. Complete the following steps to deploy (install) J2EE modules on an application server provided by the WebSphere Application Server platform.

1. Code a Java program that can access the JSR-88 DeploymentManager class for WebSphere Application Server.

   a. Write code that finds the JAR manifest file key J2EE-DeploymentFactory-Implementation-Class. Under JSR-88, your code finds the DeploymentFactory using the JAR manifest file key J2EE-DeploymentFactory-Implementation-Class. For WebSphere Application Server, the application management JAR file containing this key and providing support is *app_server_root*/lib/

wjmxapp.jar. After your code finds the DeploymentFactory, the deployment tool can create an instance of the WebSphere DeploymentFactory and register the instance with its DeploymentFactoryManager. For example:

```
import javax.enterprise.deploy.shared.factories.DeploymentFactoryManager;
import javax.enterprise.deploy.spi.DeploymentManager;
import javax.enterprise.deploy.spi.factories.DeploymentFactory;
import java.util.jar.JarFile;

// Get the DeploymentFactory implementation class from the MANIFEST.MF file.
JarFile wjmxappJar = new JarFile(new File(wasHome + "/lib/wjmxapp.jar"));
java.util.jar.Manifest manifestFile = wjmxappJar.getManifest();
Attributes attributes = manifestFile.getMainAttributes();
String key = "J2EE-DeploymentFactory-Implementation-Class";
String className = attributes.getValue(key);
// Get an instance of the DeploymentFactoryManager
DeploymentFactoryManager dfm = DeploymentFactoryManager.getInstance();

// Create an instance of the WebSphere Application Server DeploymentFactory.
Class deploymentFactory = Class.forName(className);
DeploymentFactory deploymentFactoryInstance =
    (DeploymentFactory) deploymentFactory.newInstance();

// Register the DeploymentFactory instance with the DeploymentFactoryManager.
dfm.registerDeploymentFactory(deploymentFactoryInstance);

// Provide WebSphere Application Server URL, user ID, and password.
// For more information, see the step that follows.
wsDM = dfm.getDeploymentManager(
    "deployer:WebSphere:myserver:8880", null, null);
```

   b. Write code that accesses the DeploymentManager instance for WebSphere Application Server. The WebSphere Application Server URL for deployment has the format

   `"deployer:WebSphere:host:port"`

   The example in the previous step, `"deployer:WebSphere:myserver:8880"`, tries to connect to host *myserver* at port *8880* using the SOAP connector, which is the default.

   The URL for deployment can have an optional parameter *connectorType*. For example, to use the RMI connector to access *myserver*, code the URL as follows:

   `"deployer:WebSphere:myserver:2809?connectorType=RMI"`

2. **Optional:** Code a Java program that can customize or deploy J2EE applications or modules using the JSR-88 support provided by WebSphere Application Server.

3. Start the deployed J2EE applications or standalone J2EE modules using the JSR-88 API used to start applications or modules.

Test the deployed applications or modules. For example, point a Web browser at the URL for a deployed application and examine the performance of the application. If necessary, update the application.

Customizing modules using DConfigBeans

You can configure J2EE applications or standalone modules during deployment using the DConfigBean class in the Java 2 Platform, Enterprise Edition (J2EE) Deployment API Specification (JSR-88).

This topic assumes that you are deploying (installing) J2EE modules on an application server provided by the WebSphere Application Server platform using the WebSphere Application Server support for JSR-88.

Read about the JSR-88 specification and using the DConfigBean class at http://java.sun.com/j2ee/tools/deployment/.

The DConfigBean class in JSR-88 provides JavaBeans-based support for platform-specific configuration of J2EE applications and modules during deployment. Your code can inspect DConfigBean instances to get

platform-specific configuration attributes. The DConfigBean instances provided by WebSphere Application Server contain a single attribute which has an array of java.util.Hashtable objects. The hashtable entries contain configuration attributes, for which your code can get and set values.

1. Write code that installs J2EE modules on an application server using JSR-88.
2. Write code that accesses DConfigBeans generated by WebSphere Application Server during JSR-88 deployment. You (or a deployer) can then customize the accessed DConfigBeans instances. The following pseudocode shows how a J2EE tool provider can get DConfigBean instance attributes generated by WebSphere Application Server during JSR-88 deployment and set values for the attributes:

```
import javax.enterprise.deploy.model.*;
import javax.enterprise.deploy.spi.*;
{
DeploymentConfiguration dConfig = ___; // Get from DeploymentManager
DDBeanRoot ddRoot = ___;                // Provided by J2EE tool

// Obtain root bean.
DConfigBeanRoot dcRoot = dConfig.getDConfigBeanRoot(dr);

// Configure DConfigBean.
configureDCBean (dcRoot);
}

// Get children from DConfigBeanRoot and configure each child.
method configureDCBean (DConfigBean dcBean)
{
   // Get DConfigBean attributes for a given archive.
   BeanInfo bInfo = Introspector.getBeanInfo(dcBean.getClass());
   IndexedPropertyDescriptor ipDesc =
      (IndexedPropertyDescriptor)bInfo.getPropertyDescriptors()[0];

   // Get the 0th table.
   int index = 0;
   Hashtable tbl = (Hashtable)
      ipDesc.getIndexedReadMethod().invoke
         (dcBean, new Object[]{new Integer(index)});

   while (tbl != null)
   {
      // Iterate over the hashtable and set values for attributes.

      // Set the table back into the DCBean.
      ipDesc.getIndexedWriteMethod().invoke
         (dcBean, new Object[]{new Integer(index), tbl});

      // Get the next entry in the indexed property
      tbl = (Hashtable)
         ipDesc.getIndexedReadMethod().invoke
            (dcBean, new Object[]{new Integer(++index)});
   }
}
```

Enterprise application collection

Use this page to view and manage enterprise applications.

This page lists installed enterprise applications. System applications, which are central to the product, are not shown in the list because users cannot edit them. Examples of system applications include *isclite*, *managementEJB* and *filetransfer*.

To view this administrative console page, click **Applications > Enterprise Applications**.

To view the values specified for an application's configuration, click the application name in the list. The displayed application settings page shows the values specified. On the settings page, you can change existing configuration values and link to additional console pages that assist you in configuring the application.

To manage an installed enterprise application, enable the **Select** check box beside the application name in the list and click a button:

Button	Resulting action
Start	Attempts to run the application. After the application starts up successfully, the state of the application changes to *Started* if the application starts up on all deployment targets, else the state changes to *Partial Start*.
Stop	Attempts to stop the processing of the application. After the application stops successfully, the state of the application changes to *Stopped* if the application stops on all deployment targets, else the state changes to *Partial Stop*.
Install	Opens a wizard that helps you deploy an application or a module such as a `.jar`, `.war` or `.rar` file onto a server or a cluster.
Uninstall	Deletes the application from the WebSphere Application Server configuration repository and deletes the application binaries from the file system of all nodes where the application modules are installed after the configuration is saved and synchronized with the nodes.
Update	Opens a wizard that helps you update application files deployed on a server. You can update the full application, a single module, a single file, or part of the application. If a new file or module has the same name as a file or module already existing on the server, the new file or module replaces the existing file or module. If the new file or module does not exist on the server, it is added to the deployed application.
Rollout Update	Sequentially updates an application installed on multiple cluster members across a cluster. After you update an application's files or configuration, click **Rollout Update** to install the application's updated files or configuration on all cluster members of a cluster on which the application is installed. **Rollout Update** does the following for each cluster member in sequence: 1. Saves the updated application configuration. 2. Stops all of the cluster members on one node. 3. Updates the application on the node by synchronizing the configuration. 4. Restarts the stopped cluster members. 5. Repeats steps 2 through 4 for all of the nodes that have cluster members. Use **Rollout Update** if the application is deployed on one or more clusters spread across multiple nodes. This action reduces the amount of time that any single cluster member is unavailable to serve requests to the smallest interval possible. Pending IIOP transactions will complete before a cluster member stops; in-flight HTTP and JMS transactions might be lost while the cluster member is stopping. For an application server without clusters, use **Update** and then save and synchronize the node instead. For a standalone application server, simply update and save.
Remove File	Deletes a file of the deployed application or module. **Remove File** deletes a file from the configuration repository and from the file system of all nodes where the file is installed. If the application or module is deployed on a cluster, after removing a file click **Rollout Update** to roll out the changes across the entire cluster.
Export	Accesses the Export Application EAR files page, which you use to export an enterprise application to an EAR file at a location of your choice. Use the **Export** action to back up a deployed application and to preserve its binding information.
Export DDL	Accesses the Export Application DDL files page, which you use to export DDL files (`Table.ddl`) in the EJB modules of an enterprise application to a location of your choice.

These buttons are not available when this page is accessed from an application server settings page. When this page is accessed from an application server settings page, it is entitled the Installed applications page.

Name

Specifies the name of the installed (or deployed) application. Application names must be unique within a cell and cannot contain an unallowed character.

Application Status

Indicates whether the application deployed on the application server is started, stopped, or unavailable.

	Started	Application is running.
	Started	Application is running.
	Partial Start	Application is in the process of changing from a *Stopped* state to a *Started* state. Application is starting to run but is not fully running yet. Or, it cannot fully start because a server mapped to one or more application modules is stopped.
	Stopped	Application is not running.
	Partial Stop	Application is in the process of changing from a *Started* state to a *Stopped* state. Application has not stopped running yet.
	Unavailable	Status cannot be determined.
	Not applicable	Application does not provide information as to whether it is running.

Startup order

Specifies the order in which applications are started when the server starts. The application with the lowest startup order is started first.

This table column is available only when this page is accessed from an application server settings page; thus when this page is entitled the Installed applications page.

Enterprise application settings

Use this page to configure an enterprise application.

To view this administrative console page, click **Applications > Enterprise Applications >** *application_name*.

Name

Specifies a logical name for the application. An application name must be unique within a cell and cannot contain an unallowed character.

An application name cannot begin with a period (**.**), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

Unallowed characters					
/	forward slash	**$**	dollar sign	**'**	single quote mark
****	backslash	**=**	equal sign	**″**	double quote mark
*	asterisk	**%**	percent sign	**|**	vertical bar
,	comma	**+**	plus sign	**<**	left angle bracket
:	colon	**@**	at sign	**>**	right angle bracket
;	semi-colon	**#**	hash mark	**&**	ampersand (and sign)

Unallowed characters	
? question mark	**]]>** No specific name exists for this character combination

Data type   String

Application reference validation

Specifies whether the product examines the application references specified during application installation or updating and, if validation is enabled, warns you of incorrect references or fails the operation.

An application typically refers to resources using data sources for container managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the application is defined in the scope of the deployment target of that application.

The resource can be defined on the server, its node, cell or the cluster if the server belongs to a cluster. Select **Don't validate** for no resource validation, **Issue warnings** for warning messages about incorrect resource references, or **Stop installation if validation fails** to stop operations that fail as a result of incorrect resource references.

This **Application reference validation** setting is the same as the **Validate input off/warn/fail** field on the application installation and update wizards.

Data type   String
Default   Issue warnings

Configuring an application

You can change the configuration of an application or module deployed on a server.

You can change the contents of and deployment descriptors for an application or module before deployment, such as in an assembly tool. However, it is assumed that the module is already deployed on a server.

Changing an application or module configuration consists of one or more of the following:
* Changing the settings of the application or module.
* Removing a file from an application or module.
* Updating the application or its modules.

This topic describes how to change the settings of an application or module using the administrative console.
* View current settings of the application or module.

  Click **Applications > Enterprise Applications >** *application_name* to access the settings page for the enterprise application.

  Many application or module settings are available on other console pages that you can access by clicking links on the settings page for the enterprise application. For detailed information on the settings and allowed values, examine the online help for the console pages. When you installed the application or module, you specified most of the settings values.
* Map each module of your application to a target server.

  Specify the application servers, clusters of application servers, or Web servers onto which to install modules of your application.
* Change how quickly your application starts compared to other applications or to the server.
* Configure the use of binary files.

- Change how your application or Web modules use class loaders.
- Map a virtual host for each Web module of your application. Configuring virtual hosts provides information on virtual hosts.
- Change application bindings or other settings of the application or module.
  1. Click **Applications > Enterprise Applications >** *application_name* **>** *property_or_item_name* in the console navigation tree. From the application settings page, you can access console pages for further configuring of the application or module.
     - Target specific application status
     - Remote request dispatch properties
     - Security role to user/group mapping
     - View deployment descriptor
     - Last participant support extension
     - Application scope resources
     - Resource references
     - EJB references
     - Shared library references
     - Initial parameters for servlets
     - Session management
     - Context root for Web modules
     - JSP reloading options for Web modules
     - Environment entries for Web modules
     - Virtual hosts
     - Stateful session bean failover (applications)
     - Stateful session bean failover (EJB modules)
     - Application profiles
     - 2.x CMP bean data sources
     - 2.x entity bean data sources.
     - EJB JNDI names
     - Correct use of system identity
     - Provide JMS and EJB endpoint URL information
     - Publish WSDL files
     - Provide HTTP endpoint URL information
     - Web modules
     - EJB modules
  2. Change the values for settings as needed, and click **OK**.
- **Optional:** Configure the application so it does not start automatically when the server starts. By default, an installed application starts when the server on which the application resides starts. You can configure the target mapping for the application so the application does not start automatically when the server starts. To start the application, you must then start it manually.
- If the installed application or module uses a resource adapter archive (RAR file), ensure that the **Classpath** setting for the RAR file enables the RAR file to find the classes and resources that it needs. Examine the **Classpath** setting on the console Resource adapter settings page.

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Save changes to your administrative configuration.

In the Network Deployment product, the application binaries are transferred to nodes when the configuration changes on the deployment manager synchronize with configurations for individual nodes on which the application will run.

Application bindings

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 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. An example referenceable artifact is an EJB defined in an application. An example referenced artifact is an EJB or a resource reference used by the application. Binding definitions are stored in the `ibm-xxx-bnd.xmi` files of an application. The *xxx* can be `ejb-jar`, `web`, `application` or `application-client`.

This topic provides the following information about bindings:
- "Times when bindings can be defined"
- "Required bindings" on page 1397
- "Other bindings that might be needed" on page 1400

Times when bindings can be defined

You can define bindings at the following times:
- During application development

  An application developer can create binding definitions in `ibm-xxx-bnd.xmi` files using a tool such as an IBM Rational developer tool. The developer then gives an enterprise application (`.ear` file) complete with bindings to an application assembler or deployer. When assembling the application, the assembler does not modify the bindings. Similarly, when installing the application onto a server supported by WebSphere Application Server, the deployer does not modify or override the bindings or generate default bindings unless changes to the bindings are necessary for successful deployment of the application.

- During application assembly

  An application assembler can define bindings when modifying deployment descriptors of an application. Bindings are specified in the **WebSphere Bindings** section of a deployment descriptor editor. Modifying the deployment descriptors might change the binding definitions in the `ibm-xxx-bnd.xmi` files created when developing an application. After defining the bindings, the assembler gives the application to a deployer. When installing the application onto a server supported by WebSphere Application Server, the deployer does not modify or override the bindings or generate default bindings unless changes to the bindings are necessary for successful deployment of the application.

- During application installation

  An application deployer or server administrator can modify the bindings when installing the application onto a server supported by WebSphere Application Server using the administrative console. New binding definitions can be specified on the install wizard pages.

  If the deployer or administrator selects to override any existing bindings or to generate default bindings during application installation, default bindings are assigned to the application and new bindings might need to be specified using the console.

  Selecting **Generate Default Bindings** during application installation causes any incomplete bindings in the application to be filled in with default values. Existing bindings are not changed.

  Restriction: Bindings can be defined or overridden during application installation for all modules except application clients. For clients, you must define bindings for application client modules during assembly and store the bindings in the `ibm-application-client-bnd.xmi` file.

- During configuration of an installed application

  After an application is installed onto a server supported by WebSphere Application Server, an application deployer or server administrator can modify the bindings by changing values in administrative console pages such as those accessed from the settings page for the enterprise application.

Required bindings

Before an application can be successfully deployed, bindings must be defined for references to the following artifacts:

EJB JNDI names

For each enterprise bean (EJB), you must specify a JNDI name. The name is used to bind an entry in the global JNDI name space for the EJB home object. An example JNDI name for a *Product* EJB in a *Store* application might be `store/ejb/Product`. The binding definition is stored in the `META-INF/ibm-ejb-jar-bnd.xmi` file.

If a deployer chooses to generate default bindings when installing the application, the install wizard assigns EJB JNDI names having the form `prefix/EJB_name` to incomplete bindings. The default prefix is `ejb`, but can be overridden. The *EJB_name* is as specified in the deployment descriptor `<ejb-name>` tag.

During and after application installation, EJB JNDI names can be specified on the Provide JNDI names for beans panel. After installation, click **Applications > Enterprise Applications >** *application_name* **> EJB JNDI names** in the administrative console.

Data sources for entity beans

Entity beans such as container-managed persistence (CMP) beans store persistent data in data stores. With CMP beans, an EJB container manages the persistent state of the beans. You specify which data store a bean uses by binding an EJB module or an individual EJB to a data source. Binding an EJB module to a data source causes all entity beans in that module to use the same data source for persistence.

An example JNDI name for a *Store* data source in a *Store* application might be `store/jdbc/store`. The binding definition is stored in IBM binding files such as `ibm-ejb-jar-bnd.xmi`. A deployer can also specify whether authentication is handled at the container or application level.

If a deployer chooses to generate default bindings when installing the application, the install wizard generates the following for incomplete bindings:
- For EJB 2.x `.jar` files, connection factory bindings based on the JNDI name and authorization information specified
- For EJB 1.1 `.jar` files, data source bindings based on the JNDI name, data source user name and password specified

The generated bindings provide default connection factory settings for each EJB 2.x `.jar` file and default data source settings for each EJB 1.1 `.jar` file in the application being installed. No bean-level connection factory bindings or data source bindings are generated unless they are specified in the custom strategy rule supplied during default binding generation.

During and after application installation, data sources can be mapped to 2.x entity beans on the 2.x CMP bean data sources panel and on the 2.x entity bean data sources panel. After installation, click **Applications > Enterprise Applications >** *application_name* in the administrative console, then select **2.x CMP bean data sources** or **2.x entity bean data sources**. Data sources can be mapped to 1.x entity beans on the Map data sources for all 1.x CMP beans panel and on the Provide default data source mapping for modules containing 1.x entity beans panel. After installation, access console pages similar to those for 2.x CMP beans, except click links for 1.x CMP beans.

Backend ID for EJB modules

If an EJB `.jar` file that defines CMP beans contains mappings for multiple backend databases, specify the appropriate backend ID that determines which persister classes are loaded at run time.

Specify the backend ID during application installation. You cannot select a backend ID after the application is installed onto a server.

To enable backend IDs for individual EJB modules:

1. During application installation, select **Deploy enterprise beans** on the Select installation options panel. Selecting **Deploy enterprise beans** enables you to access the **Provide options to perform the EJB Deploy** panel.

2. On the **Provide options to perform the EJB Deploy** panel, set the database type to *""* (null).

During application installation, if you select **Deploy enterprise beans** on the Select installation options panel and specify a database type for the EJB deployment tool on the **Provide options to perform the EJB Deploy** panel, previously defined backend IDs for all of the EJB modules are overwritten by the chosen database type.

The default database type is `DB2UDB_V81`.

For information on backend databases, refer to EJB deployment tool. For information on EJB Deploy options, refer to The ejbdeploy command.

EJB references

An enterprise bean (EJB) reference is a logical name used to locate the home interface of an enterprise bean. EJB references are specified during deployment. At run time, EJB references are bound to the physical location (global JNDI name) of the enterprise beans in the target operational environment. EJB references are made available in the `java:comp/env/ejb` Java naming subcontext.

For each EJB reference, you must specify a JNDI name. An example JNDI name for a *Supplier* EJB reference in a *Store* application might be `store/ejb/Supplier`. The binding definition is stored in IBM binding files such as `ibm-ejb-jar-bnd.xmi`. When the referenced EJB is also deployed in the same application server, you can specify a server-scoped JNDI name. But if the referenced EJB is deployed on a different application server or if `ejb-ref` is defined in an application client module, then you should specify the global cell-scoped JNDI name.

If a deployer chooses to generate default bindings when installing the application, the install wizard binds EJB references as follows: If an `<ejb-link>` is found, it is honored. If the `ejb-name` of an EJB defined in the application matches the `ejb-ref` name, then that EJB is chosen. Otherwise, if a unique EJB is found with a matching home (or local home) interface as the referenced bean, the reference is resolved automatically.

During and after application installation, EJB reference JNDI names can be specified on the Map EJB references to beans panel. After installation, click **Applications > Enterprise Applications >** *application_name* **> EJB references** in the administrative console.

For more information, refer to "EJB references" on page 198.

Resource references

A resource reference is a logical name used to locate an external resource for an application. Resource references are specified during deployment. At run time, the references are bound to the physical location (global JNDI name) of the resource in the target operational environment. Resource references are made available as follows:

Resource reference type	Subcontext declared in
Java DataBase Connectivity (JDBC) data source	`java:comp/env/jdbc`
JMS connection factory	`java:comp/env/jms`
JavaMail connection factory	`java:comp/env/mail`

| Uniform Resource Locator (URL) connection factory | `java:comp/env/url` |

For each resource reference, you must specify a JNDI name. If a deployer chooses to generate default bindings when installing the application, the install wizard generates resource reference bindings derived from the `<res-ref-name>` tag, assuming that the `java:comp/env` name is the same as the resource global JNDI name.

During application installation, resource reference JNDI names can be specified on the Map resource references to references panel. Specify JNDI names for the resources that represent the logical names defined in resource references. You can optionally specify login configuration name and authentication properties for the resource. After specifying authentication properties, click **OK** to save the values and return to the mapping step. Each resource reference defined in an application must be bound to a resource defined in your WebSphere Application Server configuration. After installation, click **Applications > Enterprise Applications >** *application_name* **> Resource references** in the administrative console to access the Resource references panel.

Virtual host bindings for Web modules

You must bind each Web module to a specific virtual host. The binding informs a Web server plug-in that all requests that match the virtual host must be handled by the Web application. An example virtual host to be bound to a *Store* Web application might be `store_host`. The binding definition is stored in IBM binding files such as `WEB-INF/ibm-web-bnd.xmi`.

If a deployer chooses to generate default bindings when installing the application, the install wizard sets the virtual host to `default_host` for each `.war` file.

During and after application installation, you can map a virtual host to a Web module defined in your application. On the Map virtual hosts for Web modules panel, specify a virtual host. The port number specified in the virtual host definition is used in the URL that is used to access artifacts such as servlets and JSP files in the Web module. For example, an external URL for a Web artifact such as a JSP file is `http://host_name:virtual_host_port/context_root/jsp_path`. After installation, click **Applications > Enterprise Applications >** *application_name* **> Virtual hosts** in the administrative console.

Message-driven beans

For each message-driven bean, you must specify a queue or topic to which the bean will listen. A message-driven bean is invoked by a Java Messaging Service (JMS) listener when a message arrives on the input queue that the listener is monitoring. A deployer specifies a listener port or JNDI name of an activation specification as defined in a connector module (`.rar` file) under **WebSphere Bindings** on the **Beans** page of an assembly tool EJB deployment descriptor editor. An example JNDI name for a listener port to be used by a `Store` application might be `StoreMdbListener`. The binding definition is stored in IBM bindings files such as `ibm-ejb-jar-bnd.xmi`.

If a deployer chooses to generate default bindings when installing the application, the install wizard assigns JNDI names to incomplete bindings.
- For EJB 2.x message-driven beans deployed as JCA 1.5-compliant resources, the install wizard assigns JNDI names corresponding to activationSpec instances in the form `eis/MDB_ejb-name`.
- For EJB 2.x message-driven beans deployed against listener ports, the listener ports are derived from the message-driven bean `<ejb-name>` tag with the string `Port` appended.

During application installation using the administrative console, you can specify a listener port name or an activation specification JNDI name for every message-driven bean on the panel **Bind listeners for message-driven beans**. A listener port name must be provided when using the JMS providers: Version 5 default messaging, WebSphere MQ, or generic. An activation specification 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. If neither is specified, then a validation error is displayed after you click **Finish** on the Summary panel. Also, if

the module containing the message-driven bean is deployed on a 5.x deployment target and a listener port is not specified, then a validation error is displayed after you click **Next**.

After application installation, you can specify JNDI names and configure message-driven beans on console pages under **Resources > JMS Providers** or under **Resources > Resource Adapters**. For more information, refer to "Using asynchronous messaging" on page 804.

Message destination references

A message destination reference is a logical name used to locate an enterprise bean in an EJB module that acts as a message destination. Message destination references exist only in J2EE 1.4 artifacts such as--
- J2EE 1.4 application clients
- EJB 2.1 projects
- 2.4 Web applications

If multiple message destination references are associated with a single message destination link, then a single JNDI name for an enterprise bean that maps to the message destination link, and in turn to all of the linked message destination references, is collected during deployment. At run time, the message destination references are bound to the administered message destinations in the target operational environment.

If a message destination reference and a message-driven bean are linked by the same message destination, both the reference and the bean should have the same destination JNDI name. When both have the same name, only the destination JNDI name for the message-driven bean is collected and applied to the corresponding message destination reference.

If a deployer chooses to generate default bindings when installing the application, the install wizard assigns JNDI names to incomplete message destination references as follows: If a message destination reference has a `<message-destination-link>`, then the JNDI name is set to `ejs/message-destination-linkName`. Otherwise, the JNDI name is set to `eis/message-destination-refName`.

Other bindings that might be needed

Depending on the references in and artifacts used by your application, you might need to define bindings for references and artifacts not listed in this article.

Configuring application startup

You can configure the startup behavior of an application. The values set affect how quickly an application starts and what occurs when an application starts.

This topic assumes that your application or module is already deployed on a server.

This topic also assumes that your application or module is configured to start automatically when the server starts. By default, an installed application starts when the server on which the application resides starts.

This topic describes how to change the settings of an application or module using the administrative console.

1. Click **Applications > Enterprise Applications >** *application_name* **> Startup behavior** in the console navigation tree.
2. Specify the startup order for the application.

   If your application starts automatically when its server starts, the value for **Startup order** controls how quickly the application starts. **Startup order** specifies the order in which applications are started when the server starts. The application with the lowest startup order, or starting weight, is started first.
3. Specify whether the application must initialize fully before its server is considered started.

If your application starts automatically when its server starts, **Launch application before server completes startup** specifies whether the application must initialize fully before its server is considered started. Background applications can be initialized on an independent thread, thus allowing the server startup to complete without waiting for the application. This setting applies only if the application is run on a Version 6 (or later) application server.

4. Specify whether to create MBeans for resources such as servlets or JavaServer Pages (JSP) files within an application when the application starts.

   The default for **Create MBeans for resources** is to create MBeans.

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Save changes to your administrative configuration.

The application binaries are transferred to nodes when the configuration changes on the deployment manager synchronize with configurations for individual nodes on which the application will run.

Startup behavior settings

Use this page to configure when an application starts compared to other applications and to the server, and to configure whether MBeans for resources are created when an application starts.

To view this administrative console page, click **Applications > Enterprise Applications >** *application_name* **> Startup behavior**.

Startup order:

Specifies the order in which applications are started when the server starts. The startup order is like a starting weight. The application with the lowest starting weight is started first.

Data type	Integer
Default	1
Range	0 to 2147483647

Launch application before server completes startup:

Specifies whether the application must initialize fully before the server starts.

The default setting of `false` indicates that server startup will not complete until the application starts.

A setting of `true` informs the product that the application might start on a background thread and thus server startup might continue without waiting for the application to start. Thus, the application might not be ready for use when the application server starts.

This setting applies only if the application is run on a Version 6 application server.

Data type	Boolean
Default	false

Create MBeans for resources:

Specifies whether to create MBeans for various resources (such as servlets or JSP files) within an application when the application starts. The default is to create MBeans.

Data type   Boolean
Default   true

Configuring binary location and use

You can designate where binary files (binaries) used by your application reside, whether the product distributes binaries for you automatically, and otherwise configure the use of binaries.

This topic assumes that your application or module is already deployed on a server.

This topic describes how to change the settings of an application or module using the administrative console.

1. Click **Applications > Enterprise Applications >** *application_name* **> Application binaries** in the console navigation tree. The Application binaries page is displayed.
2. Specify the directory to hold the application binaries.

   The default is `APP_INSTALL_ROOT/`*cell_name*, where the `APP_INSTALL_ROOT` variable is *app_server_root*/`installedApps`. For example:

   `C:\WebSphere\AppServer\profiles\`*profile_name*`\installedApps\`*cell_name*

   Refer to "Application binary settings" on page 1403 for a detailed description of the **Location (full path)** setting.
3. Specify the bindings, extensions, and deployment descriptors that an application server uses.

   By default, an application server uses the bindings, extensions, and deployment descriptors located with the application deployment document, the `deployment.xml` file.

   To specify that the application server use the bindings, extensions, and deployment descriptors located in the application archive (EAR) file, select **Use configuration information in binary**. Select this setting for applications installed on 6.x deployment targets only. This setting is not valid for applications installed on 5.x deployment targets.
4. Specify whether the product distributes application binaries automatically to other nodes on the cell.

   By default, **Enable binary distribution, expansion and cleanup post uninstallation** is selected and binaries are distributed automatically.

   If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

   Important: If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you.
5. Specify access permissions for binaries.
   a. Ensure that the **Enable binary distribution, expansion and cleanup post uninstallation** option is enabled. That option must be enabled to specify access permissions for binaries.
   b. For **File permissions**, specify a string that defines access permissions for binaries that are expanded in the named location.

      You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the drop-down list. Drop-down list selections overwrite file permissions set in the text field.

      For details on **File permissions**, refer to "Application binary settings" on page 1403.
6. Click **OK**.

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Save changes to your administrative configuration.

In the Network Deployment product, the application binaries are transferred to nodes when the configuration changes on the deployment manager synchronize with configurations for individual nodes on which the application will run.

Application binary settings

Use this page to configure the location and distribution of application binary files.

To view this administrative console page, click **Applications > Enterprise Applications >** *application_name* **> Application binaries**.

Location (full path):

Specifies the directory to which the application EAR file is installed. This **Location** setting is the same as the **Directory to install application** field on the application installation and update wizards.

The default value is the value of `APP_INSTALL_ROOT`/*cell_name*, where the `APP_INSTALL_ROOT` variable is *app_server_root*/`installedApps`; for example, *app_server_root*/`installedApps`/*cell_name*.

Important: If an installation directory is not specified when an application is installed on a single-server configuration, the application is installed in `APP_INSTALL_ROOT`/*cell_name*. When the server is made a part of a Network Deployment configuration (using the addNode utility), the cell name of the new configuration becomes the cell name of the deployment manager node. If the `-includeapps` option is used for the addNode utility, then the applications that are installed prior to the addNode operation still use the installation directory `APP_INSTALL_ROOT`/*cell_name*. However, an application that is installed after the server is added to the network configuration uses the default installation directory `APP_INSTALL_ROOT`/*network_cell_name*. To move the application to the `APP_INSTALL_ROOT`/*network_cell_name* location upon running the addNode operation, explicitly specify the installation directory as `${APP_INSTALL_ROOT}/${CELL}` during installation. In such a case, the application files can always be found under `APP_INSTALL_ROOT`/*current_cell_name*.

You can specify an absolute path or use a pathmap variable such as `${MY_APPS}`. You can use a pathmap variable in any installation.

A pathmap variable is particularly needed when installing an application on a cluster with members on heterogeneous nodes because, in such cases, there might not be a single way to specify an absolute path. A WebSphere Application Server variable `${CELL}` that denotes the current cell name can also be in the pathmap variable; for example, `${MY_APP}/${CELL}`. You can define WebSphere Application Server variables on the WebSphere Variables console page, accessed by clicking **Environment > WebSphere Variables**.

Data type	String
Units	Full path name

Use configuration information in binary:

Specifies whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the `deployment.xml` file (default), or those located in the enterprise application resource (EAR) file.

This **Use configuration information in binary** setting is the same as the **Use binary configuration** field on the application installation and update wizards. Select this setting for applications installed on 6.x deployment targets only. This setting is not valid for applications installed on 5.x deployment targets.

Data type   Boolean
Default   false

Enable binary distribution, expansion and cleanup post uninstallation:

Specifies whether the product expands application binaries in the installation location during installation and deletes application binaries during uninstallation. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified. The binaries are deleted when you uninstall and save changes to the configuration, and, on the Network Deployment product, synchronize changes.

If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

Important: If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you.

This **Enable binary distribution, expansion and cleanup post uninstallation** setting is the same as the **Distribute application** field on the application installation and update wizards.

Data type   Boolean
Default   true

File permissions:

Specifies access permissions for application binaries for installed applications that are expanded to the directory specified.

The **Enable binary distribution, expansion and cleanup post uninstallation** option must be enabled to specify file permissions.

You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the drop-down list. Drop-down list selections overwrite file permissions set in the text field.

You can set one or more of the following file permission strings in the drop-down list. Selecting multiple options combines the file permission strings.

Drop-down list option	File permission string set
Allow all files to be read but not written to	.*=755
Allow executables to execute	.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755
Allow HTML and image files to be read by everyone	.*\.htm=755#.*\.html=755#.*\.gif=755#.*\.jpg=755

Instead of using the drop-down list to specify file permissions, you can specify a file permission string in the text field. File permissions use a string that has the following format:

file_name_pattern=permission#file_name_pattern=permission

where *file_name_pattern* is a regular expression file name filter (for example, `.*\\.jsp` for all JSP files), *permission* provides the file access control lists (ACLs), and `#` is the separator between multiple entries of *file_name_pattern* and *permission*. If `#` is a character in a *file_name_pattern* string, use `\#` instead.

If multiple file name patterns and file permissions in the string match a uniform resource identifier (URI) within the application, then the product uses the most stringent applicable file permission for the file. For example, if the file permission string is `.*\\.jsp=775#a.*\\.jsp=754`, then the `abc.jsp` file has file permission 754.

Tip: Using regular expressions for file matching pattern compares an entire string URI against the specified file permission pattern. You must provide more precise matching patterns using regular expressions as defined by Java programming API. For example, suppose the following directory and file URIs are processed during a file permission operation:

1	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war
2	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
3	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF/ MANIFEST.MF
4	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/WEB-INF/classes/ MyClass.class
5	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/mydir/ MyClass2.class
6	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF

The file pattern matching results are:
*   `MyWarModule.war` does not match any of the URIs
*   `.*MyWarModule.war.*` matches all URIs
*   `.*MyWarModule.war$` matches only URI 1
*   `.*\\.jsp=755` matches only URI 2
*   `.*META-INF.*` matches URIs 3 and 6
*   `.*MyWarModule.war/.*/.*\.class` matches URIs 4 and 5

If you specify a directory name pattern for **File permissions**, then the directory permission is set based on the value specified. Otherwise, the **File permissions** value set on the directory is the same as its parent. For example, suppose you have the following file and directory structure:

`/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp`

and you specify the following file pattern string:

`.*MyApp.ear$=755#.*\.jsp=644`

The file pattern matching results are:
*   Directory `MyApp.ear` is set to 755
*   Directory `MyWarModule.war` is set to 755
*   Directory `MyWarModule.war` is set to 755

Important: Regardless of the operation system, always use a forward slash (/) as a file path separator in file patterns.

Windows You cannot unset read permission on a file on Windows platforms. With POSIX style permission bits, the bit for denoting `readable` on a file is 4, `writable` is 2, and `executable` is 1. Thus, permission of a file on a Windows platform is either 5 or 7. Also, in POSIX style there are `user`, `group` and `world` permissions. You can only set the `user` permission for a file on Windows platforms. The `group` and `world` permission bits are ignored.

Access permissions specified here are at the application level. You can also specify access permissions for application binaries in the node level configuration. The node level file permissions specify the maximum (most lenient) permissions that can be given to application binaries. Access permissions specified here at application level can only be the same as or more restrictive than those specified at the node level.

This setting is the same as the **File permission** field on the application installation and update wizards.

Data type   String

Application build level:

Specifies an uneditable string that identifies the build version of the application.

Data type   String

Configuring the use of class loaders by an application

You can configure whether your application and Web modules use their own class loaders to load classes or use different class loaders, as well as configure the reloading of classes when application files are updated. Class loaders enable an application to access repositories of available classes and resources.

This topic assumes that your application or module is already deployed on a server.

Selection of class loaders to be used by an application and Web modules affects whether your application and its modules find the resources that they need to run effectively. You can select whether your application and Web modules use their own class loaders to load classes, or use a parent class loader. Detailed information on class loaders is available in "Class loaders" on page 1351, Chapter 8, "Class loading," on page 1351 and Troubleshooting class loaders.

An application class loader groups enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Dependency JAR files are JAR files that contain code which can be used by both enterprise beans and servlets.

An application class loader is the parent of a Web application archive (WAR) class loader. By default, a Web module has its own WAR class loader to load the contents of the Web module. The WAR class-loader policy value of an application class loader determines whether the WAR class loader or the application class loader is used to load the contents of the Web module.

You can also select whether classes are reloaded when application files are updated. For enterprise bean (EJB) modules or any non-Web modules, enabling class reloading causes the application server run time to stop and start the application to reload application classes. For Web modules such as servlets and JavaServer Pages (JSP) files, a Web container reloads a Web module only when the IBM extension reloadingEnabled in the `ibm-web-ext.xmi` file is set to `true`.

To configure use of class loaders by your application and Web modules, use the Class loading and update detection page of the administrative console.

1. Click **Applications > Enterprise Applications >** *application_name* **> Class loading and update detection** to access the settings page for an application class loader.
2. Specify whether to reload application classes when the application or its files are updated.

   By default, class reloading is not enabled. Select **Reload classes when application files are updated** to choose to reload application classes. You might specify different values for EJB modules and for Web modules such as servlets and JavaServer Pages (JSP) files.
3. Specify the number of seconds to scan the application's file system for updated files.

The value specified for **Polling interval for updated files** takes effect only if class reloading is enabled. The default is the value of the reloading interval attribute in the IBM extension (`META-INF/ibm-application-ext.xmi`) file of the enterprise application (EAR file). You might specify different values for EJB modules and for Web modules such as servlets and JSP files.

To enable reloading, specify an integer value that is greater than zero (for example, 1 to 2147483647).

To disable reloading, specify zero (0).

4. Specify the class loader order for the application.

   The application class loader order specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The default is to search in the parent class loader before searching in the application class loader to load a class.

   Select either of the following values for **Class loader order**:

Option	Description
Classes loaded with parent class loader first	Causes the class loader to search in the parent class loader first to load a class. This value is the standard for Development Kit class loaders and WebSphere Application Server class loaders.
Classes loaded with application class loader first	Causes the class loader to search in the application class loader first to load a class. By specifying `Classes loaded with application class loader first`, your application can override classes contained in the parent class loader. **Attention:** Specifying the `Classes loaded with application class loader first` value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes.

5. Specify whether to use a single or multiple class loaders to load Web application archives (WAR files) of your application.

   By default, Web modules have their own WAR class loader to load the contents of the `WEB-INF/classes` and `WEB-INF/lib` directories. The default WAR class loader value is `Class loader for each WAR file in application`, which uses a separate class loader to load each WAR file. Setting the value to `Single class loader for application` causes the application class loader to load the Web module contents as well as the EJB modules, shared libraries, RAR files, and dependency JAR files associated to the application. The application class loader is the parent of the WAR class loader.

   Select either of the following values for **WAR class loader policy**:

Option	Description
Class loader for each WAR file in application	Uses a different class loader for each WAR file.
Single class loader for application	Uses a single class loader to load all of the WAR files in your application.

6. Click **OK**.

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Save changes to your administrative configuration.

In the Network Deployment product, the application binaries are transferred to nodes when the configuration changes on the deployment manager synchronize with configurations for individual nodes on which the application will run.

Class loading and update detection settings

Use this page to configure use of class loaders by an application.

To view this administrative console page, click **Applications > Enterprise Applications >** *application_name* **> Class loading and update detection**.

Reload classes when application files are updated:

Specifies whether to enable class reloading when application files are updated.

Select **Reload classes when application files are updated** to set reloadEnabled to `true` in the `deployment.xml` file for the application. If an application's class definition changes, the application server run time stops and starts the application to reload application classes.

For JavaServer Pages (JSP) files in a Web module, a Web container reloads JSP files only when the IBM extension jspReloadingEnabled in the jspAttributes of the `ibm-web-ext.xmi` file is set to `true`. You can enable JSP reloading during deployment on the JSP Reload Options panel.

Data type	Boolean
Default	false

Polling interval for updated files:

Specifies the number of seconds to scan the application's file system for updated files. The default is the value of the reloading interval attribute in the IBM extension (`META-INF/ibm-application-ext.xmi`) file of the EAR file.

This **Polling interval for updated files** setting is the same as the **Reload interval in seconds** field on the application installation and update wizards.

To enable reloading, specify a value greater than zero (for example, 1 to 2147483647). To disable reloading, specify zero (0). The range is from 0 to 2147483647.

The reloading interval attribute takes effect only if class reloading is enabled.

Data type	Integer
Units	Seconds
Default	3

Class loader order:

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for development kit class loaders and WebSphere Application Server class loaders is `Classes loaded with parent class loader first`. By specifying `Classes loaded with application class loader first`, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

The options are `Classes loaded with parent class loader first` and `Classes loaded with application class loader first`. The default is to search in the parent class loader before searching in the application class loader to load a class.

For your application to use the default configuration of Jakarta Commons Logging in WebSphere Application Server, set this application class loader mode to `Classes loaded with parent class loader first`. For your application to override the default configuration of Jakarta Commons Logging in WebSphere Application Server, your application must provide the configuration in a form supported by Jakarta Commons Logging and this class loader mode must be set to `Classes loaded with application class loader first`. Also, to override the default configuration, set the class loader mode for each Web module in your application so that the correct logger factory loads.

Data type	String
Default	Classes loaded with parent class loader first

WAR class loader policy:

Specifies whether to use a single class loader to load all WAR files of the application or to use a different class loader for each WAR file.

The options are `Class loader for each WAR file in application` and `Single class loader for application`. The default is to use a separate class loader to load each WAR file.

Data type	String
Default	Class loader for each WAR file in application

Manage modules settings

Use this panel to specify deployment targets where you want to install the modules contained in your application. Modules can be installed on the same deployment target or dispersed among several deployment targets. A deployment target can be an application server, cluster of application servers or Web server.

To view this administrative console panel, click **Applications > Enterprise Applications > *application_name* > Manage modules**. This panel is the similar to the **Map modules to servers** panel on the application installation and update wizards.

On this panel, each **Module** must map to one or more desired targets, identified under **Server**. To change a mapping:

1. In the list of mappings, select the **Select** check box beside each module that you want mapped to the same target(s).
2. From the **Clusters and Servers** drop-down list, select one or more targets. Select only appropriate deployment targets for a module. Modules that use WebSphere Application Server Version 6.x features cannot be installed onto a Version 5.x target server.

   Use the Ctrl key to select multiple targets. For example, to have a Web server serve your application, press the Ctrl key and then select an application server or cluster and the Web server together. The plug-in configuration file `plugin-cfg.xml` for that Web server will be generated based on the applications which are routed through it.
3. Click **Apply**.

If this Manage modules panel was accessed from a console enterprise application page for an already installed application, you can also use this panel to view and manage modules in your application.

To view the values specified for a module configuration, click the module name in the list. The displayed module settings page shows the values specified. On the settings page, you can change existing configuration values and link to additional console pages that assist you in configuring the module.

To manage a module, enable the **Select** check box beside the module name in the list and click a button:

Button	Resulting action
Remove	Removes the selected module from the deployed application. The module is deleted from the application in the configuration repository and also from all of the nodes where the application is installed and running (or expected to run). If the application is running on a node when the module file is deleted from the node as a result of configuration synchronization then the application is stopped, the module file is deleted from the node's file system, and the application is restarted.
Update	Opens a wizard that helps you update module in an application. If a module has the same URI as a module already existing in the application, the new module replaces the existing module. If the new module does not exist in the application, it is added to the deployed application. If the application is running on a node when the module file is updated on the node as a result of configuration synchronization then the application is stopped, the module file is updated on the node's file system, and the application is restarted. If the application is running on a node when the module file is added as a result of configuration synchronization then the newly added module is started without stopping and restarting the running application.
Remove File	Deletes a file from a module of a deployed application. The file is also deleted from all the nodes where the module is installed after configuration is synchronized with nodes. If the application is running on a node when the module file is updated on the node as a result of configuration synchronization then the application is stopped, the module file is updated on the node's file system, and the application is restarted.

Clusters and Servers

Lists the names of available target servers and clusters. This list is the same for every application that is installed in the cell.

From this list, select only appropriate deployment targets for a module. You can install an application, enterprise bean (EJB) module or Web module developed for a Version 5.x product on a 5.x or 6.x deployment target, provided the module--
* Does not support Java 2 Platform, Enterprise Edition (J2EE) 1.4;
* Does not call any 6.x runtime application programming interfaces (APIs); and
* Does not use any 6.x product features.

If the module supports J2EE 1.4, then you must install the module on a 6.x deployment target. If the module calls a 6.1.x API or uses a 6.1.x feature, then you must install the module on a 6.1.x deployment target. Modules that call a 6.0.x API or use a 6.0.x feature can be installed on a 6.0.x or 6.1.x deployment target.

Module

Specifies the name of a module in the installed (or deployed) application.

URI

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

Module type

Specifies the type of module, for example a Web module or EJB module.

This setting is shown on the Manage modules panel accessed from a console enterprise application page.

Server

Specifies the name of each server or cluster to which the module currently is mapped--that is, the deployment targets.

To change the deployment targets for a module, select one or more targets from the **Clusters and Servers** drop-down list and click **Apply**. The new mapping replaces the previous mapping.

Mapping modules to servers

Each module of a deployed application must be mapped to one or more target servers. The target server can be an application server, cluster of application servers or Web server.

You can map modules of an application or standalone Web module to one or more target servers during or after application installation using the console. This topic assumes that the module is already installed on a server and that you want to change the mappings.

Before you change a mapping, check the deployment targets. You must specify an appropriate deployment target for a module. Modules that use Version 6.x features cannot be installed onto a Version 5.x target server.

During application installation, different deployment targets might have been specified.

You use the Manage modules panel of the administrative console to view and change mappings. This panel is displayed during application installation using the console and, after the application is installed, can be accessed from the settings page for an enterprise application.

On the Manage modules panel, specify target servers where you want to install the modules contained in your application. Modules can be installed on the same application server or dispersed among several application servers. Also, specify the Web servers as targets that will serve as routers for requests to your application. The plug-in configuration file `plugin-cfg.xml` for each Web server is generated based on the applications which are routed through it.

1. Click **Applications > Enterprise Applications >** *application_name* **> Manage modules** in the console navigation tree. The Manage modules panel is displayed.
2. Examine the list of mappings. Ensure that each **Module** entry is mapped to the desired target(s), identified under **Server**.
3. Change a mapping as needed.
   a. Select each module that you want mapped to the same target(s). In the list of mappings, place a check mark in the **Select** check boxes beside the modules.
   b. From the **Clusters and Servers** drop-down list, select one or more targets. Use the **Ctrl** key to select multiple targets. For example, to have a Web server serve your application, use the **Ctrl** key to select an application server or cluster and the Web server together in order to have the plug-in configuration file `plugin-cfg.xml` for that Web server generated based on the applications which are routed through it.
   c. Click **Apply**.
4. Repeat steps 2 and 3 until each module maps to the desired target(s).
5. Click **OK**.

The application or module configurations are changed. The application or standalone Web module is restarted so the changes take effect.

If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Save changes to your administrative configuration.

In the Network Deployment product, the application binaries are transferred to nodes when the configuration changes on the deployment manager synchronize with configurations for individual nodes on which the application will run.

Tip: If, at a future time, you intend to delete the server or cluster to which the modules are mapped, add a new server or cluster and then remap the modules to it before deleting the old server or cluster. If you do not remap the modules to a new server or cluster before deleting the old one, you must uninstall each application or module and then reinstall them on an existing server or cluster. After a server or cluster to which modules are mapped is deleted, the modules cannot be remapped to another existing server or cluster.

Mapping virtual hosts for Web modules

A virtual host must be mapped to each Web module of a deployed application. Web modules can be installed on the same virtual host or dispersed among several virtual hosts.

You can map a virtual host to a Web module during or after application installation using the console. This article assumes that the Web module is already installed on a server and that you want to change the mappings.

Before you change a mapping, check the virtual hosts definitions. You can install a Web module on any defined virtual host. To view information on previously defined virtual hosts, click **Environment > Virtual Hosts** in the administrative console. Virtual hosts enable you to associate a unique port with a module or application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host alias is used in the URL that is used to access artifacts such as servlets and JavaServer Pages (JSP) files in a Web module. For example, the alias `myhost:8080` is the `host_name:port_number` portion of the URL `http://myhost:8080/servlet/snoop`.

During application installation, a virtual host other than the one you want mapped to your Web module might have been specified.

The default virtual host setting usually is `default_host`, which provides several port numbers through its aliases:

80   An internal, insecure port used when no port number is specified
9080 An internal port
9443 An external, secure port

Unless you want to isolate your Web module from other modules or resources on the same node (physical machine), `default_host` is a suitable virtual host for your Web module.

In addition to `default_host`, WebSphere Application Server provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the Web module relates to system administration.

Use the Virtual hosts page of the administrative console to view and change mappings. This page is displayed during application installation using the console and, after the application is installed, can be accessed from the settings page for an enterprise application.

On the Virtual hosts page, specify a virtual host for each Web module. Web modules of an application can be installed on the same virtual host or on different virtual hosts.

1. Click **Applications > Enterprise Applications >***application_name* **> Virtual hosts** in the console navigation tree. The Virtual hosts page is displayed.

2. Examine the list of mappings. Ensure that each **Web module** entry has the desired virtual host mapped to it, identified under **Virtual host**.

3. Change the mappings as needed.

a. Select each Web module that you want mapped to a particular virtual host. In the list of mappings, place a check mark in the **Select** check boxes beside the Web modules.

b. From the **Virtual host** drop-down list, select the desired virtual host. If you selected more than one virtual host in step 1:

   1) Expand **Apply Multiple Mappings**.
   2) Select the desired virtual host from the **Virtual host** drop-down list.
   3) Click **Apply**.

4. Repeat steps 2 and 3 until a desired virtual host is mapped to each Web module.

5. Click **OK**.

The application or Web module configurations are changed. The application or standalone Web module is restarted so the changes take effect.

After mapping virtual hosts, do the following:

1. If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

2. Regenerate the plug-in configuration file.
   a. Click **Servers > Web servers**.
   b. Select the Web server for which you want to generate a plug-in.
   c. Click **Generate Plug-in**.

3. Save changes to your administrative configuration.

   In the Network Deployment product, the application binaries are transferred to nodes when the configuration changes on the deployment manager synchronize with configurations for individual nodes on which the application will run.

Virtual hosts settings

Use this panel to specify virtual hosts for Web modules contained in your application. Web modules can be installed on the same virtual host or dispersed among several virtual hosts.

To view this administrative console panel, click **Applications > Enterprise Applications > **application_name** > Virtual hosts**. This panel is the same as the **Map virtual hosts for Web modules** panel on the application installation and update wizards.

On this panel, each Web module must map to a previously defined virtual host, identified under **Virtual host**. You can see information on previously defined virtual hosts by clicking **Environment > Virtual Hosts** in the administrative console. Virtual hosts enable you to associate a unique port with a module or application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host alias is used in the URL that is used to access artifacts such as servlets and JavaServer Pages (JSP) files in a Web module. For example, the alias `myhost:8080` is the `host_name:port_number` portion of the URL `http://myhost:8080/servlet/snoop`.

The default virtual host setting usually is `default_host`, which provides several port numbers through its aliases:

80   An internal, insecure port used when no port number is specified
9080   An internal port
9443   An external, secure port

Unless you want to isolate your Web module from other modules or resources on the same node (physical machine), `default_host` is a suitable virtual host for your Web module.

In addition to `default_host`, the product provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the Web module relates to system administration.

To change a mapping:

1. In the list of mappings, select the **Select** check box beside each Web module that you want mapped to a particular virtual host.
2. From the **Virtual host** drop-down list, select the desired virtual host. If you selected more than one virtual host in step 1:
   a. Expand **Apply Multiple Mappings**.
   b. Select the desired virtual host from the **Virtual Host** drop-down list.
   c. Click **Apply**.
3. Click **OK**.

Web module:

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

Virtual host:

Specifies the name of the virtual host to which the Web module is currently mapped.

Expanding the drop-down list displays a list of previously defined virtual hosts. To change a mapping, select a different virtual host from the list.

Do not specify the same virtual host for different Web modules that have the same context root and are deployed on targets belonging to the same node even if the Web modules are contained in different applications. Specifying the same virtual host causes a validation error.

Mapping properties for a custom login configuration

Use this page to view and manage the mapping properties for a custom login configuration.

To access the administrative console panel, complete the following steps:

1. Click **Applications > Enterprise applications >** *application_name*.
2. Under Enterprise Java Bean Properties, click **2.x entity bean data sources**.
3. For Container authorization, modify the authorization type by selecting you rEJB module and selecting **Container** from the Resource authorization menu.
4. Click **Apply**.
5. Under Specify authentication method, select **Use custom login configuration** and the name of the application login configuration.
6. Select the name of your EJB module.
7. Click **Apply**.
8. Click **Mapping properties** in the Resource authorization column. This property is not available until after you click apply in the previous step.

Name
Specifies the name for the mapping property.

Do not use the `MAPPING_ALIAS` property name because the name is reserved by the product.

Value
Specifies the value paired with the specified name.

Description

Provides additional information about the name and value pair.

Viewing deployment descriptors

A deployment descriptor is an extensible markup language (XML) file that specifies configuration and container options for an application or module.

This topic assumes that you have installed an application or module on a server and that you want to view its deployment descriptor.

When you create an application or module in an assembly tool such as the Application Server Toolkit (AST) or Rational Application Developer, the assembly tool creates deployment descriptor files for the application or module.

You can edit a deployment descriptor file manually. However, it is preferable to edit a deployment descriptor using an assembly tool deployment descriptor editor to ensure that the deployment descriptor has valid properties and that its references contain appropriate values.

After an application or module is installed on a server, you can view its deployment descriptor in the administrative console.

1. Access a deployment descriptor view.

   Click the navigational option stated in **Accessing a console view** to view the deployment descriptor for a given module:

Module	Deployment descriptor file	Accessing a console view
Enterprise application	`application.xml`	**Applications > Enterprise Applications >** *application_name* > **View deployment descriptor**
Web application	`WEB-INF/web.xml`	**Applications > Enterprise Applications >** *application_name* > **Manage Modules >** *module_name* > • **View deployment descriptor**
	`WEB-INF/portlet.xml`	**Applications > Enterprise Applications >** *application_name* > **Manage Modules >** *module_name* > • **View portlet deployment descriptor**
Enterprise bean	`ejb-jar.xml`	**Applications > Enterprise Applications >** *application_name* > **Manage Modules >** *module_name* > **View deployment descriptor**
Application client	`application-client.xml`	No console view
Web service	`webservices.xml`	**Applications > Enterprise Applications >** *application_name* > **Manage Modules >** *module_name* > • **View Web services client deployment descriptor extension** • **View Web services server deployment descriptor** • **View Web services server deployment descriptor extension** "Viewing Web services deployment descriptors in the administrative console" on page 450 describes the views.
Resource adapter	`ra.xml`	**Resource Adapters > Resource Adapters >** *module_name* > **View deployment descriptor**

2. Click **Expand All** to view the deployment descriptor contents.

A deployment descriptor such as the following for the product DefaultApplication is displayed:

```
<application id="Application_ID" >
  <display-name> DefaultApplication.ear</display-name>
  <description> This is the IBM WebSphere Application Server Default Application.</<description>
  <module id="WebModule_1" >
    <web>
      <web-uri> DefaultWebApplication.war</web-uri>
      <context-root> /</context-root>
    </web>
  </module>
  <module id="EjbModule_1" >
    <ejb> Increment.jar</ejb>
  </module>
  <security-role id="SecurityRole_1130344639273" >
    <description> All Authenticated users role.</description>
    <role-name> All Role</role-name>
  </security-role>
</application>
```

Verify the deployment descriptor contents, including any configurations that it has for bindings, security roles, references to other resources, or Java Naming and Directory Interface (JNDI) names.

Change a deployment descriptor as needed in an assembly tool or using the console.

Starting or stopping applications

You can start an application that is not running (has a status of *Stopped*) or stop an application that is running (has a status of *Started*).

This topic assumes that the application is installed on a server. By default, the application starts automatically when the server starts.

You can start and stop applications manually using the following:
* Administrative console
* wsadmin startApplication and stopApplication commands
* Java programs that use ApplicationManager or AppManagement MBeans

This topic describes how to use the administrative console to start or stop an application.
1. Go to the Enterprise Applications page. Click **Applications > Enterprise Applications** in the console navigation tree.
2. Select the check box for the application you want started or stopped.
3. Click a button:

Option	Description
Start	Runs the application and changes the state of the application to *Started*. The status is changed to *partially started* if not all servers on which the application is deployed are running.
Stop	Stops the processing of the application and changes the state of the application to *Stopped*.

To restart a running application, select the application you want to restart, click **Stop** and then click **Start**.

The status of the application changes and a message stating that the application started or stopped displays at the top the page.

You can configure an application so it does not start automatically when the server on which it resides starts. You then start the application manually using options described in this article.

If you want your application to start automatically when its server starts, you can adjust values that control how quickly the application or its server starts:

1. Go the settings page for your enterprise application. Click **Applications > Enterprise Applications >** *application_name* **> Startup behavior**.

2. Specify a different value for **Startup order**.

   This setting specifies the order in which applications are started when the server starts. The default value is 1 in a range from 0 to 2147483647. The application with the lowest starting weight is started first.

3. Specify a different value for **Launch application before server completes startup**.

   This setting specifies whether the application must initialize fully before its server starts. The default value of `false` prevents the server from starting completely until the application starts. To reduce the amount of time it takes to start the server, you can set the value to `true` and have the application start on a background thread, thus allowing server startup to continue without waiting for the application

4. If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

5. Save the changes to the application configuration.

Disabling automatic starting of applications

You can enable and disable the automatic starting of an application. By default, an installed application starts automatically when the server on which the application resides starts.

This topic assumes that the application is installed on an application server and that the application starts automatically when the server starts.

This topic also assumes that you mapped the installed application to a server or cluster.

You might want an application to run only after you start it manually and not to run every time after the server starts. The target mapping for an application controls whether an application starts automatically when the server starts or requires you to start the application manually.

1. Go to the Target specific application status page for your application.

   Click **Applications > Enterprise Applications >** *application_name* **> Target specific application status**.

2. Select the target server or cluster on which the application resides.

3. Click **Disable Auto Start**.

4. Save changes to the administrative configuration.

The application does not start when its server starts. You must start the application manually.

To enable automatic starting of the application, do the following:
1. On the Target specific application status page for the application, select the target on which the application resides.
2. Click **Enable Auto Start**.
3. Save changes to the configuration.

Target specific application status

Use this page to view mappings of deployed applications or modules to servers or clusters.

Also use this page to enable or disable the automatic starting of an application when the server on which the application resides starts.

To view this administrative console page, click **Applications > Enterprise Applications >** *application_name* **> Target specific application status**.

Target

States the name of the target server or cluster to which the application or module maps. You specify the target on the Manage modules page accessed from the settings for an application.

Node

Specifies the node name if the target is a server.

Version

Specifies the version level of the target. The target can be a 5.x deployment target or a 6.x deployment target.

A *5.x deployment target* is a server or a cluster with at least one member on a WebSphere Application Server Version 5 product.

A *6.x deployment target* is a server or cluster with all members on a WebSphere Application Server Version 6 product.

An application, enterprise bean (EJB) module Session Initiation Protocol (SIP) module (SAR), or Web module developed for a Version 5.x product can reside on a 5.x or 6.x deployment target, provided the module--
- Does not support Java 2 Platform, Enterprise Edition (J2EE) 1.4;
- Does not call any 6.x runtime application programming interfaces (APIs); and
- Does not use any 6.x product features.

Similarly, a resource adapter (connector) module, or RAR file, developed for a Version 5.x product can reside on a 5.x or 6.x node, provided the module does not support Java Cryptography Architecture (JCA) 1.5 and does not call any 6.x runtime application programming interfaces (APIs). If the module supports JCA 1.5 or calls a 6.x API, then the module must reside on a 6.x node.

If JavaServer Pages (JSP) precompilation, EJB deployment (`ejbdeploy`), or Web Services deployment (`wsdeploy`) are enabled, then you can deploy applications to only those targets that have same product version as the deployment manager. If applications are targeted to servers that have an earlier version than the deployment manager, then you cannot deploy to those targets. Thus, if JSP precompilation, `ejbdeploy`, or `wsdeploy` are enabled, then you can deploy applications to only a 6.1 target.

Auto Start

Specifies whether the application modules installed on the target server are started (or enabled) when the server starts. This setting specifies the initial state of application modules. A `Yes` value indicates that the corresponding modules are enabled and thus are accessible when the server starts. A `No` value indicates that the corresponding modules are not enabled and thus are not accessible when the server starts.

By default, Auto Start is enabled. Thus, by default an installed application starts automatically when the server on which the application resides starts.

You can enable and disable the automatic starting of the application. To disable the automatic starting of the application, enable the **Select** check box beside the target server or cluster and click **Disable Auto Start**. When automatic starting is disabled, the application does not start when its server starts. To enable the automatic starting of the application, select the target and click **Enable Auto Start**.

Application Status

Indicates whether the application deployed on the application server is started, stopped, or unavailable.

	Started	Application is running.

	Partial Start	Application is in the process of changing from a *Stopped* state to a *Started* state. Application is starting to run but is not fully running yet. The application might be in the Partial Start state because one of its application servers is not started.
	Stopped	Application is not running.
	Partial Stop	Application is in the process of changing from a *Started* state to a *Stopped* state. Application has not stopped running yet.
	Unavailable	Status cannot be determined.
	Not applicable	Application does not provide information as to whether it is running.

Exporting applications

You can export an enterprise application to a location of your choice.

Exporting applications enables you to back up your applications and preserve binding information for the applications. You might export your applications before updating installed applications or migrating to a later version of the WebSphere Application Server product.

1. Click **Applications > Enterprise Applications** in the console navigation tree to access the Enterprise Applications page.
2. Select the check box beside the application and click **Export**.
3. On the Export Application EAR Files page, click on the link to download the exported EAR file.
4. Use the browser dialogue to specify a location at which to save the exported EAR file.
5. Click **Back** to return to the Enterprise Applications page.

The file containing binding information is exported to the specified node and directory, and has the name *enterprise_application_name*.ear.

Exporting DDL files

You can export data definition language (DDL) files in the enterprise bean (EJB) modules of an application.

Exporting DDL (`Table.ddl`) files in the EJB modules of an application downloads the DDL files to a location of your choice.

1. Click **Applications > Enterprise Applications** in the administrative console navigation tree to access the Enterprise Applications page.
2. Place a check mark in the check box beside the application and click **Export DDL**. If the application has no DDL files in any of its EJB modules, then the message *No DDL files were found* is displayed at the top of the page. If the application has DDL files in its EJB modules, then a page listing DDL files in the format *application_name.ear/_module*.jar_Table.ddl is displayed.
3. Click on a file in the list and specify the location to which to download the file.

   Tip: Mozilla browsers might display the contents of the `Table.ddl` file instead of saving the file to disk. To save the file, edit the **Helper Application** preference settings of the Mozilla browser by adding a new type for DDL and specifying that you want to save DDL files to disk. That is, set `MIME type` = `ddl` and `Extension` = `ddl`.

The DDL file is downloaded to the specified location.

Updating applications

You can update application files deployed on a server.

Update your application or modules and reassemble them using an assembly tool. Typical tasks include adding or editing assembly properties, adding or importing modules into an application, and adding enterprise beans, Web components, and files.

Also, determine whether the updated files can be installed to your deployment targets. WebSphere Application Server Version 6.x supports Java 2 Platform, Enterprise Edition (J2EE) 1.4 enterprise applications and modules. If you are deploying J2EE 1.4 modules, ensure that the target server and its node support Version 6.x. The administrative console Server collection pages show the versions for servers and cluster members. You can deploy J2EE 1.4 modules to Version 6.x servers or to clusters that contain Version 6.x cluster members only. You cannot deploy J2EE 1.4 modules to servers on Version 5.x nodes or to clusters that contain Version 5.x cluster members. See "Installable module versions" on page 1365 for details.

Updating consists of adding a new file or module to an installed application, or replacing or removing an installed application, file or module. After replacement of a full application, the old application is uninstalled. After replacement of a module, file or partial application, the old installed module, file or partial application is removed from the installed application.

1. Determine which method to use to update your application files. WebSphere Application Server provides several ways to update modules.
2. Update the application files using
   - Administrative console
   - wsadmin scripts
   - Java application programming interfaces
   - WebSphere rapid deployment of J2EE applications

   In some situations, you can update applications or modules without restarting the application server using hot deployment. Do not use hot deployment unless you are an experienced user and are updating applications in a development or test environment.
3. Start the deployed application files using
   - Administrative console
   - wsadmin startApplication
   - Java programs that use ApplicationManager or AppManagement MBeans

Save the changes to your administrative configuration.

When saving the configuration, synchronize the configuration with the nodes where the application is expected to run.

Next, test the application. For example, point a Web browser at the URL for a deployed application (typically http://*hostname*:9060/*Web_module_name*, where *hostname* is your valid Web server and 9060 is the default port number) and examine the performance of the application. If the application does not perform as desired, edit the application configuration, then save and test it again.

Ways to update application files

You can update application files deployed on a server or cluster in several ways.

Table 51. Ways to update application files

Option	Method	Comments	Starting after update
Administrative console update wizard See "Updating applications with the console" on page 1422.	Briefly, do the following: 1. Go to the Enterprise Applications page. Click **Applications > Enterprise Applications** in the console navigation tree. 2. Select the application to update and click **Update**. 3. On the Preparing for application update page, identify the application, module or files to update and click **Next**. 4. Complete steps in the update wizard and click **Finish**.	On the Preparing for application update page: • Use **Full application** to update an `.ear` file. • Use **Single module** to update a `.war`, `.sar`, enterprise bean `.jar`, or connector `.rar` file. • Use **Single file** to update a file other than an `.ear`, `.war`, `.sar`, EJB `.jar`, or `.rar` file. • Use **Partial application** to update or remove multiple files.	On the Enterprise Applications page, select the updated application and click **Start**.
wsadmin scripts	Use the **update** command or the **updateInteractive** command in a script or at a command prompt. For more information on the **update** and **updateInteractive** commands, see the Commands for the AdminApp object topic.	Getting started with scripting provides an overview of wsadmin.	Start the application using the **invoke** command and the startApplication attribute. For more information about the **invoke** command, see the Commands for the AdminControl object topic.
Java application programming interfaces See Using administrative programs (JMX).	Update deployed applications by completing the steps in Managing applications through programming.	Update an application in the following ways: • Update the entire application • Add to, replace or delete multiple files in an application • Add a module to an application • Update a module in an application • Delete a module in an application • Add a file to an application • Update a file in an application • Delete a file in an application	• Invoke the AdminApp *startApplication* command. • Invoke the *startApplication* method on an ApplicationManager MBean using AdminControl.

Table 51. Ways to update application files (continued)

Option	Method	Comments	Starting after update
WebSphere rapid deployment See topics under Rapid deployment of J2EE applications in this information center.	Briefly, do the following: 1. Update your J2EE application files. 2. Set up the rapid deployment environment. 3. Create a free-form project. 4. Launch a rapid deployment session. 5. Drop your updated application files into the free-form project.	WebSphere rapid deployment offers the following advantages: • You do not need to assemble your J2EE application files prior to deployment. • You do not need to use other installation tools mentioned in this table to deploy the files.	Use any of the above options to start the application. Clicking **Start** on the Enterprise Applications page is the easiest option.
Hot deployment and dynamic reloading	Briefly, do the following: 1. Update your application (.ear), Web module (.war), enterprise bean .jar or HTTP plug-in configuration file. 2. Follow instructions in Hot deployment and dynamic reloading to update your file.	If you are new to WebSphere Application Server, use the administrative console to update applications. That option is easier. Hot deployment and dynamic reloading is more difficult to complete. You must directly manipulate the application or module file on the server where the application is deployed.	Use any of the above options to start the application. Clicking **Start** on the Enterprise Applications page is the easiest option.

You can update .ear, enterprise bean .jar, Web module .war, Session Initiation Protocol (SIP) module (.sar), connector .rar, application client .jar, and any other files used by an installed application.

If the application is updated while it is running, WebSphere Application Server automatically stops the application, updates the application logic and restarts the application. If the application does not start automatically, start it manually using one of the **Starting** options. For more information on the restarting of updated applications, refer to Fine-grained recycle behavior in *IBM WebSphere Developer Technical Journal: System management for WebSphere Application Server V6 -- Part 5 Flexible options for updating deployed applications*.

Updating applications with the console

Updating applications consists of adding a new file or module to an installed application, or replacing or removing an installed application, file or module.

Before you update the application files on a server, ensure that the files are assembled in deployable modules.

Next, refer to "Ways to update application files" on page 1420 and decide how to update your application files. You can update enterprise applications or modules using the administrative console, the wsadmin tool, or Java MBean programming. These ways provide similar updating capabilities.

Further, ensure that the updated files can be installed to your deployment targets.

This topic describes how to update deployed applications or modules using the administrative console.

1. Back up the installed application.

   a. Go to the Enterprise Applications page of the administrative console. Click **Applications > Enterprise Applications** in the console navigation tree.

b. Export the application to an EAR file. Select the application you want uninstalled and click **Export**. Exporting the application preserves the binding information.

2. With the application selected on the Enterprise Applications page, click **Update**. The Preparing for application update page is displayed.

3. Under **Specify the EAR, WAR, SAR or JAR module to upload and install**:

   a. Ensure that **Application to be updated** refers to the application to be updated.

   b. Under **Application update options**, select the installed application, module, or file that you want to update.

   The online help Preparing for application update settings provides detailed information on the options.

4. If you selected the **Replace the entire application** or **Replace or add a single module** option:

   a. Click **Next** to display a wizard for updating application files.

   b. Complete the steps in the update wizard.

   This update wizard, which is similar to the installation wizard, provides fields for specifying or editing application binding information. Refer to information on installing applications and on the settings page for application installation for guidance.

   Note that the installation steps have the merged binding information from the new version and the old version. If the new version has bindings for application artifacts such as EJB JNDI names, EJB references or resource references, then those bindings will be part of the merged binding information. If new bindings are not present, then bindings are taken from the installed (old) version. If bindings are not present in the old version and if the default binding generation option is enabled, then the default bindings will be part of the merged binding information.

   You can select whether to ignore bindings in the old version or ones in the new version.

5. Click **Finish**.

6. If you did not use the Manage modules page of the update wizard, after updating the application, map the installed application or module to servers or clusters.

   Use the Manage modules page accessed from the Enterprise Applications page.

   a. Go to the Manage modules page. Click **Applications > Enterprise Applications >** *application_name* **> Manage modules**.

   b. Specify the application server where you want to install modules contained in your application and click **OK**.

   You can deploy J2EE 1.4 modules to servers on Version 6.x nodes or to clusters that contain cluster members on Version 6.x nodes only.

After replacement of a full application, the old application is uninstalled. After replacement of a module, file or partial application, the old installed module, file or partial application is removed from the installed application.

After the application file or module installs successfully, do the following:

1. If a changed application or module is deployed on a cluster, roll out the changes to all cluster members of the cluster on which the application or module is deployed. Click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

   Tip: At the end of the *Installing* messages displayed by the console during application or module installation, click **Manage applications** to go to the Enterprise Applications page. Do not save changes to your configuration until after you roll out the changes.

2. Save the changes to your configuration.

In the Network Deployment product, after you click **Save** the old application files are deleted and new files are copied when the configuration on the deployment manager synchronizes with the configuration on the node where the application is installed.

If the application is running when you update it, the application stops running before its files are copied to the destination directory of the node and restarts after the copy operation completes. Thus, the application is unavailable on the node during the time the node is synchronizing its configuration with the deployment manager.

3. If needed, restart the application manually so the changes take effect.

If the application is updated while it is running, WebSphere Application Server automatically stops the application or only its changed components, updates the application logic, and restarts the stopped application or its components. For more information on the restarting of updated applications, refer to Fine-grained recycle behavior in *IBM WebSphere Developer Technical Journal: System management for WebSphere Application Server V6 -- Part 5 Flexible options for updating deployed applications*.

4. If the application you are updating is deployed on a server that has its application class loader policy set to `Single`, restart the server.

Preparing for application update settings

Use this page to update enterprise applications, modules or files already installed on a server.

To view this administrative console page, do the following:
1. Click **Applications > Enterprise Applications**.
2. Select the installed application or module that you want to update.
3. Click **Update**.

Clicking **Update** displays a page that helps you update application files deployed in the cell. You can update the full application, a single module, a single file, or part of the application. If a new file or module has the same relative path as a file or module already existing on the server, the new file or module replaces the existing file or module. If the new file or module does not exist on the server, it is added to the deployed application.

Application to be updated

Specifies the name of the installed (or deployed) application that you selected on the Enterprise Applications page.

Replace the entire application

Under **Application update options**, specifies to replace the application already installed on the server with a new (updated) enterprise application `.ear` file.

After selecting this option, do the following:

1. Specify whether the `.ear` file is on a local or remote file system and the full path name of the application. The path provides the location of the updated `.ear` file before installation.

   Use **Local file system** if the browser and the updated files or modules are on the same machine, whether or not the server is on that machine too. **Local file system** is available for all update options.

   Use **Remote file system** if the application file resides on any node in the current cell context. You can browse the entire file system of a node if the node agent or deployment manager is running on that selected node. Only `.ear`, `.jar`, `.sar`, or `.war` files are shown during the browsing.

   Also use the **Remote file system** option to specify an application file already residing on the machine running the application server. For example, the field value might be `app_server_install_root/installableApps/test.ear`. If you are installing a standalone WAR module, then specify the context root as well.

   Tip: During application installation, application files typically are uploaded from a client machine running the browser to the server machine running the administrative console, where they are deployed. In such cases, use the Web browser running the administrative console to select `.ear`,

.war, .sar, or .jar modules to upload to the server machine. In some cases, however, the application files reside on the file system of any of the nodes in a cell. To have the application server install these files, use the **Remote file system** option.

2. If you are installing a standalone Web application (WAR) or a Session Initiation Protocol (SIP) module (SAR), specify the context root of the WAR or SAR file.

   The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is http://host:port/gettingstarted/MySession.

3. Specify whether to show only installation options that require you to supply information or to show all installation options.

   The **Prompt me only when additional information is required** option enables you to install your application more easily because you do not need to examine all available installation options.

   However, to use the **Generate default bindings** option, which might be the quickest and easiest option for installing your application, you must select the **Show me all installation options and parameters** and then select **Generate default bindings** on the next panel.

4. Click **Next** to display a wizard for updating application files. The update wizard, which is similar to the installation wizard, provides fields for specifying or editing application binding information. Complete the steps in the update wizard as needed.

When the full application is updated, the old application is uninstalled and the new application is installed. When the configuration changes are saved and subsequently synchronized, the application files are expanded on the node where application will run. If the application is running on the node while it is updated, then the application is stopped, application files are updated, and application is started.

Replace or add a single module

Under **Application update options**, specifies to replace a module in or add a module to an installed application. The module can be a Web module (.war file), enterprise bean module (EJB .jar file), SIP module (.sar file), or resource adapter module (connector .rar file).

After selecting this option, specify whether the module is on a local or remote file system and the full path name of the module. The path provides the location of the updated module before installation. For information on **Local file system** and **Remote file system**, refer to the description of **Replace the entire application** above.

To replace a module, the specified relative path (module URI) must match the path of the module to be updated in the installed application.

To add a new module to the installed application, the specified relative path must *not* match the path of a module in the installed application. The value specifies the desired path for the new module.

If you are installing a standalone Web or SIP module, specify a value for **Context root**. The context root is combined with the defined servlet mapping (from the .war file) to compose the full URL that users type to access the servlet. For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is http://host:port/gettingstarted/MySession.

Next, specify whether to show only installation options that require you to supply information or to show all installation options.

After specifying the required information on the module, click **Next** to display a wizard for updating application files. The update wizard, which is similar to the installation wizard, provides fields for specifying or editing module binding information. Complete the steps in the update wizard as needed.

After a single module is added or updated, when configuration changes are saved, the new or updated module is stored in the deployed application in the WebSphere Application Server configuration repository.

When these changes are synchronized with the node, the module is added or updated to the node's file system. If the application is running on the node when the module is added or updated, then one of the following occurs:

- For updates to a Web module, the running Web module is stopped, Web module files are updated, and then the Web module is started.
- For module additions, the added module is started on the application servers where the application is running after it is expanded on the node. An application restart is not necessary.
- If the class loader policy for the application is set to `Single` so that all modules share a class loader, then the entire application is stopped and restarted for module level changes.
- If the security provider configured with WebSphere Application Server does not support dynamic updates, then the entire application is stopped and restarted for module level changes.
- For all other updates to a module, the entire application is stopped, the module files are updated, then the entire application is started.

Replace or add a single file

Under **Application update options**, specifies to replace a file in or add a file to an installed application.

Use this option to update a file used by the application that is not an `.ear`, `.war`, `.sar`, `.rar` or, in some instances, a `.jar` file. You can use this option to add or update `.jar` files that are not defined as modules in the application. To update an `.ear`, file use the **Replace the entire application** option. To update a `.war` file, `.sar` file, `.rar` file, or `.jar` file that is defined as a module in the application, use the **Replace or add a single module** option.

After selecting this option, specify whether the file is on a local or remote file system and the full path name of the file. The path provides the location of the updated file before installation. For information on **Local file system** and **Remote file system**, refer to the description of **Replace the entire application** above.

For the relative path, specify a relative path to the file that starts from the root of the `.ear` file. For example, if the file is located at `com/company/greeting.class` in module `hello.jar`, specify a relative path of `hello.jar/com/company/greeting.class`.

To replace a file, the relative path must match the path of the file to be updated in the installed application.

To add a new file to the installed application, the relative path must *not* match the path of a file in the installed application. The value specifies the desired path for the new file.

After you specify the file system and relative paths, click **Next**.

After a single file is added or updated, when configuration changes are saved, the new or updated file is stored in the deployed application in the WebSphere Application Server configuration repository. When these changes are synchronized with the node, the file is added or updated to the node's file system. If the application is running on the node when the file is added or updated, then one of the following occurs:

- When files are added at application metadata scope (`META-INF` directory) or updated at any application scope or in non-Web modules, the entire application is stopped, the file is added or updated, and then the entire application is restarted.
- When files are added at application non-metadata scope (outside of `META-INF` directory but not in any module), the changes are saved in the file system without restarting the running application.
- When files are added or updated to Web module metadata (`META-INF` or `WEB-INF` directory), the running Web module is stopped, the Web module file is added or updated, and then the Web module is started.
- For all other files in Web modules, the file is added or updated on the node's file system without stopping the application or any of its components.

Replace, add, or delete multiple files

Under **Application update options**, specifies to update multiple files of an installed application by uploading a compressed file. Depending on the contents of the compressed file, a single use of this option

can replace files in, add new files to, and delete files from the installed application. Each entry in the compressed file is treated as a single file and the path of the file from the root of the compressed file is treated as the relative path of the file in the installed application.

After selecting this option, specify whether the compressed file is on a local or remote file system and the full path name of the compressed file. You will likely use **Local file system** because you are uploading a compressed file and remote browsing only works for `.ear`, `.sar`, `.war` or `.jar` files. Specify a valid compressed file format such as `.zip` or `.gzip`. The path provides the location of the compressed file before installation. This option unzips the compressed file into the installed application directory.

Use **Local file system** if the browser and the updated files or modules are on the same machine, whether or not the server is on that machine too. **Local file system** is available for all update options.

To replace a file, a file in the compressed file must have the same relative path as the file to be updated in the installed application.

To add a new file to the installed application, a file in the compressed file must have a different relative path than the files in the installed application.

The relative path of a file in the installed application is formed by concatenation of the relative path of the module (if the file is inside a module) and the relative path of the file from the root of the module separated by /.

To remove a file from the installed application, specify metadata in the compressed file using a file named `META-INF/ibm-partialapp-delete.props` at any archive scope. The `ibm-partialapp-delete.props` file must be an ASCII file that lists files to be deleted in that archive with one entry for each line. The entry can contain a string pattern such as a regular expression that identifies multiple files. The file paths for the files to be deleted must be relative to the archive path that has the `META-INF/ibm-partialapp-delete.props` file.

Level of files to delete	Metadata .props file to include in compressed file
Application	Include `META-INF/ibm-partialapp-delete.props` in the compressed file. In the metadata `.props` file, list files to be deleted. File paths are relative to the location of the `META-INF/ibm-partialapp-delete.props` file. For example, to delete a file named `utils/config.xmi` from the root of the `my.ear` file, include the line `utils/config.xmi` in the `META-INF/ibm-partialapp-delete.props` file.
Module	Include `module_uri/META-INF/ibm-partialapp-delete.props` in the compressed file. To delete one file from a module, include the file path relative to the module in the metadata `.props` file. For example, to delete `a/b/c.jsp` from the `my.jar` module, include `a/b/c.class` in `my.jar/META-INF/ibm-partialapp-delete.props` file in the compressed file. To delete multiple files within a module, list the files to be deleted in the metadata `.props` file with one entry on each line. For example, to delete all JavaServer Pages (`.jsp` files) from the `my.war` file, include the line `.*jsp` in the `my.war/META-INF/ibm-partialapp-delete.props` file. The line uses a regular expression, `.*jsp`, to identify all `.jsp` files in `my.war`.

You can use a single partial application file to add, delete and update multiple files.

After you specify a file system path, click **Next**.

After a partial application update, when configuration changes are saved, the new or updated application file is stored in the deployed application in the WebSphere Application Server configuration repository. When these changes are synchronized with the node, the files are added or updated to the node's file system. Because the partial application option updates multiple files, the application components that are restarted are determined using individual files in the partial application.

An example of entries in a partial application compressed file follows:

```
util.jar
META-INF/ibm-partialapp-delete.props
foo.jar/com/mycomp/xyz.class
xyz.war/welcome.jsp
xyz.war/WEB-INF/web.xml
webmod.war/META-INF/ibm-partialapp-delete.props
```

For this example, the `META-INF/ibm-partialapp-delete.props` file contains the `.*.dat` and `tools/test.jar` files. The `webmod.war/META-INF/ibm-partialapp-delete.props` file contains the `com/test/.*.jsp` and `WEB-INF/test.xmi` files.

The partial application update option does the following:
- Adds or replaces `util.jar` in the deployed application.
- Adds or replaces `com/mycomp/xyz.class` inside the `foo.jar` file of the deployed application.
- Deletes `*.dat` files from the application, but not from any modules.
- Deletes `tools/test.jar` from the application.
- Adds or replaces `welcome.jsp` inside the `xyz.war` module of the deployed application.
- Replaces `WEB-INF/web.xml` inside the `xyz.war` module of the deployed application.
- Deletes `com/test/*.jsp` from the `webmod.war` module.
- Deletes `WEB-INF/test.xmi` from the `webmod.war` module.

Hot deployment and dynamic reloading

You can make various changes to applications and their modules without having to stop the server and start it again. Making these types of changes is known as *hot deployment and dynamic reloading*.

This topic assumes that your application files are deployed on a server and you want to upgrade the files.

See "Ways to update application files" on page 1420 and determine whether hot deployment is the appropriate way for you to update your application files. Other ways are easier and hot deployment is appropriate only for experienced users.

Important: Do not use hot deployment to update components in a production deployment manager managed cell. Hot deployment is well-suited for development and testing, but poses unacceptable risks to production environments. Full or partial resynchronization might erase hot deployed components. Also, running the `restoreconfig` command might overwrite changes made to expanded application files. Further, hot deployed components are not migrated between versions of WebSphere Application Server. To add new components or modules to an enterprise application, reassemble the application EAR file so it has the new components or modules and then redeploy the EAR file.

Hot deployment is the process of adding new components (such as WAR files, EJB Jar files, enterprise Java beans, servlets, and JSP files) to a running server without having to stop the application server process and start it again.

Dynamic reloading is the ability to change an existing component without needing to restart the server in order for the change to take effect. Dynamic reloading involves:

- Changes to the implementation of a component of an application, such as changing the implementation of a servlet
- Changes to the settings of the application, such as changing the deployment descriptor for a Web module

As opposed to the changes made to a deployed application described in "Updating applications" on page 1420, changes made using hot deployment or dynamic reloading do not use the administrative console or a wsadmin scripting command. You must directly manipulate the application files on the server where the application is deployed.

If the application you are updating is deployed on a server that has its application class loader policy set to `Single`, you might not be able to dynamically reload your application. At minimum, you must restart the server after updating your application.

1. Locate your expanded application files.

   The application files are in the directory you specified when installing the application or, if you did not specify a custom target directory, are in the default target directory, *app_server_root*/`installedApps/`*cell_name*. Your EAR file, `${APP_INSTALL_ROOT}/`*cell_name*/*application_name*.`ear`, points to the target directory. The `variables.xml` file for the node defines `${APP_INSTALL_ROOT}`.

   It is important to locate the expanded application files because, as part of installing applications, a WebSphere application server unjars portions of the EAR file onto the file system of the computer that will run the application. These expanded files are what the server looks at when running your application. If you cannot locate the expanded application files, look at the binariesURL attribute in the `deployment.xml` file for your application. The attribute designates the location the run time uses to find the application files.

   For the remainder of this information on hot deployment and dynamic reloading, *application_root* represents the root directory of the expanded application files.

2. Locate application metadata files. The metadata files include the deployment descriptors (`web.xml`, `application.xml`, `ejb-jar.xml`, and the like), the bindings files (`ibm-web-bnd.xmi`, `ibm-app-bnd.xmi`, and the like), and the extensions files (`ibm-web-ext.xmi`, `ibm-app-ext.xmi`, and the like).

   Metadata XML files for an application can be loaded from one of two locations. The metadata files can be loaded from the same location as the application binary files (such as *application_root*/`META-INF`) or they can be loaded from the WebSphere configuration tree, `${CONFIG_ROOT}/cells/`*cell_name*/ `applications ` /*application_EAR_name*/`deployments/`*application_name*/. The value of the useMetadataFromBinary flag specified during application installation controls which location is used. If specified, the metadata files are loaded from the same location as the application binary files. If not specified, the metadata files are loaded from the application deployment folder in the configuration tree.

   For the remainder of this information, *metadata_root* represents the location of the metadata files for the specified application or module.

3. **Required:** If you are running WebSphere Application Server on a group of machines using Network Deployment and you are changing an application on a particular node, disable automatic synchronization.

   a. Click **System administration > Node agents >** *node_agent_name* **> File synchronization service** in the console navigation tree.

   b. On the File synchronization service page, clear the check box for **Automatic synchronization** and click **OK**.

   When you run WebSphere Application Server on a group of machines using Network Deployment and you change a file on the disk in the expanded application directory for a particular node, you can lose those changes the next time node synchronization occurs. In the Network Deployment environment, the configuration stored by the deployment manager is the master copy and any changes detected between that master copy and the copy on a particular machine trigger the master copy to be downloaded to the node.

4. **Optional:** Examine the values specified for **Reload classes when application files are updated** and **Polling interval for updated files** on the settings page for your application's class loader.

   If reloading of classes is enabled and the polling interval is greater than zero (0), the application files are reloaded after the application is updated. For JavaServer Pages (JSP) files in a Web module, a Web container reloads JSP files only when the IBM extension jspReloadingEnabled in the jspAttributes of the `ibm-web-ext.xmi` file is set to `true`. You can set jspReloadingEnabled to `true` when editing your Web module's extended deployment descriptors in an assembly tool.

5. Change or add the following components or modules as needed:
   - Application files
   - WAR files
   - EJB Jar files
   - HTTP plug-in configuration files

6. For changes to take effect, you might need to start, stop, or restart an application. "Starting or stopping applications" on page 1416 provides information on using the administrative console to start, stop, or restart an application. Starting applications with scripting and Stopping applications with scripting provide information on using the wsadmin scripting tool.

7. If you disabled automatic synchronization in step 3, enable automatic synchronization again:

   a. Return to the File synchronization service page.

   b. Select **Automatic synchronization**.

   c. Click **OK**.

Changing or adding application files

You can change or add application files on application servers without having to stop the server and start it again.

There are several changes that you can make to deployed application files without stopping the server and starting it again.

Important: See "Ways to update application files" on page 1420 and determine whether hot deployment is the appropriate way for you to update your application files. Other ways are easier and hot deployment is appropriate only for experienced users. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server.

This topic describes how to make the following changes by manipulating an application file on the server where the application is deployed:
- Updating an existing application on a running server, providing a new enterprise application (EAR file)
- Adding a new application to a running server
- Removing an existing application from a running server
- Changing or adding files to existing enterprise bean (EJB) or Web modules
- Changing the `application.xml` file for an application
- Changing the `ibm-app-ext.xmi` file for an application
- Changing the `ibm-app-bnd.xmi` file for an application
- Changing a non-module Jar file contained in the EAR file

Updating an existing application on a running server (providing a new EAR file)

Reinstall an updated application using the administrative console or the wsadmin `$AdminApp install` command with the `-update` option.

Both reinstallation methods enable you to update an existing application using any of the other steps listed in this file, including changing classes, adding modules, removing modules, changing modules, or changing metadata files. The application reinstallation methods detect the changes in your application and prompt you for additional binding data that might be needed to install the application. The reinstallation process automatically stops and restarts your application on the appropriate servers.

Hot deployment   Yes
Dynamic reloading   Yes

Adding a new application to a running server

Install an application using the administrative console or the wsadmin `install` command.

Hot deployment   Yes
Dynamic reloading   No

Removing an existing application from a running server

Stop the application and then uninstall it from the server. Use the administrative console to stop the application and then uninstall it. Or run the wasadmin `stopApplication` command and then the `uninstall` command.

Hot deployment   Yes
Dynamic reloading   No

Changing or adding files to existing EJB or Web modules
1. Update the application files in the *application_root* location.
2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment   Yes
Dynamic reloading   No

Changing the application.xml file for an application

Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment   Not applicable
Dynamic reloading   Yes

Changing the ibm-app-ext.xmi file for an application

Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment   Not applicable
Dynamic reloading   Yes

Changing the ibm-app-bnd.xmi file for an application

Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment   Not applicable
Dynamic reloading   Yes

Changing a non-module Jar file contained in the EAR file
1. Update the non-module Jar file in the *application_root* location.
2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

   If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment	Yes
Dynamic reloading	Yes

Changing or adding WAR files

You can change Web application archives (WAR files) on application servers without having to stop the server and start it again.

There are several changes that you can make to WAR files without stopping the server and starting it again.

Important: See "Ways to update application files" on page 1420 and determine whether hot deployment is the appropriate way for you to update your WAR files. Other ways are easier and hot deployment is appropriate only for experienced users. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server.

This topic describes how to make the following changes by manipulating a WAR file on the server where the application is deployed:
- Changing an existing JavaServer Pages (JSP) file
- Adding a new JSP file to an existing application
- Changing an existing servlet class (editing and recompiling)
- Changing a dependent class of an existing servlet class
- Adding a new servlet using the Invoker (Serve Servlets by class name) facility or adding a dependent class to an existing application
- Adding a new servlet, including a new definition of the servlet in the `web.xml` deployment descriptor for the application
- Changing the `web.xml` file of a WAR file
- Changing the `ibm-web-ext.xmi` file of a WAR file
- Changing the `ibm-web-bnd.xmi` file of a WAR file

Changing an existing JSP file

Place the changed JSP file directly in the *application_root*/*module_name* directory or the appropriate subdirectory. The change will be automatically detected and the JSP will be recompiled and reloaded.

Hot deployment	Not applicable
Dynamic reloading	Yes

Adding a new JSP file to an existing application

Place the new JSP file directly in the *application_root*/*module_name* directory or the appropriate subdirectory. The new file will be automatically detected and compiled on the first request to the page.

Hot deployment	Yes
Dynamic reloading	Yes

Changing an existing servlet class (editing and recompiling)

1. Place the new version of the servlet `.class` file directly in the *application_root*/*module_name*/WEB-INF/`classes` directory. If the `.class` file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/`lib`. In either case, the change will be detected, the Web application will be shut down and reinitialized, picking up the new class.
2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

    If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment	Not applicable
Dynamic reloading	Yes

Changing a dependent class of an existing servlet class
1. Place the new version of the dependent `.class` file directly in the *application_root*/*module_name*/WEB-INF/`classes` directory. If the `.class` file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/`lib`. In either case, the change will be detected, the Web application will be shut down and reinitialized, picking up the new class.
2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

    If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment	Not applicable
Dynamic reloading	Yes

Adding a new servlet using the Invoker (Serve Servlets by class name) facility or adding a dependent class to an existing application
1. Place the new `.class` file directly in the *application_root*/*module_name*/WEB-INF/`classes` directory. If the `.class` file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/`lib`. In either case, the change will be detected, the Web application will be shut down and reinitialized, picking up the new class.

    This case is treated the same as changing an existing class. The difference is that adding the servlet or class does not immediately cause the Web application to reload because the class has never been loaded before. The class simply becomes available for execution.
2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

    If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment	Yes
Dynamic reloading	Not applicable

Adding a new servlet, including a new definition of the servlet in the web.xml deployment descriptor for the application
1. Place the new `.class` file directly in the *application_root*/*module_name*/WEB-INF/`classes` directory. If the `.class` file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/`lib`.

    You can edit the `web.xml` file in place or copy it into the *application_root*/*module_name*/WEB-INF/`classes` directory. The new `.class` file will not trigger a reloading of the application.
2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands. After the application restarts, the new servlet is available for service.

| **Hot deployment** | Yes |
| **Dynamic reloading** | Not applicable |

Changing the web.xml file of a WAR file
1. Edit the `web.xml` file in place or copy it into the *metadata_root*/*module_name*/`WEB-INF` directory.
2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

| **Hot deployment** | Yes |
| **Dynamic reloading** | Yes |

Changing the ibm-web-ext.xmi file of a WAR file

Edit the extension settings as needed. You can change all of the extension settings. The only warning is if you set the reloadInterval property to zero (0) or the reloadEnabled property to `false`, the application no longer automatically detects changes to class files. Both of these changes disable the automatic reloading function. The only way to re-enable automatic reloading is to change the appropriate property and restart the application. See other task descriptions in this file for information on restarting an application.

| **Hot deployment** | Not applicable |
| **Dynamic reloading** | Yes |

Changing the ibm-web-bnd.xmi file of a WAR file
1. Edit the bindings as needed. You can change all of the values but ensure that the entities you are binding to are present in the configuration of the server.
2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

| **Hot deployment** | Not applicable |
| **Dynamic reloading** | Yes |

Changing or adding EJB Jar files
You can change enterprise bean (EJB) Jar files on application servers without having to stop the server and start it again.

There are several changes that you can make to EJB Jar files without stopping the server and starting it again.

Important: See "Ways to update application files" on page 1420 and determine whether hot deployment is the appropriate way for you to update your EJB Jar files. Other ways are easier and hot deployment is appropriate only for experienced users. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server.

This topic describes how to make the following changes by manipulating an EJB file on the server where the application is deployed:
- Changing the `ejb-jar.xml` file of an EJB Jar file
- Changing the `ibm-ejb-jar-ext.xmi` or `ibm-ejb-jar-bnd.xmi` file of an EJB Jar file
- Changing the `Table.ddl` file for an EJB Jar file
- Changing the `Map.mapxmi` or `Schema.dbxmi` file for an EJB Jar file
- Updating the implementation class for an EJB file or a dependent class of the implementation class for an EJB file
- Updating the Home/Remote interface class for an EJB file
- Adding a new EJB file to an existing EJB Jar file

Changing the ejb-jar.xml file of an EJB Jar file

Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment	Not applicable
Dynamic reloading	Yes

Change the ibm-ejb-jar-ext.xmi or ibm-ejb-jar-bnd.xmi file of an EJB Jar file

Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment	Not applicable
Dynamic reloading	Yes

Changing the Table.ddl file for an EJB Jar file

Rerun the DDL file on the user database server. Changing the `Table.ddl` file has no effect on the application server and is a change to the database table schema for the EJB files.

Hot deployment	Not applicable
Dynamic reloading	Not applicable

Changing the Map.mapxmi or Schema.dbxmi file for an EJB Jar file
1. Change the `Map.mapxmi` or `Schema.dbxmi` file for an EJB Jar file.
2. Regenerate the deployed code artifacts for the EJB file.
3. Apply the new EJB Jar file to the server.
4. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment	Not applicable
Dynamic reloading	Yes

Updating the implementation class for an EJB file or a dependent class of the implementation class for an EJB file
1. Update the class file in the *application_root/module_name*.jar file.
2. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

   If automatic reloading is not enabled, restart the application of which the EJB file is a member. If the updated module is used by other modules in other applications, restart those applications as well. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment	Not applicable
Dynamic reloading	Yes

Updating the Home/Remote interface class for an EJB file
1. Update the interface class of the EJB file.
2. Regenerate the deployed code artifacts for the EJB file.
3. Apply the new EJB Jar file to the server.
4. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

If automatic reloading is not enabled, restart the application of which the EJB file is a member. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment   Not applicable
Dynamic reloading   Yes

Adding a new EJB file to an existing EJB Jar file
1. Apply the new or updated Jar file to the *application_root* location.
2. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

   If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment   Yes
Dynamic reloading   Yes

Changing the HTTP plug-in configuration
You can change the HTTP plug-in configuration without having to stop the server and start it again.

There are several change that you can make to the HTTP plug-in configuration without stopping the server and starting it again.

Important: See "Ways to update application files" on page 1420 and determine whether hot deployment is the appropriate way for you to update your HTTP plug-in configuration. Other ways are easier and hot deployment is appropriate only for experienced users.

This file describes--
* Changing the `application.xml` file to change the context root of a Web application archive (WAR file)
* Changing the `web.xml` file to add, remove, or modify a servlet mapping
* Changing the `server.xml` file to add, remove, or modify an HTTP transport or changing the `virtualhost.xml` file to add or remove a virtual host or to add, remove, or modify a virtual host alias

Changing the application.xml file to change the context root of a WAR file
1. Change the `application.xml` file.
2. If the plug-in configuration property Automatically propagate plug-in configuration file is selected for this plug-in, it is automatically regenerated whenever the `application.xml` file changes. (See Web server plug-in properties settings for information on how to set this property.) You can also run the `GenPluginCfg.bat/sh` script, or issue a wsadmin command to regenerate the plug-in configuration file.

Hot deployment   Yes
Dynamic reloading   No

Changing the web.xml file to add, remove, or modify a servlet mapping
1. Change the `web.xml` file.
2. If the plug-in configuration property Automatically propagate plug-in configuration file is selected for this plug-in, it is automatically regenerated whenever the `web.xml` file changes. (See Web server plug-in properties settings for information on how to set this property.) You can also run the `GenPluginCfg.bat/sh` script, or issue a wsadmin command to regenerate the plug-in configuration file.

   If the Web application has file serving enabled or has a servlet mapping of /, the plug-in configuration does not have to be regenerated. In all other cases a regeneration is required.

Hot deployment   Yes
Dynamic reloading   Yes

Changing the server.xml file to add, remove, or modify an HTTP transport or changing the virtualhost.xml file to add or remove a virtual host or to add, remove, or modify a virtual host alias

1. Change the `server.xml` file to add, remove, or modify an HTTP transport or change the `virtualhost.xml` file to add or remove a virtual host or to add, remove, or modify a virtual host alias.

2. If the plug-in configuration property **Automatically propagate plug-in configuration file** is selected for this plug-in, it is automatically regenerated whenever the `server.xml` file changes. (See Web server plug-in properties settings for information on how to set this property.) You can also run the `GenPluginCfg.bat/sh` script, or issue a wsadmin command to regenerate the plug-in configuration file.

Hot deployment   Yes
Dynamic reloading   Yes

Uninstalling applications

After an application no longer is needed, you can uninstall it.

Uninstalling an application deletes the application from the WebSphere Application Server configuration repository and it deletes the application binaries from the file system of all nodes where the application modules are installed.

1. Click **Applications > Enterprise Applications** in the administrative console navigation tree to access the Enterprise Applications page.

2. If you need to retain a copy of the application, back up the application.

    a. Select the application you want uninstalled.

    b. Click **Export**.

    The application is exported to an enterprise application (`.ear` file), preserving the binding information.

3. Uninstall the application.

    a. Select the application you want uninstalled.

    b. Click **Uninstall**.

4. Save changes made to the administrative configuration.

Application binaries are deleted when configuration changes on the deployment manager synchronize with configurations for individual nodes.

Removing a file

After a file is no longer needed, you can remove the file from an application or module deployed on a server.

Removing a file deletes the file from the WebSphere Application Server configuration repository and it deletes the file from the file system of all nodes where the file is installed.

- Remove a file from an application.

    1. Go to the Enterprise Applications page. Click **Applications > Enterprise Applications** in the console navigation tree.

    2. Select the application that contains a file you want removed.

    3. Click **Remove File**. The Remove a file page is displayed

    4. Select the URI of the file that you want removed from the application.

    5. Back up the application.

        Under **Export before removing file**, select the application name and then specify the location to which you want the file exported.

6. Click **OK** to remove the file.
- Remove a file from a module.
  1. Go to the Manage modules page.

     Click **Applications > Enterprise Applications >** *application_name* **> Manage modules** in the console navigation tree.
  2. Select the module from which you want to delete a file.
  3. Click **Remove File**. The Remove a file from a module page is displayed.
  4. Select the URI of the file that you want removed from the module.
  5. Back up the application.

     Under **Export before removing file**, select the application name and then specify the location to which you want the file exported.
  6. Click **OK** to remove the file.

The file is exported to the designated location and removed from the application or module. The application or standalone Web module that had a file removed is restarted so the changes take effect.

If the application or module is deployed on a cluster and you have no more configuration changes to make, click **Rollout Update** on the Enterprise Applications page to propagate the changed configuration on all cluster members of the cluster on which the application or module is deployed. **Rollout Update** sequentially updates the configuration on the nodes that contain cluster members.

Save the changes to your administrative configuration. Application binaries are deleted when configuration changes on the deployment manager synchronize with configurations for individual nodes.

Common deployment framework

The *common deployment framework* enables you to implement plug-ins that add steps to default Java 2 Platform, Enterprise Edition (J2EE) application management operations such as install, uninstall, edit and update.

Using the framework, you can implement management operations on specific types of deployable contents. For example, the deployable contents might include EAR, WAR, JAR or other J2EE modules and the management operations might include install and uninstall. Each operation is divided into a number of steps. For example, the install operation has steps for EJBDeploy and JavaServer Pages (JSP) compilation, among others. Using the common deployment framework, you can add steps to the default logic for J2EE operations.

Version 6.1 supports framework plug-ins that extend deployment of EAR files. An EAR file has operations such as createEarWrapper, installApplication, uninstallApplication and editApplication. Using a framework plug-in, you can add steps to default install operations that support, for example, creating additional configuration artifacts in a configuration session, modifying an input EAR file using code generation, or additional validating of input parameters.

To extend application management operations using the framework, a plug-in must do the following:
- Implement each step.

  A *step* runs logic that performs an operation. A step can access the deployment context and the deployable object. The *deployment context* provides information such as the operation name, the configuration session identifier, the temporary location for creating temporary files, operations parameters, and the like. A step is added by the extension provider.
- Implement an extension provider that adds each implemented step.

  An *extension provider* is a class that provides steps for an operation on a given type. For Version 6.1, it is the EAR file type.

- Register the plug-in with a WebSphere Application Server server.

  The plug-in is implemented as an Eclipse plug-in and is placed in *app_server_root*/`plugins` directory. Add the extension point for the extension provider in the `META-INF/plugin.xml` file within the plug-in JAR file.

For an example of these steps, refer to Extending application management operations through programming.

Deploying and administering applications: Resources for learning

Use the following links to find relevant supplemental information about deploying and administering applications using the administrative console. 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.

Refer to "Web resources for learning" on page 15 for links to information applicable to WebSphere Application Server generally, such as lists of IBM technical papers, Redbooks and samples.

View links to additional information about:
- "Programming model and decisions"
- "Programming instructions and examples"
- "Administration"

Programming model and decisions
- Designing Enterprise Applications with the Java^{TM} 2 Platform, Enterprise Edition, Second Edition
- The J2EE^{TM} Tutorial
- Building Java^{TM} Enterprise Applications Volume I: Architecture
- Recommended reading list: J2EE and WebSphere Application Server

Programming instructions and examples
- *IBM WebSphere: Deployment and Advanced Configuration*, Roland Barcia, *et al.*, ISBN 0131468626 (Prentice Hall, 2004)
- *IBM WebSphere Developer Technical Journal:* Co-hosting multiple versions of J2EE applications
- Automated Deployment of Enterprise Application Updates: Part 1 - Basic concepts
- *IBM WebSphere Developer Technical Journal:* The top 10 (more or less) J2EE best practices

Administration
- *IBM WebSphere Developer Technical Journal:* System management for WebSphere Application Server V6 -- Part 1 Overview of system management enhancements
- *IBM WebSphere Developer Technical Journal:* System management for WebSphere Application Server V6 -- Part 5: Flexible options for updating deployed applications
- WebSphere Application Server V6 System Management & Configuration Handbook
- WebSphere Application Server Network Deployment V6: High Availability Solutions
- WebSphere Application Server V6 Migration Guide
- WebSphere Version 6 Web Services Handbook Development and Deployment
- Listing of all IBM WebSphere Application Server Redbooks

Chapter 10. Troubleshooting deployment

- Select the problem you are having with deploying or installing developed code for WebSphere Application Server.
  - Errors or problems deploying, installing, or promoting applications
  - Class loader exceptions
- To troubleshoot other deployment issues, use the following resources.
  - 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.
  - If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see Troubleshooting help from IBM.

Errors or problems deploying, installing, or promoting applications

This topic describes problems that you might encounter when deploying, installing, or promoting applications and suggests ways to resolve the problems.

What kind of problem are you having?
- "I installed my application using the wsadmin tool, but the application does not display under Applications > Enterprise Applications " on page 1442
- "Unable to save a deployed application" on page 1442
- "I get a java.lang.RuntimeException: Failed_saving_bytes_to_wor_ERROR_ error in the assembly tool, administrative console or the wsadmin tool." on page 1442
- "WASX7015E error running wsadmin command $AdminApp installInteractive or $AdminApp install" on page 1443
- "Data definition language (DDL) generated by an assembly tool throws SQL error on target platform " on page 1443
- "Error message ADMA0004E: Validation error in task Specifying the Default Datasource for EJB Modules returned when installing application using the administrative console or the wsadmin tool" on page 1443
- "Cannot load resource WEB-INF/ibm-web-bnd.xmi in archive file" on page 1444
- "Error message No valid target is specified in ObjectName anObject for module module_name from installation " on page 1444
- "addNode -includeapps option does not appear to upload all applications to the deployment manager" on page 1445
- ""Timeout!!!" error displays when attempting to install an enterprise application in the administrative console " on page 1445
- "I get a NameNotFoundException message when deploying an application that contains an EJB module" on page 1445
- "During application installation, the call to EJB deploy throws an exception" on page 1445
- "I get compilation errors and EJB deploy fails when installing an EJB JAR file generated for Version 5.x or earlier" on page 1445
- "While uploading documents, addNode -includeapps fails with an OutOfMemoryError exception" on page 1446

Check the following first:
- Verify that the logical name that you have specified to appear on the console for your application, enterprise bean module or other resource does not contain invalid characters such as these: - / \ : * ? " < > |.

- If the application was installed using the wsadmin $AdminApp install command with the **-local** flag, restart the server or rerun the command without the `-local` flag.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, check to see if the problem is identified and documented by looking at available online support including hints and tips, technotes, and fixes. If the problem has not been identified, see Troubleshooting help from IBM.

I installed my application using the wsadmin tool, but the application does not display under Applications > Enterprise Applications

The application might be installed but you have not saved the configuration:
1. Verify that the application subdirectory is located under the *app_server_root*/installedApps directory.
2. Run the $AdminApp list command and verify that the application is not among those displayed.
   - In the bin directory, run the wsadmin.bat or wsadmin.sh command.
   - From the wsadmin prompt, enter `$AdminApp list` and verify that the problem application is not among the items that display.
3. Reinstall your application using the wsadmin tool. Run the $AdminConfig save command in the wsadmin tool before exiting.

Unable to save a deployed application

If you are unable to save a deployed application, the problem might be that too many files are opened, exceeding the limit of the operating system.

On the SuSE9 or other Linux platform, you can either increase the number of files that can be opened to resolve the problem or you can modify the application to close files with disciplines. To increase the number of files that you can open at the same time, run the following command in the shell before invoking the process that needs to open a number of files:

```
ulimit -n number_of_files
```

Only root has authority to adjust the maximum number of files for each process. Complete the following steps to modify the application to close files with disciplines:

1. After you open a file and complete your work, call the close method of the file to release the file handle back to the operating system.

2. Using the java.io.FileInputStream and the FileOutputStream classes as examples, you can invoke their close method to release any system resources that are associated with the stream.

I get a `java.lang.RuntimeException: Failed_saving_bytes_to_wor_ERROR_` error in the assembly tool, administrative console or the wsadmin tool.

If you see this error when attempting to generate deployed code in an assembly tool, installing an application or module in the administrative console, or using the wsadmin tool to install an application or module, the file path length of the temporary system file might be exceeded. This situation is typically an issue only on Windows platforms.

To verify this problem, check the `TEMP` and `TMP` environment variables for your system. Long environment variables add path length to the file names accessed by the EJB deployment tool.

To resolve the problem:
1. Stop all WebSphere Application Server processes and close all DOS prompts.
2. Set the `TMP` and `TEMP` environment variables to something short, for example `C:\TMP` and `C:\TEMP`.
3. Reinstall the application.

Otherwise, try rebooting and redeploying or reinstalling the application.

WASX7015E error running wsadmin command `$AdminApp installInteractive` or `$AdminApp install`

This problem has two possible causes:
* If the full text of the error is similar to:

```
WASX7015E: Exception running command: "$AdminApp installInteractive C:/Documents and Settings/
myUserName/Desktop/MyApp/myapp.ear"; exception information:
com.ibm.bsf.BSFException: error while
eval'ing Jacl expression: can't find method "installInteractive"
with 3 argument(s) for class
"com.ibm.ws.scripting.AdminAppClient"
```

  The file and path name are incorrectly specified. In this case, since the path included spaces, it was interpreted as multiple parameters by the wsadmin program.

  Enter the path of the `.ear` file correctly. In this case, by enclosing it in double quotes:

```
$AdminApp installInteractive "C:\Documents
and Settings\myUserName\Desktop\MyApps\myapp.ear"
```

* If the full text of the error is similar to:

```
WASX7015E: Exception running command: "$AdminApp installInteractive c:\MyApps\myapp.ear ";
exception information: com.ibm.ws.scripting.ScriptingException:  WASX7115E:
Cannot read input file
"c:\WebSphere\AppServer\bin\MyAppsmyapp.ear"
```

  The application path is incorrectly specified. In this case, you must use ″forward-slash″ (/) separators in the path.

Data definition language (DDL) generated by an assembly tool throws SQL error on target platform

If you receive SQL errors in attempting to execute data definition language (DDL) statements generated by an assembly tool on a different platform, for example if you are deploying a container-managed persistence (CMP) enterprise bean designed on Windows onto a UNIX operating system server, try the following actions:
* Browse the DDL statements for dependencies on specific user identifiers and passwords, and correct as necessary.
* Browse the DDL statements for dependencies on specific server names, and correct as necessary.
* Refer to the message reference of the vendor for causes and suggested actions regarding specific SQL errors. For IBM DB2, you can view the message references online at http://www.ibm.com/cgi-bin/db2www/data/db2 /udb/winos2unix/support/index.d2w/report.

If you receive the following error after executing a DDL file created on the Windows operating system or on operating systems such as AIX or Linux, the problem might come from a difference in file formats:

```
SQL0104N  An unexpected token "CREATE TABLE AGENT  (COMM DOUBLE,  PERCENT  DOUBLE,  P"
was found following "       ".  Expected tokens may include:  " ".
SQLSTATE=42601
```

To resolve this problem:
* For operating systems other than Linux, edit the DDL in the vi editor, removing the Ctl-M character at the beginning of each line.
* For Linux systems, regenerate the deployment code for the application `EAR` file on a Linux platform.

Error message `ADMA0004E: Validation error in task Specifying the Default Datasource for EJB Modules` returned when installing application using the administrative console or the wsadmin tool

If you see the following error when trying to install an application through the administrative console or the wsadmin command prompt:

```
AppDeploymentException: [ADMA0014E: Validation failed.
ADMA0004E: Validation error in task Specifying the Default Datasource for
EJB Modules  JNDI name is not
specified for module beannameBean Jar with URI filename.jar,META-INF/ejb-jar.xml.
You have not specified the
data source for each CMP bean belonging to this module. Either specify the data
source for each CMP beans or
specify the default data source for the entire module.]
```

one possible cause is that in WebSphere Application Server version 4.0, it was mandatory to have a data source defined for each CMP bean in each JAR. In versions 5.0 and later releases, you can specify either a data source for a container-managed persistence (CMP) bean or a default data source for all CMP beans in the JAR file. Thus during installation interaction, such as the installation wizard in the administrative console, the data source fields are optional, but the validation performed at the end of the installation checks to see that at least one data source is specified.

To correct this problem, step through the installation again, and specify either a default data source or a data source for each CMP-type enterprise bean. If you are using the wsadmin tool:
- Use the `$AdminApp installInteractive` *filename* command to receive prompts for data sources during installation, or to provide them in a response file.
- Specify data sources as an option to the `$AdminApp install` command. For details on the syntax, see Installing applications with the wsadmin tool.

Cannot load resource `WEB-INF/ibm-web-bnd.xmi` in archive file

The Web application `tmp.war` installs on WebSphere Application Server versions 5.0 and 5.1, but fails on a WebSphere Application Server version 6 server. The application fails to install because the `WEB-INF/ibm-web-bnd.xmi` file contains `xmi` tags that the underlying WCCM model no longer recognizes.

The following error messages display:
```
IWAE0007E Could not load resource "WEB-INF/ibm-web-bnd.xmi" in archive "tmp.war"
[2/24/05 14:53:10:297 CST] 000000bc SystemErr     R
AppDeploymentException:
com.ibm.etools.j2ee.commonarchivecore.exception.ResourceLoadException:
IWAE0007E Could not load resource "WEB-INF/ibm-web-bnd.xmi" in archive "tmp.war"
[2/24/05 14:53:10:297 CST] 000000bc SystemErr     R
com.ibm.etools.j2ee.commonarchivecore.exception.ResourceLoadException:
IWAE0007E Could not load resource "WEB-INF/ibm-web-bnd.xmi" in archive "tmp.war"
!Stack_trace_of_nested_exce!
com.ibm.etools.j2ee.exception.WrappedRuntimeException: Exception occurred loading
WEB-INF/ibm-web-bnd.xmi
!Stack_trace_of_nested_exce!
```

To work around this problem, remove the `xmi:type=EJBLocalRef` tag from the `ibm-web-bnd.xmi` file. Removing this tag does not affect the application because the tag was previously used for matching the cross document reference type. The application now works for the WebSphere Application Server v5.1, v6.0, and later releases.

Error message `No valid target is specified in ObjectName` *anObject* `for module` *module_name* `from installation`

This error can occur in a clustered environment if the target cell, node, server or cluster into which the application is to be installed is incorrectly specified. For example, it can occur if the target is misspelled.

To correct this problem, check the target names against the actual WebSphere Application Server topology and reenter them with corrections.

addNode -includeapps option does not appear to upload all applications to the deployment manager

This error can occur when some or all applications on the target node are already uploaded to the deployment manager. The addNode program detects which applications are already installed and does not upload them again.

Use the administrative console to browse the deployment manager configuration and see the applications that are already installed.

″Timeout!!!″ error displays when attempting to install an enterprise application in the administrative console

This error can occur if you attempt to install an enterprise application that has not been deployed.

To correct this problem:
* Open the *file_name*.ear file in an assembly tool and then click **Deploy**. This action creates a file with a name like Deployed_*file_name*.ear.
* In the administrative console, install the deployed .ear file.

I get a NameNotFoundException message when deploying an application that contains an EJB module

If you specify that EJB deploy be run during application installation and the installation fails with a NameNotFoundException message, ensure that the input JAR or EAR file does not contain source files. If there are source files in the input JAR or EAR file, the EJB deployment tools runs a rebuild before generating the deployment code.

To work around this problem, either remove the source files or include all dependent classes and resource files on the class path. Otherwise, the source files or the lack of access to dependent classes and resource files might cause problems during rebuilding of your application on the server.

During application installation, the call to EJB deploy throws an exception

When you specify that the EJB deployment tool be run during application installation and if installation fails with the error command line too long, the problem is that the deployment command generated during installation exceeds the character limit for a command line on the Windows platform. This problem occurs only on Windows platforms.

To work around this problem, you can reduce the length of the EAR file name, reduce the length of the JAR file name within the EAR file, reduce the class path or other options specified for deployment, or change the %TEMP% location of the Windows system to make its path shorter.

I get compilation errors and EJB deploy fails when installing an EJB JAR file generated for Version 5.x or earlier

When installing an old application that uses EJB modules that were built to run on WebSphere Application Server Version 5.x or earlier, compilation errors result and EJB deploy fails. The EJB JAR file contains Java source for the old generated code. The old Java source was generated for Version 5.x or before but, when deployed to a WebSphere Application Server Version 6.x product, it is compiled using the Version 6.x run-time JAR files.

To work around this problem, remove all .java files from the application .ear file. After the Java source files are removed, you can deploy the application onto a server successfully.

While uploading documents, addNode -includeapps fails with an OutOfMemoryError exception

This error can occur when you use addNode -includeapps while you are installing applications with large EAR files. To correct this problem:

- If you are using addNode to add a node from the base server, modify the addNode script to include the following parameter:

  -Xmx_size_

- If you are adding a node from the administrative console, increase the _maximumHeapSize_ in the Java virtual machine settings of the deployment manager, then restart the deployment manager. See addNode command and Java virtual machine settings for details.

For example, the addNode.bat file that follows sets a maximum heap size of 512 MB on a Windows platform:

```
"%JAVA_HOME%\bin\java" -Xmx512m %DEBUG% %WAS_TRACE% %CONSOLE_ENCODING%
"%CLIENTSOAP%" "%CLIENTSAS%" "-classpath" "%WAS_CLASSPATH%"
"-Dws.ext.dirs=%WAS_EXT_DIRS%" %USER_INSTALL_PROP%
-Dwas.install.root=%WAS_HOME%" "com.ibm.ws.bootstrap.WSLauncher"
"com.ibm.ws.management.tools.NodeFederationUtility" "%CONFIG_ROOT%" "%WAS_CELL%"
"%WAS_NODE%" %*
```

Troubleshooting testing and first time run problems

Select the problem you are having with testing or the first run of deployed code for WebSphere Application Server:
- The server process does not start or starts with errors.
- "The application does not start or starts with errors" on page 1451.
- "A Web resource does not display" on page 1453.
- Cannot access a data source.
- "Cannot access an enterprise bean from a servlet, a JSP file, a stand-alone program, or another client" on page 151.
- Cannot look up an object hosted by WebSphere Application Server from a servlet, JSP file, or other client.
- Access problems after enabling security.
- Errors after enabling security.
- Errors after configuring or enabling Secure Sockets Layer.
- Errors in messaging.
- Errors returned to a client sending a SOAP request.
- A client program does not work.
- Errors connecting to WebSphere MQ and creating WebSphere MQ queue connection factory.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see Troubleshooting help from IBM.

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 Must gather documents page for information to gather to send to IBM Support page.

Errors starting an application

Use this information for troubleshooting problems that occur when starting an application.

What kind of error do you see when you start an application?
- "HTTP server and Application Server are working separately, but requests are not passing from HTTP server to Application Server"
- "File serving problems" on page 1448
- "Graphics do not appear in the JSP file or servlet output" on page 1448
- "SRVE0026E: [Servlet Error]-[Unable to compile class for JSP file" on page 1449
- "After modifying and saving a JSP file, the change does not show up in the browser (the old JSP file displays)" on page 1450
- "Message like ″Message: /jspname.jsp(9,0) Include: Mandatory attribute page missing″ appears when attempting to browse JSP file" on page 1450
- "The Java source generated from a JSP file is not retained in the temp directory (only the class file is found)" on page 1450
- "The JSP Batch Compiler fails with the message ″Enterprise Application [application name you typed in] not found.″" on page 1450
- "There is a translation problem with non-English browser input" on page 1450
- "Scroll bars do not appear around items in the browser window" on page 1451
- "Error ″Page cannot be displayed... server not found or DNS error″ appears when attempting to browse a JavaServer Pages (JSP) file using Internet Explorer" on page 1451

HTTP server and Application Server are working separately, but requests are not passing from HTTP server to Application Server

If your HTTP server appears to be functioning correctly, and the Application Server also works on its own, but browser requests sent to the HTTP server for pages are not being served, a problem exists in the WebSphere Application Server plug-in.

In this case:

1. Determine whether the HTTP server is attempting to serve the requested resource itself, rather than forwarding it to the WebSphere Application Server.
   a. Browse the HTTP server access log (*IHS install root*/logs/access.log for IBM HTTP Server). It might indicate that it could not find the file in its own document root directory.
   b. Browse the plug-in log file as described below.

2. Refresh the plugin-cfg.xml file that determines which requests sent to the HTTP server are forwarded to the WebSphere Application Server, and to which Application Server.

   Use the console to refresh this file:
   - In the WebSphere Application Server administrative console, expand the Environment tree control.
   - Click **Update WebSphere Plugin**.
   - Stop and restart the HTTP server.
   - Retry the Web request.

3. Browse the *plugin_install_root*/logs/*web_server_name*/http_plugin.log file for clues to the problem. Make sure the timestamps with the most recent plug-in information stanza, which is printed out when the plug-in is loaded, correspond to the time the Web server started.

4. Turn on plug-in tracing by setting the LogLevel attribute in the plugin-cfg.xml file to Trace and reloading the request. Browse the *plugin_install_root*/logs/*Web_server_name*/http_plugin.log file. You should be able to see the plug-in attempting to match the request URI with the various URI definitions for the routes in the plugin-cfg.xml. Check which rules the plug-in is not matching against and then figure out if you need to add additional ones. If you just recently installed the application you might need to manually regenerate the plug-in configuration to pick up the new URIs related to the new application.

For further details on troubleshooting plug-in-related problems, see Web server plug-in troubleshooting tips.

File serving problems

If text output appears on your JSP- or servlet-supported Web page, but image files do not:

- Verify that your files are in the right place: the **document root** directory of your Web application WebSphere Application Server follows the J2EE standard, which means that the document root is the *Web_module_name*.war directory of your deployed Web application.

  Typically this directory will be found in the *install_root*/installedApps/*nodename*/*appname*.ear directory or *install_root*/installedApps/*nodename*/*appname*Network.ear directory.

  If the files are in a subdirectory of the document root, verify that the reference to the file reflects that. That is, if the invoices.html file is stored in Windows directory *Web_module_name*.war\invoices, then links from other pages in the Web application to display it should read "invoices\invoices.html", not "invoices.html".

- Verify that your Web application is configured to enable file serving (in other words, that it is enabled to display static resources like image and .html files):
  1. View the file serving property of the hosting Web module by browsing the source .war file in an assembly tool. If necessary, update the property and redeploy the module.
  2. Edit the **fileServingEnabled** property in the deployed Web application ibm-web-ext.xmi configuration file.

     The file typically is found in the *install_root*/config/cells/*nodename* or *nodename*Network/ applications/*application_name*/deployments/*application name*/*Web_module_name*/web-inf directory.

Graphics do not appear in the JSP file or servlet output

If text output appears on your JSP- or -servlet-supported Web page, but image files do not:

- Verify that your graphic files are in the right place: the **document root** directory of your Web application. WebSphere Application Server Version 5 follows the J2EE standard, which means that the document root is the *Web_module_name*.war directory of your deployed Web application.

  Typically, this directory is found in the *install_root*/installedApps/*nodename*/*appname*.ear directory or *install_root*/installedApps/*nodename*/*appname*Network.ear directory.

  If the graphics files are in a subdirectory of the document root, verify that the reference to the graphic reflects that; for example, if the banner.gif file is stored in Windows directory *Web_module_name*.war/ images, the tag to display it should read: **<img SRC="images/banner.gif">**, not <img SRC="banner.gif">.

- Verify that your Web application is configured to enable file serving (that is, display of static resources like image and .html files).
  1. View the file serving property of the hosting Web module by browsing the source .war file in an assembly tool. If necessary, update the property and re-deploy the module.
  2. Edit the **fileServingEnabled** property in the deployed Web application ibm-web-ext.xmi configuration file.

     The file typically is found in the *install_root*/config/cells/*nodename* or *nodename*Network/ applications/*application_name*/deployments/*application name*/*Web_module_name*/web-inf directory.
  3. After completing the previous step:
     - In the administrative console, expand the **Environment** tree control .
     - Click **Update WebSphere Plugin**.
     - Stop and restart the HTTP server and retry the Web request.

SRVE0026E: [Servlet Error]-[Unable to compile class for JSP file

If this error appears in a browser when trying to access a new or modified .jsp file for the first time, the most likely cause is that the JSP file Java source failed (was incorrect) during the javac compilation phase.

Check the SystemErr.log file for a compiler error message, such as:

```
C:\WASROOT\temp\ ... test.war\_myJsp.java:14: \Duplicate variable declaration: int myInt was int myInt
int myInt = 122;
String myString = "number is 122";
static int myStaticInt=22;
int myInt=121;
       ^
```

Fix the problem in the JSP source file, save the source and request the JSP file again.

If this error occurs when trying to serve a JSP file that was copied from another system where it ran successfully, then there is something different about the new server environment that prevents the JSP file from running. Browse the text of the error for a statement like:

```
Undefined variable or class name: MyClass
```

This error indicates that a supporting class or jar file is not copied to the target server, or is not on the class path. Find the MyClass.class file, and place it on the Web module WEB-INF/classes directory, or place its containing .jar file in the Web module WEB-INF/lib directory.

Verify that the URL used to access the resource is correct by doing the following:

- For a JSP file, `html` file, or image file: **`http://host_name/Web_module_context_root/subdir under doc root, if any/filename.ext`**. The document root for a Web application is the `application_name`.WAR directory of the installed application.
  - For example, to access the myJsp.jsp file, located in c:\WebSphere\ApplicationServer\installedApps\ myEntApp.ear\myWebApp.war\invoices on myhost.mydomain.com, and assuming the context root for the myWebApp Web module is myApp, the URL is `http://myhost.mydomain.com/myApp/invoices/ myJsp.jsp`.
  - JSP serving is enabled by default. File serving for HTML and image files must be enabled as a property of the Web module, in an assembly tool, or by setting the **`fileServingEnabled`** property to **true** in the `ibm-web-ext.xmi` file of the installed Web application and restarting the application.
- For servlets served by class name, the URL is `http://hostname/Web_module_context_root/servlet/packageName.className`.

  For example, to access myCom.myServlet.class, located in c:\WebSphere\ApplicationServer\ installedApps\ myEntApp.ear\myWebApp.war\WEB-INF\classes, and assuming the context root for the myWebApp module is ″myApp″, the URL would be `http://myhost.mydomain.com/myApp/servlet/ myCom.MyServlet`.
- Serving servlets by class name must be enabled as a property of the Web module, and is enabled by default. File serving for HTML and image files must be enabled as a property of the Web application, in an assembly tool, or by setting the **`fileServingEnabled`** property to **true** in the **`ibm-web-ext.xmi`** file of the installed Web application and restarting the application.

Correct the URL in the ″from″ HTML file, servlet or JSP file. An HREF with no leading slash (/) inherits the calling resource context. For example:
- an HREF in `http://[hostname]/myapp/servlet/MyServlet` to ″ServletB″ resolves to ″http://hostname/myapp/servlet/ServletB″
- an HREF in `http://[hostname]/myapp/servlet/MyServlet` to ″servlet/ServletB″ resolves to ″http://hostname/myapp/servlet/servlet/ServletB″ (an error)
- an HREF in `http://[hostname]/myapp/servlet/MyServlet` to ″/ServletB″ resolves to ″http://hostname/ServletB″ (an error, if `ServletB` requires the same context root as `MyServlet`)

After modifying and saving a JSP file, the change does not show up in the browser (the old JSP file displays)

It is probable that the Web application is not configured for servlet reloading, or the reload interval is too high.

To correct this problem, in an assembly tool, check the **Reloading Enabled** flag and the **Reload Interval** value in the IBM Extensions for the Web module in question. Enable reloading, or if it is already enabled, then set the Reload Interval lower.

Message like ″Message: /jspname.jsp(9,0) Include: Mandatory attribute page missing″ appears when attempting to browse JSP file

It is probable that the JSP file failed during the translation to Java phase. Specifically, a JSP directive, in this case an Include statement, was incorrect or referred to a file that could not be found.

To correct this problem, fix the problem in the JSP source, save the source and request the JSP file again.

The Java source generated from a JSP file is not retained in the temp directory (only the class file is found)

It is probable that the JSP processor is not configured to keep generated Java source.

In an assembly tool, check the **JSP Attributes** under **Assembly Property Extensions** for the Web module in question. Make sure the **keepgenerated** attribute is there and is set to `true`. If not, set this attribute and restart the Web application. To see the results of this operation, delete the class file from the temp directory to force the JSP processor to translate the JSP source into Java source again.

The JSP Batch Compiler fails with the message ″Enterprise Application [application name you typed in] not found.″

It is probable that the full enterprise application path and name, starting with the `.ear` subdirectory that resides in the `applications` directory is expected as an argument to the JspBatchCompiler tool, not just the display name.

The directory path is *install_root*\config\cells*node_name*Network\applications.

For example:
- ″JspBatchCompiler -enterpriseapp.name sampleApp.ear/deployments/sampleApp″ is correct, as opposed to
- ″JspBatchCompiler -enterpriseapp.name sampleApp″, which is incorrect.

There is a translation problem with non-English browser input

If non-English-character-set browser input cannot be translated after being read by a servlet or JSP file, ensure that the request parameters are encoded according to the expected character set before reading. For example, if the site is Chinese, the target `.jsp` file should have a line:

```
req.setCharacterEncoding("gb2312");
```

before any req.getParameter method calls.

This problem affects servlets and `jsp` files ported from earlier versions of WebSphere Application Server, which converted characters automatically based upon the locale of the WebSphere Application Server.

Scroll bars do not appear around items in the browser window

In some browsers, tree or list type items that extend beyond their allotted windows do not have scroll bars to permit viewing of the entire list.

To correct this problem, right-click on the browser window and click **Reload** from the menu.

Error ″Page cannot be displayed... server not found or DNS error″ appears when attempting to browse a JavaServer Pages (JSP) file using Internet Explorer

This error can occur when an HTTP timeout causes the servant to be brought down and restarted. To correct this problem, increase the ConnectionIOTimeOut value:

1. From the administrative console, select **System administration > Deployment manager > Administration Services > Custom Properties**
2. Select ConnectionIOTimeOut.
3. Increase the ConnectionIOTimeOut value.
4. Click **OK**.

The application does not start or starts with errors

When an application is not starting or starting with errors, the problem could be from one of various sources.

What kind of error do you see when you start an application?
- A "java.lang.ClassNotFoundException: classname Bean_AdderServiceHome_04f0e027Bean" error occurs
- A "ConnectionFac E J2CA0102E: Invalid EJB component: Cannot use an EJB module with version 1.1 using The Relational Resource Adapter" on page 1452 error occurs
- "NMSV0605E: ″A Reference object looked up from the context...″ error when starting an application" on page 1452.
- "Deployed application that uses Java Native Interface (JNI) code does not start" on page 1453
- A parsing error when running an application that uses the JSF configuration occurs.

If none of these errors match the error you see:
- Browse the log files of the application server for this application looking for clues. By default, these files are: *install_dir*/logs/*server_name*/SystemErr.log and SystemOut.log.
- Look up any error or warning messages in the message reference table by clicking the Reference view and expanding the ″Messages″ heading.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see Troubleshooting help from IBM.

java.lang.ClassNotFoundException: *classname* Bean_AdderServiceHome_04f0e027Bean

An similar exception occurs when you try to start an undeployed application containing enterprise beans, or containing undeployed enterprise bean modules.

Enterprise JavaBeans modules created in an assembly tool intentionally have incomplete configuration information. Deploying these modules completes the configuration by reading the module's deployment descriptor and completing platform- or installation-dependent settings and adding related classes to the Enterprise JavaBeans JAR file.

To avoid this problem, do the following:

- Use an assembly tool and administrative console to generate deployment code and install the application or Enterprise JavaBeans module onto a server.
  1. Uninstall the application or Enterprise JavaBeans module in the administrative console.
  2. Configure your assembly tool so the target server is a WebSphere Application Server installation such as **WebSphere Application Server v6**. If you do not have access to the target server, you can specify a false location such as `c:\temp`. Specifying a false location enables you to assemble and generate deployment code for the enterprise bean.
  3. In the Project Explorer view of an assembly tool, right-click the enterprise bean (Enterprise JavaBeans) in the undeployed .ear file containing the Enterprise JavaBeans module or the standalone undeployed Enterprise JavaBeans JAR file, and click **Deploy**. If your assembly tool can access the WebSphere Application Server target server, deployment code is generated for the Enterprise JavaBeans and the assembly tool attempts to install the application or module onto the target server. If your assembly tool cannot access the WebSphere Application Server target server or the installation fails, use the deployment code that is generated for the next step.

     For information on using an assembly tool, refer to Chapter 7, "Assembling applications," on page 1345.
  4. Use the `wsadmin $AdminApp install` command or the administrative console to install the deployed version created by the assembly tool.
- If you use the `wsadmin $AdminApp install` command, uninstall it and then reinstall using the `-EJBDeploy` option. Follow the install command with the `$AdminConfig` save command.

ConnectionFac E J2CA0102E: Invalid EJB component: Cannot use an EJB module with version 1.1 using The Relational Resource Adapter

This error occurs when an enterprise bean developed to the Enterprise JavaBeans 1.1 specification is deployed with a WebSphere Application Server V5 J2C-compliant data source, which is the default data source. By default, persistent enterprise beans created under WebSphere Application Server V4.0's using the Application Assembly Tool fulfill the Enterprise JavaBeans 1.1 specification. To run on WebSphere Application Server V6, these enterprise beans must be associated with a WebSphere Application Server V4.0-type data source.

Either modify the mapping in the application of enterprise beans to associate 1.x container managed persistence (CMP) beans to associate them with a V4.0 data source or delete the existing data source and create a V4.0 data source with the same name.

To modify the mapping in the application of enterprise beans, in the WebSphere Application Server administrative console, select the properties for the problem application and use **map resource references to resources** or **Map data sources for all 1.x CMP beans** to switch the data source the enterprise bean uses. Save the configuration and restart the application.

To delete the existing data source and create a V4.0 data source with the same name:
1. In the administrative console, click **Resources**>**Manage JDBC Providers**>*JDBC_provider_name*>**Data sources**.
2. Delete the data source associated with the Enterprise JavaBeans 1.1 module.
3. Click **Resources**>**Manage JDBC Providers**>*JDBC_provider_name*>**Data sources (Version 4)**.
4. Create the data source for the Enterprise JavaBeans 1.1 module.
5. Save the configuration and restart the application.

NMSV0605E: ″A Reference object looked up from the context...″ error when starting an application

If the full text of the error is similar to:

```
[7/17/02 15:20:52:093 CDT] 5ae5a5e2 UrlContextHel W NMSV0605E: A Reference object looked up from the context
    "java:" with the name "comp/PM/WebSphereCMPConnectionFactory" was sent to the JNDI Naming Manager
    and an exception resulted. Reference data follows:
    Reference Factory Class Name: com.ibm.ws.naming.util.IndirectJndiLookupObjectFactory
```

```
        Reference Factory Class Location URLs:
        Reference Class Name: java.lang.Object
        Type: JndiLookupInfo
        Content: JndiLookupInfo: ; jndiName="eis/jdbc/MyDatasource_CMP"; providerURL=""; initialContextFactory=""
```

then the problem might be that the data source intended to support a CMP enterprise bean is not correctly associated with the enterprise bean.

To resolve this problem:
1. Select the **Use this Data Source in container managed persistence (CMP)** check box in the data source ″General Properties″ panel of the administrative console.
2. Verify that the JNDI Name given in administrative console under **Resources-> Manage JDBC Provider > DataSource > JNDI Name** for DataSource matches the JNDI Name given for CMP or BMP Resource Bindings at the time of assembling the application in an assembly tool, or
3. Check the JNDI Name for CMP or BMP resource bindings specified in the code by J2EE Application Developer. Open the deployed .ear folder in an assembly tool, and look for the JNDI Name for your entity beans under CMP or BMP resource bindings. Verify that the names match.

Deployed application that uses Java Native Interface (JNI) code does not start

An application that uses JNI code might not start after deployment. The JNI allows Java code running in a virtual machine to operate with applications and libraries written in other languages, such as C, C++, and assembly. If your J2EE application uses JNI in a 32-bit environment, you must recompile your code in a 64-bit environment. The JNI calls might be different after the compilation because JNI specifications can change from version to version. WebSphere Application Server now supports 64-bit environments.

For a ″pure″ Java application that does not use JNI code, 32-bit to 64-bit compatibility is not an issue.

Parsing error when running an application that uses the JSF configuration

If you are using double-byte characters in the profile name, you receive a parsing error when running an application that uses the JavaServer Faces™ (JSF) configuration. The problem is related to the JSF configuration that is part of the `jsf-ibm.jar`, which is included when building JSF applications in Rational® Application Development (RAD). The configuration files are referencing entities from inside the main `faces-config.xml` file.

Avoid using double-byte characters when you create a profile.

A Web resource does not display

If you are not able to display a resource in your browser, follow these steps:
1. Verify that your HTTP server is healthy by accessing the URL `http://server_name` from a browser and seeing whether the Welcome page appears. This action indicates whether the HTTP server is up and running, regardless of the state of WebSphere Application Server.
2. If the HTTP server Welcome page does not appear, that is, if you get a browser message like `page cannot be displayed` or something similar, try to diagnose your Web server problem.
3. If the HTTP server appears to function, the Application Server might not be serving the target resource. Try accessing the resource directly through the Application Server instead of through the HTTP server.

   If you cannot access the resource directly through the Application Server, Verify that the URL used to access the resource is correct.

   If the URL is incorrect and it is created as a link from another JSP file, servlet, or HTML file, try correcting it in the browser URL field and reloading, to confirm that the problem is a malformed URL. Correct the URL in the ″from″ HTML file, servlet or JSP file.

   If the URL appears to be correct, but you cannot access the resource directly through the Application Server, verify the health of the hosting Application Server and Web module:

a. View the hosting Application Server and Web module in the administrative console to verify that they are up and running.
b. Copy a simple HTML or JSP file (such as `SimpleJsp.jsp` in the WebSphere Application Server directory structure) to your Web module document root, and try to access it. If successful, the problem is with your resource.

   View the JVM log of your Application Server to find out why your resource cannot be found or served .
4. If you can access the resource directly through the Application Server, but not through an HTTP server, the problem lies with the HTTP plug-in -- the component that communicates between the HTTP server and the WebSphere Application Server.
5. If the JSP file and the servlet output are served, but not static resources such as `.html` and image files, see the steps for enabling file serving.
6. If some kinds of resources display correctly, but you cannot display a servlet by its class name:
   - Verify that the servlet is in a directory in the Web module class path, such as in the /*Web_module_name*.war/WEB-INF/classes directory.
   - Verify that you specify the full class name of the servlet, including its package name, in the URL.
   - Verify that `"/servlet"` precedes the class name in the URL. For example, if the root context of a Web module is `"myapp"`, and the servlet is `com.mycom.welcomeServlet`, then the URL reads:

     `http://hostname/myapp/servlet/com.mycom.welcomeServlet`
   - Verify that serving the servlets by class name is enabled for the hosting Web module by opening the source Web module in an assembly tool and browse the *serve servlets by classname* setting in the IBM Extensions property page. If necessary, enable this flag and redeploy the Web module.
   - For servlets or other resources served by mapped URLs, the URL is http://*hostname*/*Web module context root*/*mappedURL*.

If none of these steps fixes your problem, see if the problem has been identified and documented by looking at available online support (hints and tips, technotes, and fixes). If you do not find your problem listed there, see Troubleshooting help from IBM.

Diagnosing Web server problems

If you are unable to view the welcome page of your HTTP server, determine if the server is operating properly.

On Windows systems, look in the Services panel for the service corresponding to your HTTP server, and verify that the state is **Started**. If not, start it. If the service does not start, try starting it manually from the command prompt. If you are using IBM HTTP Server, the command is *IHS_install_dir*\apache .

On UNIX systems, execute the ps -ef | grep httpd command. There should be several processes running with a name of `"httpd"`. If not, start your HTTP server manually. If you are using IBM HTTP Server, the command is *IHS_install_dir*/bin/apachectl start.

If the HTTP server does not start:
- Examine the HTTP server error log for clues.
- Try restoring the HTTP server to its configuration prior to installing WebSphere Application Server and restarting it. If you are using IBM HTTP Server:
  – Rename the file *IHS_install_dir*\httpd.conf.
  – Copy the httpd.conf.default file to the httpd.conf directory.
  – If Apache is running, stop and restart it.
- For the Sun ONE (iPlanet) Web server, restore the obj.conf configuration file for Sun ONE V4.1 and both obj.conf and magnus.conf files for Sun ONE V6.0 and later.
- For the Microsoft Internet Information Server (IIS), remove the WebSphere Application Server plug-in through the IIS administrative GUI.

If restoring the HTTP server default configuration file works, manually review the configuration file that has WebSphere Application Server updates to verify directory and file names for WebSphere Application

Server files. If you cannot manually correct the configuration, you can uninstall and reinstall WebSphere Application Server to create a clean HTTP configuration file.

If restoring the default configuration file does not help, contact technical support for the Web server you are using. If you are using IBM HTTP Server with WebSphere Application Server, check available online support (hints and tips, technotes, and fixes). If you do not find your problem listed there, see Troubleshooting help from IBM

Accessing a Web resource through the application server and bypassing the HTTP server

Starting with WebSphere Application Server Version 4.0, you can bypass the HTTP server and access a Web resource through the application server. It is not recommended to serve a production Web site in this way, but it provides a good diagnostic tool when it is not clear whether a problem resides in the HTTP server, WebSphere Application Server, or the HTTP plug-in.

To access a Web resource through the Application Server:
1. Determine the port of the HTTP service in the target application server.
   a. In the WebSphere administrative console, click **Servers>Manage Application Servers**.
   b. Select the target server, then under Additional Properties click **Web Container**.
   c. Under the Additional Properties of the Web container, click **HTTP Transports**. You see the ports listed for virtual hosts served by the application server.
   d. There can be more than one port listed. In the default application server (server1), for example, 9060 is the port reserved for administrative requests, 9443 and 9043 are used for SSL-encrypted requests. To test the sample ″snoop″ servlet, for example, use the default application port 9080, unless it changes.
2. Use the HTTP transport port number of the application server to access the resource from a browser. For example, if the port is 9080, the URL is http://*hostname*:9080/myAppContext/myJSP.jsp.
3. If you are still unable to access the resource, verify that the HTTP transport port is in the ″Host Alias″ list:
   a. Click **Application Servers > Your_ApplicationServer > Web Container > HTTP Transports** to check the Default virtual host and the HTTP transport ports used by this application server.
   b. Click **Environment > Manage Virtual Hosts > default host > Host Aliases** to check if the HTTP transport port exists. Add an entry if necessary. For example, if the HTTP port for your application is server is 9080, add a host alias of *:9082.

Cannot uninstall an application or remove a node or application server

What kind of problem are you having?
- After uninstalling an application through wsadmin tool, the application continues to run and throws ″DocumentIOException″
- The removeNode command does not remove the installed application from the deployment manager
- I cannot display the syntax for the removeNode command.

If none of these steps fixes your problem:
- Make sure that the application and its Web and EJB modules are in a stopped state before uninstalling.
- If you are uninstalling or installing an application using **wsadmin**, make sure that you are using the **-conntype NONE** option to invoke **wsadmin** and enable local mode. To use the **-conntype NONE** option, stop the hosting application server before uninstalling the application.
- 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

After uninstalling application through the wsadmin tool, the application throws "DocumentIOException"

If this exception occurs after the application was uninstalled using wsadmin with the `-conntype NONE` option:
- Restart the server or,
- Rerun the uninstall command without the `-conntype NONE` option.

The removeNode command does not remove the installed application from the deployment manager

If the applications were installed indirectly using the **addNode** program with the **-includeapps** option, then `removeNode` will not uninstall them, since they may be in use by other nodes. These applications must be explicitly uninstalled, for example through the administrative console.

I cannot display the syntax for the removeNode command

Unlike the `addNode` command, the `removeNode` command is valid with no parameters, so executing it will execute the operation, that is, remove the node, without displaying the command syntax.

To see the valid options for removeNode, execute `removeNode -?` or `removeNode -help`.

Chapter 11. Add logging and tracing to your application

You can add logging and tracing to applications to help analyze performance and diagnose problems in WebSphere Application Server.

Deprecation: The JRas framework that is described in this information center is deprecated. However, you can achieve the same results using Java logging.

Designers and developers of applications that run with or under WebSphere Application Server, such as servlets, JavaServer Pages (JSP) files , enterprise beans, client applications, and their supporting classes, might find it useful to use Java logging for generating their application logging.

This approach has advantages over adding `System.out.println` statements to your code:
* Your messages are displayed in the WebSphere Application Server standard log files, using a standard message format with additional data, such as a date and time stamp that are added automatically.
* You can more easily correlate problems and events in your own application to problems and events that are associated with WebSphere Application Server components.
* You can take advantage of the WebSphere Application Server log file management features.
* You can view your messages with the Log and Trace Analyzer tool.
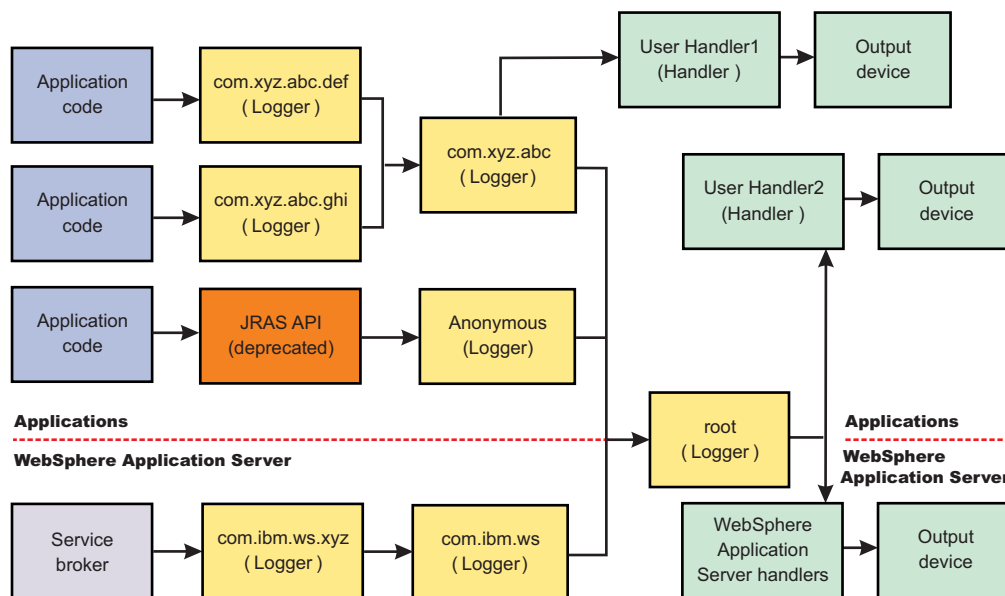
1. To use Java logging, configure properties using the administrative console.
2. Customize the properties to meet your logging needs. For example, enable or disable a particular log, specify the number of logs to be kept, and specify a format for log output.
3. Restart the application server after making static configuration changes.

Log and trace with Java logging

Java logging is the logging toolkit that is provided by the java.util.logging package. Java logging provides a standard logging API for your applications.

The application server redirects the system streams at the server startup. There is no way to allow the application to output logging to the console because the system streams can not be obtained by the application. If you would like to use console to monitor the application without using the console handler, you can either monitor the `SystemOut.log` file, or monitor a file created by another file handler.

Note: The application server uses Java logging internally and therefore certain restrictions apply for using system streams with this logging API by applications. During server startup, the standard output and error streams are replaced with special streams that write to the logging infrastructure, in order to include the output of the system streams in the log files. Because of this, applications can not use `java.util.logging.ConsoleHandler`, or any handler writing to `System.err` or `System.out` streams, attached to the root logger. If the user does attache the handler to the root logger, an infinite loop is created within the logging infrastructure, leading to stack overflow and server crash.

If the use of a handler that writes to system streams is necessary, attach it to a non-root logger so that it does not publish log records to parent handlers. The data written to the system streams is then formatted and written to the corresponding system stream log file. To monitor what is being written system streams, the configured log files (`SystemOut.log` and `SystemErr.log` by default) can be monitored.

Developing, deploying and maintaining applications are complex tasks. When an application encounters an unexpected condition, it might not be able to complete a requested operation. You might want the application to inform the administrator that the operation failed and tell the administrator why the operation failed. This information enables the administrator to take the proper corrective action. Application developers might need to gather detailed information that relates to the path of a running application to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as *logging* and *tracing*.

Message logging (messages) and diagnostic trace (trace) are conceptually similar, but do have important differences. These differences are important for application developers to understand to use these tools properly. The following operational definitions of messages and trace are provided.

Message
A message entry is an informational record that is intended for end users, systems administrators, and support personnel to view. The text of the message must be clear, concise, and interpretable by an end user. Messages are typically localized and displayed in the national language of the end user. Although the destination and lifetime of messages might be configurable, enable some level of message logging in normal system operation. Use message logging judiciously because of performance considerations and the size of the message repository.

Trace A trace entry is an information record that is intended for service engineers or developers to use. As such, a trace record might be considerably more complex, verbose, and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries can be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but can be enabled as needed to gather diagnostic information.

- To use Java logging, see "Configuring logging properties using the administrative console" on page 1500
- See the Java documentation for the java.util.logging class for a full description of the syntax and the construction of logging methods.

Loggers

Loggers are used by applications and runtime components to capture message and trace events.

When situations occur that are significant either due to a change in state, for example when a server completes startup or because a potential problem is detected, such as a timeout waiting for a resource, a message is written to the logs. Trace events are logged in debugging scenarios, where a developer needs a clear view of what is occurring in each component to understand what might be going wrong. Logged events are often the only events available when a problem is first detected, and are used during both problem recovery and problem resolution.

Loggers are organized hierarchically. Each logger can have zero or more child loggers.

Loggers can be associated with a resource bundle. If specified, the resource bundle is used by the logger to localize messages that are logged to the logger. If the resource bundle is not specified, a logger uses the same resource bundle as its parent.

You can configure loggers with a level. If specified, the level is compared by the logger to incoming events. The events that are less severe than the level set for the logger are ignored by the logger. If the level is not specified, a logger takes on the level that is used by its parent. The default level for loggers is Level.INFO.

Loggers can have zero or more attached handlers. If supplied, all events that are logged to the logger are passed to the attached handlers. Handlers write events to output destinations such as log files or network sockets. When a logger finishes passing a logged event to all of the handlers that are attached to that logger, the logger passes the event to the handlers that are attached to the parents of the logger. This process stops if a parent logger is configured not to use its parent handlers. Handlers in WebSphere Application Server are attached to the root logger. Set the useParentHandlers logger property to `false` to prevent the logger from writing events to handlers that are higher in the hierarchy.

Loggers can have a filter. If supplied, the filter is invoked for each incoming event to tell the logger whether or not to ignore it.

Applications interact directly with loggers to log events. To obtain or create a logger, a call is made to the Logger.getLogger method with a name for the logger. Typically, the logger name is either the package qualified class name or the name of the package that the logger is used by. The hierarchical logger namespace is automatically created by using the dots in the logger name. For example, the com.ibm.websphere.ras logger has a com.ibm.websphere parent logger, which has a com.ibm parent. The parent at the top of the hierarchy is referred to as the *root logger*. This root logger is created during initialization. The root logger is the parent of the com logger.

Loggers are structured in a hierarchy. Every logger, except the root logger, has one parent. Each logger can also have 0 or more children. A logger inherits log handlers, resource bundle names, and event filtering settings from its parent in the hierarchy. The logger hierarchy is managed by the LogManager function.

Loggers create log records. A log record is the container object for the data of an event. This object is used by filters, handlers, and formatters in the logging infrastructure.

The logger provides several sets of methods for generating log messages. Some log methods take only a level and enough information to construct a message. Other, more complex logp (log precise) methods support the caller in passing class name and method name attributes, in addition to the level and message information. The logrb (log with resource bundle) methods add the capability of specifying a resource bundle as well as the level, message information, class name, and method name. Using methods such as severe, warning, fine, finer, and finest you can log a message at a particular level. For more information on logging and how to use it in your applications read "Using loggers in an application" on page 1463. For a complete list of methods, see the java.util.logging documentation at http://java.sun.com/j2se/.

Log handlers

Log handlers write log record objects to output devices like log files, sockets, and notification mechanisms.

Loggers can have zero or more attached handlers. All objects that are logged to the logger are passed to the attached handlers, if handlers are supplied.

You can configure handlers with a level. The handler compares the level that is specified in the logged object to the level that is specified for the handler. If the level of the logged object is less severe than the level set in the handler, the object is ignored by the handler. The default level for handlers is ALL.

Handlers can have a filter. If a filter is supplied, the filter is invoked for each incoming object to tell the handler whether or not to ignore it.

Handlers can have a formatter. If a formatter is supplied, the formatter controls how the logged objects are formatted. For example, the formatter can decide to first include the time stamp, followed by a string representation of the level, followed by the message that is included in the logged object. The handler writes this formatted representation to the output device. Read "java.util.logging custom formatters" on page 1473 for information on using a custom formatter in your applications.

Both loggers and handlers can have levels and filters, and a logged object must pass all of these elements to be output. For example, you can set the logger level to FINE, but if the handler level is set at WARNING, only WARNING level messages are displayed in the output for that handler. Conversely, if your log handler is set to output all messages (level=All), but the logger level is set to WARNING, the logger never sends messages lower than WARNING to the log handler.

WebSphere Application Server uses the following set of log handlers that are available to all loggers:
* Diagnostic trace
* Java Management Extensions (JMX) notification object
* Service log
* SystemErr
* SystemOut

For instructions on how to configure these log handlers, see "Configuring logging properties using the administrative console" on page 1500.

Log levels

Levels control which events are processed by Java logging. WebSphere Application Server controls the levels of all loggers in the system.

The level value is set from configuration data when the logger is created and can be changed at run time from the administrative console. If a level is not set in the configuration data, a level is obtained by proceeding up the hierarchy until a parent with a level value is found. You can also set a level for each handler to indicate which events are published to an output device. When you change the level for a logger in the administrative console, the change is propagated to the children of the logger.

Levels are cumulative; a logger can process logged objects at the level that is set for the logger, and at all levels above the set level. Valid levels are:

Level	Content / Significance
Off	No events are logged.
Fatal	Task cannot continue and component cannot function.
Severe	Task cannot continue, but component can still function
Warning	Potential error or impending error
Audit	Significant event affecting server state or resources
Info	General information outlining overall task progress
Config	Configuration change or status
Detail	General information detailing subtask progress
Fine	Trace information - General trace + method entry / exit / return values
Finer	Trace information - Detailed trace

Level	Content / Significance
Finest	Trace information - A more detailed trace - Includes all the detail that is needed to debug problems
All	All events are logged. If you create custom levels, All includes your custom levels, and can provide a more detailed trace than Finest.

For instructions on how to set logging levels, see "Configuring logging properties using the administrative console" on page 1500

Note: Trace information, which includes events at the Fine, Finer and Finest levels, can be written only to the trace log. Therefore, if you do not enable diagnostic trace, setting the log detail level to Fine, Finer, or Finest does not effect the logged data.

Log filters

Log filters help control more detailed logging settings that are not handled by usual log level settings.

A filter provides an optional, secondary control over what is logged, beyond the control that is provided by setting the level. Applications can apply a filter mechanism to control logging output through the logging APIs. An example of filter usage is to suppress all the events with a particular message key.

A filter is attached to a logger or log handler using the appropriate setFilter method. Read "java.util.logging custom filters" on page 1473 for information on implementing custom filters. For a complete list of filter methods, see the java.util.logging documentation at http://java.sun.com/j2se/

Log formatters

Log formatters format log messages so they can be used by various log handlers.

Handlers can be configured with a log formatter that knows how to format log records. The event, which is represented by the log record object, is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which writes the output to the output device.

The formatter is responsible for rendering the event for output. This formatter uses the resource bundle that is specified in the event to look up the message in the appropriate language.

Formatters are attached to handlers using the setFormatter method.

With WebSphere Application Server, you can configure the formatter to work with trace, the `SystemOut.log`, and the `SystemErr.log` log files:
- **Basic (Compatible)**: Preserves only basic trace information. With this option, you can minimize the amount of space taken by the trace output.
- **Advanced**: Preserves more specific trace information. You can see detailed trace information for troubleshooting and problem determination.
- **Log analyzer trace format**: Preserves trace information in the same format as produced by Showlog tool.

You can select a formatter for a handler using the administrative console panels. See Diagnostic trace service settings for details.

You can find the java.util.logging documentation at http://java.sun.com/j2se/.

Logging properties for an application

Use the Logger.properties file to set logger attributes for specific loggers.

The properties file is loaded the first time that the Logger.getLogger(*logger_name*) method is called within an application.

Important: The name of the Logger.properties file is case sensitive. Use a capital ″L″ in the file name.

When an application calls the Logger.getLogger method for the first time, all the available logger properties files are loaded. Applications can provide Logger.properties files in:
- the META-INF directory of the Java archive (JAR) file for the application
- directories included in the class path of an application module
- directories included in the application class path

The properties file contains two categories of parameters, logger control and logger data:
- Logger control information
  - Minimum localization level: The minimum LogRecord level for which localization is attempted
  - Group: The logical group that this component belongs to
  - Event factory: The Common Base Event template file to use with the event factory. The naming convention for this template is the fully qualified component name, with a file extension of `.event.xml`. For example, a template that applies to the com.ibm.compXYZ package is called `com.ibm.compXYZ.event.xml`.
- Logger data information
  - Product name
  - Organization name
  - Component name
  - Extensions and additional properties

Syntax of the Logger.properties file

Use the following syntax to set logger properties:

`<logger base name>.<property>=value`

where:

`logger base name` is the starting part of the logger name to which the property applies. All loggers with names starting with this string have the property applied.

`property` is one of the following properties:
- organization
- product
- component
- minimum_localization_level
- group
- eventfactory

Sample Logger.properties file

In the following sample, the com.ibm.xyz.MyEventFactory event factory is used by any loggers in the com.ibm.websphere.abc package or any sub packages that do not override this value in their configuration file.

`com.ibm.websphere.abc.eventfactory=com.ibm.xyz.MyEventFactory`

Group Logger.properties file

In the following example, the group is MyTraceGroup and the components are com.ibm.stuff and com.ibm.morestuff:

```
com.ibm.stuff.group=MyTraceGroup
com.ibm.morestuff.group=MyTraceGroup
```

Sample security policy for logging

Set up a security policy to allow your applications to modify logging and handler properties.

The sample security policy that follows grants access to the file system and runtime classes. Include this security policy, with the entry `permission java.util.logging.LoggingPermission "control"`, in the META-INF directory of your application if you want your applications to programmatically alter controlled properties of loggers and handlers. The META-INF file is located in the following locations for the different module types:

EJB projects	ejbModule/META-INF/MANIFEST.MF
Application client projects	appClientModule/META-INF/MANIFEST.MF
Dynamic Web projects	WebContent/META-INF/MANIFEST.MF
Connector projects	connectorModule/META-INF/MANIFEST.MF

Below is a sample security policy that grants permission to modify logging properties:

```
//////////////////////////////////////////////
//
// WebSphere Application Server Security Policy
//
//////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// Allow all access to the file system and runtime classes
////////////////////////////////////////////////////////////////////
grant codeBase "file:${application}" {
    permission java.util.logging.LoggingPermission "control";
};
```

Using loggers in an application

This topic describes how to use Java logging within an application.

To create an application using Java logging, perform the following steps:

1. Create the necessary handler, formatter, and filter classes if you need your own log files.
2. If localized messages are used by the application, create a resource bundle, as described in "Creating log resource bundles and message files" on page 1468.
3. In the application code, get a reference to a logger instance, as described in "Using a logger."
4. Insert the appropriate message and trace logging statements in the application, as described in "Using a logger."

Using a logger

You can use Java logging to log messages and add tracing.

Use WsLevel.DETAIL level and above for messages, and lower levels for trace. The WebSphere Application Server Extension API (the com.ibm.websphere.logging package) contains the WsLevel class.

For messages use:

```
WsLevel.FATAL
Level.SEVERE
Level.WARNING
WsLevel.AUDIT
Level.INFO
Level.CONFIG
WsLevel.DETAIL
```

For trace use:

```
Level.FINE
Level.FINER
Level.FINEST
```

1. Use the logp method instead of the log or the logrb method. The logp method accepts parameters for class name and method name. The log and logrb methods will generally try to infer this information, but the performance penalty is prohibitive.

2. Avoid using the logrb method. This method leads to inefficient caching of resource bundles and poor performance.

3. Use the isLoggable method to avoid creating data for a logging call that does not get logged. For example:

```
if (logger.isLoggable(Level.FINEST)) {
  String s = dumpComponentState(); // some expensive to compute method
  logger.logp(Level.FINEST, className, methodName, "componentX state
dump:\n{0}", s);
  }
```

The following sample applies to localized messages:

```
// note - generally avoid use of FINE, FINER, FINEST levels for messages to be consistent with
// WebSphere Application Server

String componentName = "com.ibm.websphere.componentX";
String resourceBundleName = "com.ibm.websphere.componentX.Messages";
Logger logger = Logger.getLogger(componentName, resourceBundleName);

// "Convenience" methods - not generally recommended due to lack of class
/ method names
//   - cannot specify message substitution parameters
//   - cannot  specify class and method names
if (logger.isLoggable(Level.SEVERE))
 logger.severe("MSG_KEY_01");

if (logger.isLoggable(Level.WARNING))
 logger.warning("MSG_KEY_01");

if (logger.isLoggable(Level.INFO))
 logger.info("MSG_KEY_01");

if (logger.isLoggable(Level.CONFIG))
 logger.config("MSG_KEY_01");


// log methods are not generally used due to lack of class and method
names
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(WsLevel.FATAL))
 logger.log(WsLevel.FATAL, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.log(Level.SEVERE, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.log(Level.WARNING, "MSG_KEY_01", "parameter 1");
```

```
if (logger.isLoggable(WsLevel.AUDIT))
 logger.log(WsLevel.AUDIT, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.log(Level.INFO, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.log(Level.CONFIG, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.log(WsLevel.DETAIL, "MSG_KEY_01", "parameter 1");


// logp methods are the way to log
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
if (logger.isLoggable(WsLevel.FATAL))
 logger.logp(WsLevel.FATAL, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.logp(Level.SEVERE, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.logp(Level.WARNING, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.logp(WsLevel.AUDIT, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.logp(Level.INFO, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.logp(Level.CONFIG, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.logp(WsLevel.DETAIL, className, methodName, "MSG_KEY_01",
"parameter 1");


// logrb methods are not generally used due to diminished performance
of switching resource bundles dynamically
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
String resourceBundleNameSpecial =
"com.ibm.websphere.componentX.MessagesSpecial";

if (logger.isLoggable(WsLevel.FATAL))
 logger.logrb(WsLevel.FATAL, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.logrb(Level.SEVERE, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.logrb(Level.WARNING, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");
```

```
if (logger.isLoggable(WsLevel.AUDIT))
 logger.logrb(WsLevel.AUDIT, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.logrb(Level.INFO, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.logrb(Level.CONFIG, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.logrb(WsLevel.DETAIL, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");
```

For trace, or content that is not localized, the following sample applies:

```
// note - generally avoid use of FATAL, SEVERE, WARNING, AUDIT,
// INFO, CONFIG, DETAIL levels for trace
// to be consistent with WebSphere Application Server

String componentName = "com.ibm.websphere.componentX";
Logger logger = Logger.getLogger(componentName);

// Entering / Exiting methods are used for non trivial methods
if (logger.isLoggable(Level.FINER))
 logger.entering(className, methodName);

if (logger.isLoggable(Level.FINER))
 logger.entering(className, methodName, "method param1");

if (logger.isLoggable(Level.FINER))
 logger.exiting(className, methodName);

if (logger.isLoggable(Level.FINER))
 logger.exiting(className, methodName, "method result");


// Throwing method is not generally used due to lack of message - use
logp with a throwable parameter instead
if (logger.isLoggable(Level.FINER))
 logger.throwing(className, methodName, throwable);


// Convenience methods are not generally used due to lack of class
/ method names
//    - cannot specify message substitution parameters
//    - cannot specify class and method names
if (logger.isLoggable(Level.FINE))
 logger.fine("This is my trace");

if (logger.isLoggable(Level.FINER))
 logger.finer("This is my trace");

if (logger.isLoggable(Level.FINEST))
 logger.finest("This is my trace");


// log methods are not generally used due to lack of class and
method names
//    - enable use of WebSphere Application Server-specific levels
//    - enable use of message substitution parameters
//    - cannot specify class and method names
if (logger.isLoggable(Level.FINE))
 logger.log(Level.FINE, "This is my trace", "parameter 1");
```

```
if (logger.isLoggable(Level.FINER))
 logger.log(Level.FINER, "This is my trace", "parameter 1");

if (logger.isLoggable(Level.FINEST))
 logger.log(Level.FINEST, "This is my trace", "parameter 1");


// logp methods are the recommended way to log
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
if (logger.isLoggable(Level.FINE))
 logger.logp(Level.FINE, className, methodName, "This is my trace",
"parameter 1");

if (logger.isLoggable(Level.FINER))
 logger.logp(Level.FINER, className, methodName, "This is my trace",
"parameter 1");

if (logger.isLoggable(Level.FINEST))
 logger.logp(Level.FINEST, className, methodName, "This is my trace",
"parameter 1");


// logrb methods are not applicable for trace logging because no localization
is involved
```

Logger hierarchy

WebSphere Application Server handlers are attached to the Java root logger, which is at the top of the logger hierarchy. As a result, any request from anywhere in the logger tree can be processed by WebSphere Application Server handlers.

With WebSphere Application Server, you can configure the system to do the following:

- Forward all application logging requests to the WebSphere Application Server handlers. This behavior is the default.
- Forward all application logging requests to your own custom handlers. Set the **useParentHandlers** option to `false` on one of your custom loggers, and then attach your handlers to that logger.
- Forward all application logging requests to both WebSphere Application Server handlers, and your custom handlers, but do not forward WebSphere Application Server logging requests to your custom handlers. Set the **useParentHandlers** option to `true` on one of your non-root custom loggers, and then attach your handlers to that logger.True is the default setting.
- Forward all WebSphere Application Server logging requests to both WebSphere Application Server handlers, and your custom handlers. WebSphere Application Server logging requests are always forwarded to WebSphere Application Server handlers. To forward WebSphere Application Server requests to your custom handlers, attach your custom handlers to the Java root logger, so that they are at the same level in the hierarchy as the WebSphere Application Server handlers.

The following example shows how these requirements can be met using the Java logging infrastructure.

Application code → com.xyz.abc.def (Logger)

Application code → com.xyz.abc.ghi (Logger)

Application code → JRAS API (deprecated) → Anonymous (Logger)

com.xyz.abc (Logger)

User Handler1 (Handler) → Output device

User Handler2 (Handler) → Output device

root (Logger)

Applications

WebSphere Application Server

Applications

WebSphere Application Server

Service broker → com.ibm.ws.xyz (Logger) → com.ibm.ws (Logger)

WebSphere Application Server handlers → Output device

Creating log resource bundles and message files

Every method that accepts messages localizes those messages. The mechanism for providing localized messages is the resource bundle support provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.ResourceBundle, java.util.ListResourceBundle and java.util.PropertyResourceBundle classes, as well as the java.text.MessageFormat class.

The PropertyResourceBundle class is the preferred mechanism to use.

You can forward messages that are written to the internal WebSphere Application Server logs to other processes for display. For example, messages that are displayed on the administrative console, which can be running in a different location than the server process, can be localized using the *late binding* process. Late binding means that WebSphere Application Server does not localize messages when they are logged, but defers localization to the process that displays the message.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. You must package the resource bundle separately from the application, and install it in a location where the viewing process can access it.

By default, the WebSphere Application Server runtime localizes all the messages when they are logged. This localization eliminates the need to pass a `.jar` file to the application, unless you need to localize in a different location. However, you can use the early binding technique to localize messages as they log. An application that uses early binding must localize the message before logging it. The application looks up the localized text in the resource bundle and formats the message. Use the early binding technique to package the application resource bundles with the application.

To create a resource bundle, perform the following steps.
1. Create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:
   - Each property in the file is terminated with a line-termination character.
   - If a line contains white space only, or if the first non-white space character of the line is the pound sign symbol (#) or exclamation mark (!), the line is ignored. The # and ! characters can therefore be used to put comments into the file.

- Each line in the file, unless it is a comment or consists of white space only, denotes a single property. A backslash (\) is treated as the line-continuation character.
- The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (=), colon (:), and white space ( ).
- The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but doing this process is not recommended, because escaping characters is error prone and confusing. Instead, use a valid separator character that does not display in any keys in the properties file.
- White space after the key and separator is ignored until the first non-white space character is encountered. All characters remaining before the line-termination character define the element.

See the Java documentation for the java.util.Properties class for a full description of the syntax and the construction of properties files.

2. Translate the file into localized versions of the file with language-specific file names. For example, a file named `DefaultMessages.properties` can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese.

3. When the translated resource bundles are available, put the bundle in a directory that is part of the application class path.

4. When a message logger is obtained from the log manager, configure it to use a particular resource bundle. Messages logged with the Logger API use this resource bundle when message localization is performed. At run time, the user locale setting determines the properties file from which to extract the message that is specified by a message key, ensuring that the message is delivered in the correct language.

5. If the message loggers msg method is called, a resource bundle name must be explicitly provided.

The application locates the resource bundle based on the file location relative to any directory in the class path. For instance, if the DefaultMessages.properties property resource bundle is located in the *baseDir*/*subDir1*/*subDir2*/resources directory and *baseDir* is in the class path, the name *subdir1*.*subdir2*.resources.DefaultMessage is passed to the message logger to identify the resource bundle.

Resource bundle logging:

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a properties resource bundle. This sample shows how to create such a properties file.

Resource bundle sample

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs are inserted. All the normal properties file conventions and rules apply to this file. In addition, the creator must be aware of other restrictions that are imposed on the values by the Java MessageFormat class. For example, apostrophes must be escaped or they cause a problem. Avoid the use of non-portable characters. WebSphere Application Server does not support the use of extended formatting conventions that the MessageFormat class supports, such as {1, date} or {0,number, integer}.

Assume that the base directory for the application that uses this resource bundle is `baseDir` and that this directory is in the class path. Assume that the properties file is stored in the subdirectory `baseDir` that is not in the class path (for example, `baseDir/subDir1/subDir2/resources`). To allow the messages file to resolve, the `subDir1.subDir2.resources.DefaultMessage` name is used to identify the property resource bundle and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`.

```
# Contents of the DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three parameter: parm1={0}, parm2 = {1}, parm3={2}
```

When the `DefaultMessages.properties` file is created, the file can be sent to a translation center where the localized versions are generated.

Changing the message IDs used in log files

You can change the default format for message IDs in server logs by setting the com.ibm.websphere.logging.messageId.version system property.

In new releases of WebSphere Application Server logging files will be formatted according to a standardized system, but the default runtime behavior is still configured to use the older format, In new releases of WebSphere Application Server the message IDs written to log files will be changed to ensure they do not conflict with other IBM products. The default runtime behavior is still configured to use the older message IDs, deprecated in Version 6.1.

The following is a sample of an entry in a trace.log file using a default message ID. Note that the message ID is PMON0001A

```
[1/26/05 10:17:12:529 EST] 0000000a PMIImpl      A   PMON0001A: PMI is enabled
```

A sample of the same entry using a new message ID follows. Note that the message ID is CWPMI0001A. All new WebSphere Application Server message IDs begin with 'CW'.

```
[1/26/05 10:17:12:529 EST] 0000000a PMIImpl      A   CWPMI0001A: PMI is enabled.
```

If you are using a logging tool that uses the new standardized format, you might want to change the default configuration settings to format the logging output appropriately. You will need to change the configuration for each JVM in the cell if you want the output formatting to be the same across application servers.

To configure logging files to be formatted according to the new system, use the following command in the wsadmin utility:

```
set jvmEntry [$AdminConfig list JavaVirtualMachine]
$AdminConfig create Property $jvmEntry {{name com.ibm.websphere.logging.messageId.version} {value 6} {required false}}
```

Note: You must restart the application server for the changes to take effect. Also, remember to do this for each JVM in the cell for consistent output formatting.

Message IDs written to log files will now be compliant with the new standard.

Converting log files to use IBM unique Message IDs:

The convertlog command creates a new log file with either new or old message IDs substituted in place of the message IDs in the source file.

Prior to Version 6.x, components were assigned message IDs that are not necessarily unique across IBM software products. In Version 6.0, a system property was provided to map the message IDs in output logs to a set of IBM unique message IDs (all WebSphere Application Server message IDs now start with CW) that do not conflict with other IBM software products. The default runtime behavior still uses the old message IDs.

To facilitate the migration of logging tools that are reliant on the old message IDs, the convertlog command is provided to convert the message IDs of log entries from the old standard to the new standard, or the new standard back to the old. By default, the software is configured to use the old message IDs when

logging, but you can change the default output with the com.ibm.websphere.logging.messageId.version system property. Read "Changing the message IDs used in log files" on page 1470 for more information.

Use the convertlog command to convert the log output:

```
convertlog <source file name> <destination file name> [options]
        options: -newMessageFormat convert message IDs to CCCCCnnnnS format
                    (cannot be used with -m5)
                 -oldMessageFormat convert message IDs to CCCCnnnnS format
                    (cannot be used with -m6)
```

After using the convertlog command you have a new file with message IDs in the chosen format.

convertlog command:

The convertlog command is used to convert the message IDs in log entries from the old standard to the new standard, or the new standard back to the old.

Previous versions of WebSphere Application Server used message IDs that are deprecated in WebSphere Application Server Version 6.1. To facilitate the migration of tools based on the old message IDs, the convertlog command is implemented to translate log files from one message ID standard to the other.

Use the convertlog command as follows:

```
convertlog <source file name> <destination file name> [options]
        options: -newMessageFormat convert message IDs to CCCCCnnnnS format
                    (cannot be used with -m5)
                 -oldMessageFormat convert message IDs to CCCCnnnnS format
                    (cannot be used with -m6)
```

MessageConverter class:

The com.ibm.websphere.logging.MessageConverter class provides a method to convert a message ID at the front of a String into either a new message ID or an old message ID. The direction of the conversion is controlled with the *conversionType* argument.

Use the MessageConverter class with log analysis tools to convert message IDs from earlier versions of WebSphere Application Server into the corresponding message IDs that are used in later releases, or to revert message IDs to an earlier format. See the article Message reference for list of message ID mappings.

Method

```
public static java.lang.String convert(java.lang.String in, short conversionType)
```

Parameters

Use the following parameters with the MessageConverter class:

Parameter Name	Description
in	The message to convert. The method assumes the message ID is the first part of the supplied message with no leading white space.
conversionType	CONVERSION_TYPE_WASV5_TO_WASV6
	CONVERSION_TYPE_WASV6_TO_WASV5

java.util.logging custom log handlers

There may be occasions when you want to propagate log records to your own log handlers rather than participate in integrated logging.

To use a stand-alone log handler, set the useParentHandlers flag to `false` in your application.

The mechanism for creating a customer handler is the Handler class support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with handlers, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.logging API.

The following sample shows a custom handler:

```
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

/**
 * MyCustomHandler outputs contents to a specified file
 */
public class MyCustomHandler extends Handler {

 FileOutputStream fileOutputStream;
 PrintWriter printWriter;

 public MyCustomHandler(String filename) {
  super();

  // check input parameter
  if (filename == null)
   filename = "mylogfile.txt";

  try {
   // initialize the file
   fileOutputStream = new FileOutputStream(filename);
   printWriter = new PrintWriter(fileOutputStream);
  }
  catch (Exception e) {
   // implement exception handling...
  }
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#publish(java.util.logging.LogRecord)
  */
 public void publish(LogRecord record) {
  // ensure that this log record should be logged by this Handler
  if (!isLoggable(record))
   return;

  // Output the formatted data to the file
  printWriter.println(getFormatter().format(record));
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#flush()
  */
 public void flush() {
  printWriter.flush();
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#close()
  */
```

```
  public void close() throws SecurityException {
   printWriter.close();
 }
}
```

java.util.logging custom filters

A custom filter provides optional, secondary control over what is logged, beyond the control that is provided by the level.

The mechanism for creating a customer filter is the Filter interface support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with filters, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation the for java.util.logging API.

The following example shows a custom filter:

```
import java.util.Vector;
import java.util.logging.Filter;
import java.util.logging.LogRecord;

/**
 * MyCustomFilter rejects any log records whose Level is not contained in the
 * configured list of Levels.
 */
public class MyCustomFilter implements Filter {

 private Vector acceptableLevels;

 public MyCustomFilter(Vector acceptableLevels) {
  super();
  this.acceptableLevels = acceptableLevels;
 }

 /* (non-API documentation)
  * @see java.util.logging.Filter#isLoggable(java.util.logging.LogRecord)
  */
 public boolean isLoggable(LogRecord record) {
  return (acceptableLevels.contains(record.getLevel()));
 }

}
```

java.util.logging custom formatters

A formatter formats events. Handlers are associated with one or more formatters.

The mechanism for creating a customer formatter is the Formatter class support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with formatters, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.logging API.

The following example shows a custom formatter:

```
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;

/**
 * MyCustomFormatter formats the LogRecord as follows:
 * date   level   localized message with parameters
 */
public class MyCustomFormatter extends Formatter {

 public MyCustomFormatter() {
  super();
```

```
  }

  public String format(LogRecord record) {

    // Create a StringBuffer to contain the formatted record
    // start with the date.
    StringBuffer sb = new StringBuffer();

    // Get the date from the LogRecord and add it to the buffer
    Date date = new Date(record.getMillis());
    sb.append(date.toString());
    sb.append(" ");

    // Get the level name and add it to the buffer
    sb.append(record.getLevel().getName());
    sb.append(" ");

    // Get the formatted message (includes localization
    // and substitution of paramters) and add it to the buffer
    sb.append(formatMessage(record));

    return sb.toString();
  }
}
```

Custom handlers, filters, and formatters

In some cases you might want to have your own custom log files. Adding custom handlers, filters, and formatters enables you to customize your logging environment beyond what can be achieved by the configuration of the default WebSphere Application Server logging infrastructure.

The following example demonstrates how to add a new handler to process requests to the com.myCompany subtree of loggers (see "Logger hierarchy" on page 1467). The main method in this sample gives an example of how to use the newly configured logger.

```
import java.util.Vector;
import java.util.logging.Filter;
import java.util.logging.Formatter;
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyCustomLogging {

  public MyCustomLogging() {
    super();
  }

  public static void initializeLogging() {

    // Get the logger that you want to attach a custom Handler to
    String defaultResourceBundleName = "com.myCompany.Messages";
    Logger logger = Logger.getLogger("com.myCompany", defaultResourceBundleName);

    // Set up a custom Handler (see MyCustomHandler example)
    Handler handler = new MyCustomHandler("MyOutputFile.log");

    // Set up a custom Filter (see MyCustomFilter example)
    Vector acceptableLevels = new Vector();
    acceptableLevels.add(Level.INFO);
    acceptableLevels.add(Level.SEVERE);
    Filter filter = new MyCustomFilter(acceptableLevels);

    // Set up a custom Formatter (see MyCustomFormatter example)
    Formatter formatter = new MyCustomFormatter();

    // Connect the filter and formatter to the handler
```

```
  handler.setFilter(filter);
  handler.setFormatter(formatter);

  // Connect the handler to the logger
  logger.addHandler(handler);

  // avoid sending events logged to com.myCompany showing up in WebSphere
  // Application Server logs
  logger.setUseParentHandlers(false);

 }

 public static void main(String[] args) {
  initializeLogging();

  Logger logger = Logger.getLogger("com.myCompany");

  logger.info("This is a test INFO message");
  logger.warning("This is a test WARNING message");
  logger.logp(Level.SEVERE, "MyCustomLogging", "main", "This is a test SEVERE message");
 }
}
```

When the above program is run, the output of the program is written to the MyOutputFile.log file. The
content of the log is in the expected log file, as controlled by the custom handler, and is formatted as
defined by the custom formatter. The warning message is filtered out, as specified by the configuration of
the custom filter. The output is as follows:

```
C:\>type MyOutputFile.log
Sat Sep 04 11:21:19 EDT 2004 INFO This is a test INFO message
Sat Sep 04 11:21:19 EDT 2004 SEVERE This is a test SEVERE message
```

Configuring applications to use Jakarta Commons Logging

Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging
systems. WebSphere Application Server supports Jakarta Commons Logging by providing a logger. The
support does not change interfaces defined by Jakarta Commons Logging.

The WebSphere Application Server logger is a thin wrapper for the WebSphere Application Server logging
facility. The logger name is *com.ibm.websphere.commons.logging.WsJDK14Logger*. The logger can handle
logging objects defined by either of the following:
- Java Logging found in Java Specification Request 47: Logging API Specification
- Common Base Event

A *logging object* is an object that holds logging entry information.

To better understand Jakarta Commons Logging, read Jakarta Commons and the specifications for Java
Logging and for Common Base Event. To better understand use of the WebSphere Application Server
logger, read "Jakarta Commons Logging" on page 1476.

WebSphere Application Server provides the Jakarta Commons Logging binary distribution in its `libraries`
directory. By default, the product uses the Jakarta Commons Logging LogFactory implementation and
JDK14Logger.

For an application to use the WebSphere Application Server logger, the application must provide its own
configuration for the logger. To configure an application to use the WebSphere Application Server logger,
complete the steps that follow.
1. Examine "Configurations for the WebSphere Application Server logger" on page 1478 and determine
   which configuration best suits your application.

2. Change your application configuration as needed to enable use of the WebSphere Application Server logger.

After the application starts, Jakarta Commons Logging routes the application's logging output to the WebSphere Application Server logger.

Jakarta Commons Logging

Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging systems. The logging interface enables application logging to be simple and independent of the logging system that the application uses. You can change the logging implementation for a deployed application without having to change the application logging code. However, the simplicity of the logging interface prevents the application from leveraging all the functionality of the logging systems.

This topic provides the following information about Jakarta Commons Logging in WebSphere Application Server:
- "Support for Jakarta Commons Logging"
- "Benefits of support for Jakarta Commons Logging"
- "Overview of the process for using Jakarta Commons Logging"
- "Classes used to obtain a logger factory and logger" on page 1477
- "Logger level configuration and mapping" on page 1478

Support for Jakarta Commons Logging

WebSphere Application Server supports Jakarta Commons Logging by providing a logger, a thin wrapper for the WebSphere Application Server logging facility. The logger can handle both Java Logging (JSR-47) and Common Base Event logging objects. A *logging object* is an object that holds logging entry information.

The WebSphere Application Server support for Jakarta Commons Logging does not change interfaces defined by Jakarta Commons Logging.

Benefits of support for Jakarta Commons Logging

The WebSphere Application Server support for Jakarta Commons Logging provides the following benefits:
- WebSphere Application Server is pre-configured to use Jakarta Commons Logging.

  All of the functionality of Jakarta Commons Logging is provided for any application or WebSphere Application Server component. Logging calls are routed by default to the underlying WebSphere Application Server logging facility.
- A logger that uses the WebSphere Application Server logging facility.

  Applications and components can pass both Java Logging and Common Base Event logging objects to the WebSphere Application Server logger without conversion to strings, providing applications with enhanced logging. Further, Jakarta Commons Logging Logger levels are integrated into WebSphere Application Server administrative facilities.

Overview of the process for using Jakarta Commons Logging

Logging with Jakarta Commons Logging consists of the steps that follow. "Configurations for the WebSphere Application Server logger" on page 1478 provides details on configuring your application to use the WebSphere Application Server logger.
1. Obtain an instance of a logger factory.

  To obtain a logger factory, use Jakarta Commons Logging code. You can configure the code to meet your needs. In WebSphere Application Server, Jakarta Commons Logging is configured by default to instantiate the Jakarta Commons Logging default logger factory. Applications or WebSphere Application

Server components can provide their own configuration if they use a different logger factory implementation. Applications can use more than one factory.

2. Obtain an instance of a logger.

   To obtain a logger, use code implemented by a logger factory. Configuration of the code is implementation specific.

   The WebSphere Application Server logger implements the methods defined in the logging interface. The logging methods take at least one argument, which can be any Java object. The WebSphere Application Server logger, the WsJDK14Logger logger described in "Classes used to obtain a logger factory and logger," handles the following objects passed into the following logging methods:

   CommonBaseEvent
   > Wrapped into CommonBaseEventLogRecord

   CommonBaseEventLogRecord
   > Passed without change

   LogRecord
   > Passed without change

   Other objects
   > Converted to String

   Applications or WebSphere Application Server components can provide their own configuration if they use an implementation of a logger that is not specific to WebSphere Application Server. An application must know what factory is being used in order to configure it.

3. Start your application. Jakarta Commons Logging routes the application's logging output to the designated logger

Classes used to obtain a logger factory and logger

Class name	Description
LogFactory	*LogFactory* is a Jakarta Commons Logging class that implements initialization logic. LogFactory is an abstract class that every logger factory implementation has to extend. It provides static methods for obtaining: • An instance of a factory class • Instances of a logger, using an instance of the factory class LogFactory provides methods for obtaining instances of loggers, although these methods delegate the logger instantiation and configuration to an instance of a logger factory class. Logger factories, once instantiated, are cached on a per context class loader basis. The instances in a cache can be released. This functionality is designed for platform container implementations rather than for applications.
LogFactoryImpl	*LogFactoryImpl* is a Jakarta Commons Logging concrete class that implements the default logger factory using methods in LogFactory. To use Java Logging, there must always be at least one instance of a logger factory class, even if the application has not explicitly obtained one. If the configuration does not name a logger factory class, LogFactoryImpl is used as the default.
Log	*Log* is a Jakarta Commons Logging interface for loggers. Commons logging loggers have to implement the Log interface. Because the goal of Jakarta Commons Logging is to wrapper any logging system, the Log interface defines a small set of common logging methods. In WebSphere Application Server, WsJDK14Logger implements the Log interface. Logger instantiation and configuration is specific to every logger factory. Logging in WebSphere Application Server uses the default logger factory provided in Jakarta Commons Logging, which keeps instantiated loggers in cache, on a per class loader context basis.

Class name	Description
WsJDK14Logger	*WsJDK14Logger* is a WebSphere Application Server class that provides a Jakarta Commons Logging logger by implementing the Log interface. The WsJDK14Logger logger differs from the Java Logging logger in that the WsJDK14Logger logger enables Java Logging or Common Base Event objects to be passed over without converting them into String objects. This prevents any information loss the conversion to String might cause as well as allows the logging output to be more descriptive and precise. In contrast, the Java Logginglogger that is provided in Jakarta Commons Logging converts objects passed into the logging calls to String objects before passing them over to the underlying Java Logging.

Logger level configuration and mapping

Because Jakarta Commons Logging loggers are thin wrappers for specific logging systems, the loggers do not have their own level, but use the level of the logger from the underlying logging system. Although the underlying system can provide methods for changing level, there are no methods for changing level defined on the Log interface, which all Jakarta Commons Logging logger must implement. WsJDK14Logger uses the level of its underlying Java Logging logger.

Following table shows, on the left, the mapping of Jakarta Commons Logging levels within WsJDK14Logger to levels in the WebSphere Application Server implementation of Java Logging. On the right, it shows the levels defined in Java Logging and the level mapping in the Jakarta Commons Logging JDK14Logger to the Java Logging levels.

WsJDK14Logger	Java Logging in WebSphere Application Server	Java Logging	JDK14Logger
Fatal	Fatal		
Error	Severe	Severe	Fatal, Error
Warning	Warning	Warning	Warning
	Audit		
Info	Info	Info	Info
	Config	Config	
	Detail		
Debug	Fine	Fine	Debug
	Finer	Finer	
Trace	Finest	Finest	Trace

The WsJDK14Logger level is synchronized with the underlying Java Logging logger level. WebSphere Application Server administration controls the WsJDK14Logger level.

Configurations for the WebSphere Application Server logger

This topic describes several ways to configure an application to use the WebSphere Application Server logger.

The type of configuration that best suits an application depends upon the following:

- Whether the Class loader order setting for the application is `Classes loaded with parent class loader first (Parent First)` or `Classes loaded with application class loader first (Parent Last)`. You can set the class loader delegation mode on a console Class loading and update detection page.
- Whether Jakarta Commons Logging is bundled with the application configuration
- Whether Jakarta Commons Logging is provided within the application

The following tables describe the conditions required to enable an application to use the WebSphere Application Server logger.

Class loader mode is `Parent First` and Jakarta Commons Logging is bundled with the application

Jakarta Commons Logging configuration	LogFactory instance	Log instance	Comments
The application provides the configuration by either of the following: • The properties file `commons-logging.properties` in the application classpath **is not read** by the LogFactory because the parent class loader finds the WebSphere properties file first. • The class name **is read** from the file `META-INF/services/ org.apache.commons .logging.LogFactory`	The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration, unless the configuration is provided in a META-INF file of the application or module.	The log used is either of the following: • The Log implementation specified in the WebSphere Application Server default configuration • An application-specific Log implementation if an application-specific LogFactory that instantiates a different Log implementation is used.	The **application parent class loader** is the first class loader to load the Jakarta Commons Logging code. The **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes. For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**.
Not provided by the application	The log factory used is the LogFactory implementation specified in the WebSphere default configuration.	The log used is the Log implementation specified in the WebSphere default configuration.	The Jakarta Commons Logging bundled with the application is not used.

Class loader mode is `Parent First` and Jakarta Commons Logging is not bundled with the application

Jakarta Commons Logging configuration	LogFactory instance	Log instance	Comments
The application provides the configuration by either of the following: • The properties file `commons-logging.properties` in the application classpath **is not read** by the LogFactory because the parent class loader finds the WebSphere Application Server properties file first. • The class name **is read** from the file `META-INF/services/ org.apache.commons .logging.LogFactory`	The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration, unless the configuration is provided in a META-INF file of the application or module.	The log used is either of the following: • The Log implementation specified in the WebSphere Application Server default configuration • An application-specific Log implementation if an application-specific LogFactory that instantiates a different Log implementation is used.	The **application parent class loader** is the first class loader to load the Jakarta Commons Logging code. The **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes. For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**.

Jakarta Commons Logging configuration	LogFactory instance	Log instance	Comments
Not provided by the application	The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration.	The log used is the Log implementation specified in the WebSphere Application Server default configuration.	Same as in the previous row

Class loader mode is `Parent Last` and Jakarta Commons Logging is bundled with the application

Jakarta Commons Logging configuration	LogFactory instance	Log instance	Comments

The application provides the configuration by either of the following:
 	
• The properties file `commons-logging.properties` in the application classpath **is read** by the LogFactory because the parent class loader finds the WebSphere Application Server properties file first.
 	
• The class name **is read** from the file
 	
`META-INF/services/`
`org.apache.commons`
`.logging.LogFactory`
	The log factory used is either of the following:
	
	• The default Jakarta Commons Logging LogFactory
	
	• The LogFactory specified in the application configuration
		The log used is the Log implementation specified in the application configuration.
		
		If the log factory used is the default Jakarta Commons Logging LogFactory, the Log implementation must be on the classpath of the application class loader.
			The **application class loader** is the first class loader to load the Jakarta Commons Logging code. The **application bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.
			
			For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **application class loader**.

Not provided by the application	The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration.	The log used is the Log implementation specified in the WebSphere Application Server default configuration.	

Class loader mode is `Parent Last` and Jakarta Commons Logging is not bundled with the application

Jakarta Commons Logging configuration	LogFactory instance	Log instance	Comments
The application provides the configuration by either of the following: • The properties file `commons-logging.properties` in the application classpath **is read** by the LogFactory because the parent class loader finds the WebSphere properties file first. • The class name **is read** from the file `META-INF/services/ org.apache.commons .logging.LogFactory`	The log factory used is either of the following: • The default Jakarta Commons Logging LogFactory • The LogFactory specified in the application configuration	The log used is the Log implementation specified in the application configuration. If the log factory used is the default Jakarta Commons Logging LogFactory, the Log implementation must be on the classpath of the application class loader.	There is no Jakarta Commons Logging code at the application class loader. Thus, the **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes. For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**.
Not provided by the application	The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration.	The log used is the Log implementation specified in the WebSphere Application Server default configuration.	

Programming with the JRas framework

Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server applications.

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The JRas extensions allow message logging and diagnostic trace to work with WebSphere Application Server applications. They are based on the stand-alone JRas logging toolkit.

1. Retrieve a reference to the JRas manager.
2. Retrieve message and trace loggers by using methods on the returned manager.
3. Call the appropriate methods on the returned message and trace loggers to create message and trace entries, as appropriate.

JRas logging toolkit

The JRas logging toolkit provides diagnostic information to help the administrator diagnose problems or tune application performance.

Deprecated: The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Developing, deploying, and maintaining applications are complex tasks. For example, when a running application encounters an unexpected condition, it might not be able to complete a requested operation. In such a case, you might want the application to inform the administrator that the operation failed and provide information. This action enables the administrator to take the proper corrective action. Those who

develop or maintain applications might need to gather detailed information relating to the path of a running application to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as *message logging* and *diagnostic trace*.

Message logging (messages) and diagnostic trace (trace) are conceptually quite similar, but do have important differences. It is important for application developers to understand these differences to use these tools properly. To start with, the following operational definitions of messages and trace are provided.

Message

A message entry is an informational record that is intended for end users, systems administrators and support personnel to view. The text of the message must be clear, concise, and interpretable. Messages are typically localized, meaning that they display in the national language of the end user. Although the destination and lifetime of messages might be configurable, some level of message logging is always enabled in normal system operation. Message logging must be used judiciously due to both performance considerations and the size of the message repository.

Trace A trace entry is an information record that is intended for service engineers or developers to use. This trace record might be considerably more complex, verbose, and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries can be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but might be enabled as needed to gather diagnostic information.

WebSphere Application Server provides a message logging and diagnostic trace API that applications can use. This API is based on the stand-alone JRas logging toolkit, which was developed by IBM. The stand-alone JRas logging toolkit is a collection of interfaces and classes that provide message logging and diagnostic trace primitives. These primitives are not tied to any particular product or platform. The stand-alone JRas logging toolkit provides a limited amount of support, which is typically referred to as *systems management support*, including log file configuration support based on property files.

As designed, the stand-alone JRas logging toolkit does not contain the support that is required for integration into the WebSphere Application Server run time or for use in a Java 2 Platform, Enterprise Edition (J2EE) environment. To overcome these limitations, WebSphere Application Server provides a set of extension classes to address these shortcomings. This collection of extension classes is referred to as the JRas extensions. The JRas extensions do not modify the interfaces that are introduced by the stand-alone JRas logging toolkit, but provide the appropriate implementation classes. The conceptual structure that is introduced by the stand-alone JRas logging toolkit is described in the following section. It is equally applicable to the JRas extensions.

JRas concepts

The section contains a basic overview of important concepts and constructs that are introduced by the stand-alone JRas logging toolkit. This information is not an exhaustive overview of the capabilities of this logging toolkit, nor is it intended as a detailed discussion of usage or programming paradigms. More detailed information, including code examples, is available in JRas extensions and its subtopics, including in the API documentation for the various interfaces and classes that make up the logging toolkit.

Event types

The stand-alone JRas logging toolkit defines a set of event types for messages and a set of event types for trace. Examples of message types include informational, warning, and error. Examples of trace types include entry, exit, and trace.

Event classes

The stand-alone JRas logging toolkit defines both message and trace event classes.

Loggers

A logger is the primary object with which the user code interacts. Two types of loggers are defined: message loggers and trace loggers. The set of methods on message loggers and trace loggers are different because they provide different functionality. Message loggers create message records only and trace loggers create trace records only. Both types of loggers contain masks that indicate which categories of events the logger processes and which to ignore. Although every JRas logger

is defined to contain both a message and trace mask, the message logger uses only the message mask and the trace logger uses the trace mask only. For example, by setting a message logger message mask to the appropriate state, it can be configured to process only error messages and ignore informational and warning messages. Changing the trace mask state of a message logger has no effect.

A logger contains one or more handlers to which it forwards events for further processing. When the user calls a method on the logger, the logger compares the event type that is specified by the caller to its current mask value. If the specified type passes the mask check, the logger creates an event object to capture the information relating to the event that passed to the logger method. This information can include information, such as the names of the class and method which logs the event, a message, and parameters to log, among others. When the logger creates the event object, it forwards the event to all handlers currently registered with the logger.

Methods that are used within the logging infrastructure do not make calls to the logger method. When an application uses an object that extends a thread class, implements the hashCode method, and makes a call to the logging infrastructure from that method, the result is a recursive loop.

Handlers

A handler provides an abstraction over an output device or event consumer. An example is a file handler, which knows how to write an event to a file. The handler also contains a mask that is used to further restrict the categories of events the handler processes. For example, a message logger might be configured to pass both warning and error events, but a handler attached to the message logger might be configured to pass error events only. Handlers also include formatters, which the handler invokes to format the data in the passed event before it is written to the output device.

Formatters

Handlers are configured with formatters, which know how to format events of certain types. A handler can contain multiple formatters, each of which knows how to format a specific class of event. The event object is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which then writes it to the output device.

JRas Extensions

JRas extensions is the collection of implementation classes that support JRas integration into the WebSphere Application Server environment.

JRas extensions

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The stand-alone JRas logging toolkit defines interfaces and provides a variety of concrete classes that implement these interfaces. Because the stand-alone JRas logging toolkit is developed as a general purpose toolkit, the implementation classes do not contain the configuration interfaces and methods that are necessary for use in the WebSphere Application Server product. In addition, many of the implementation classes are not written appropriately for use in a Java 2 Platform, Enterprise Edition (J2EE) environment. To overcome these shortcomings, WebSphere Application Server provides the appropriate implementation classes that support integration into the WebSphere Application Server environment. The collection of these implementation classes is referred to as the *JRas extensions*.

Usage model

You can use the JRas extensions in three distinct operational modes:

Integrated

In this mode, message and trace records are written only to logs that are defined and maintained

by the WebSphere Application Server run time. This mode is the default mode of operation and is equivalent to the WebSphere Application Server V4.0 mode of operation.

stand-alone

In this mode, message and trace records are written solely to stand-alone logs that are defined and maintained by the user. You control which categories of events are written to which logs, and the format in which entries are written. You are responsible for configuration and maintenance of the logs. Message and trace entries are not written to WebSphere Application Server runtime logs.

Combined

In this mode, message and trace records are written to both WebSphere Application Server runtime logs and to stand-alone logs that you must define, control, and maintain. You can use filtering controls to determine which categories of messages and trace are written to which logs.

The JRas extensions are specifically targeted to an integrated mode of operation. The integrated mode of operation can be appropriate for some usage scenarios, but many scenarios are not adequately addressed by these extensions. Many usage scenarios require a stand-alone or combined mode of operation instead. A set of user extension points are defined that support JRas extensions in either a stand-alone or combined mode of operations.

JRas extension classes

WebSphere Application Server provides a base set of implementation classes that are collectively referred to as the *JRas extensions*. Many of these classes provide the appropriate implementations of loggers, handlers, and formatters for use in a WebSphere Application Server environment.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The collection of JRas classes is targeted at an integrated mode of operation. If you choose to use the JRas extensions in either stand-alone or combined mode, you can reuse the logger and manager class that are provided by the extensions, but you must provide your own implementations of handlers and formatters.

WebSphere Application Server message and trace loggers

The message and trace loggers that are provided by the stand-alone JRas logging toolkit cannot be directly used in the WebSphere Application Server environment. The JRas extensions provide the appropriate logger implementation classes. Instances of these message and trace logger classes are obtained directly and exclusively from the WebSphere Application Server Manager class. You cannot directly instantiate message and trace loggers. Obtaining loggers in any manner other than directly from the Manager class is not allowed and directly violates the programming model.

The message and trace logger instances that are obtained from the WebSphere Application Server Manager class are subclasses of the RASMessageLogger and RASTraceLogger classes that are provided by the stand-alone JRas logging toolkit. The RASMessageLogger and RASTraceLogger classes define the set of methods that are directly available. Public methods that are introduced by the JRas extensions logger subclasses cannot be called directly by user code because it is a violation of the programming model.

Loggers are named objects and are identified by name. When the Manager class is called to obtain a logger, the caller is required to specify a name for the logger. The Manager class maintains a name-to-logger instance mapping. Only one instance of a named logger is ever created within the lifetime of a process. The first call to the Manager class with a particular name results in the logger, which is configured by the Manager class. The Manager class caches a reference to the instance, then returns it to the caller. Subsequent calls to the Manager class that specify the same name result in a returned reference to the cached logger. Separate namespaces are maintained for message and trace loggers. You can use a single name obtain both a message logger and a trace logger from the Manager, without ambiguity, and without causing a namespace collision.

In general, loggers have no predefined granularity or scope. A single logger can be used to instrument an entire application. You might determine that having a logger per class is more effective, or the appropriate granularity might be somewhere in between. Partitioning an application into logging domains is determined by the application writer.

The WebSphere Application Server logger classes that are obtained from the Manager class are thread-safe. Although the loggers provided as part of the stand-alone JRas logging toolkit implement the serializable interface, loggers are not serializable. Loggers are stateful objects, tied to a Java virtual machine instance and are not serializable. Attempting to serialize a logger is a violation of the programming model.

Personal or individual logger subclasses are not supported in a WebSphere Application Server environment.

WebSphere Application Server handlers

WebSphere Application Server provides the appropriate handler class that is used to write message and trace events to the WebSphere Application Server run time logs. You cannot configure the WebSphere Application Server handler to write to any other destination. The creation of a WebSphere Application Server handler is a restricted operation and is not available to user code. Every logger that is obtained from the Manager comes preconfigured with an instance of this handler already installed. You can remove the WebSphere Application Server handler from a logger when you want to run in stand-alone mode. When you remove it, you cannot add the WebSphere Application Server handler again to the logger from which it is removed or any other logger. Also, you cannot directly call any method on the WebSphere Application Server handler. Attempting to create an instance of the WebSphere Application Server handler, to call methods on the WebSphere Application Server handler or to add a WebSphere Application Server handler to a logger by user code is a violation of the programming model.

WebSphere Application Server formatters

The WebSphere Application Server handler comes preconfigured with the appropriate formatter for data that is written to WebSphere Application Server logs. The creation of a WebSphere Application Server formatter is a restricted operation and not available to user code. No mechanism exists that allows the user to obtain a reference to a formatter installed in a WebSphere Application Server handler, or to change the formatter a WebSphere Application Server handler is configured to use.

WebSphere Application Server manager

WebSphere Application Server provides a Manager class in the com.ibm.websphere.ras package. All message and trace loggers must be obtained from this Manager class. A reference to the Manager class is obtained by calling the static Manager.getManager method. Message loggers are obtained by calling the createRASMessageLogger method on the Manager class. Trace loggers are obtained by calling the createRASTraceLogger method on the Manager class.

The manager also supports a *group* abstraction that is useful when dealing with trace loggers. The group abstraction supports multiple, unrelated trace loggers to register as part of a named entity called a *group*. WebSphere Application Server provides the appropriate systems management facilities to manipulate the trace setting of a group, similar to the way the trace settings of an individual trace logger work.

For example, suppose component A consists of 10 classes. Suppose each class is configured to use a separate trace logger. All 10 trace loggers in the component are registered as members of the same group, for example, Component_A_Group. You can turn on trace for a single class, or you can turn on trace for all 10 classes in a single operation using the group name, if you want a component trace. Group names are maintained within the namespace for trace loggers.

JRas framework (deprecated)

Because the JRas extensions classes do not provide the flexibility and behavior that are required for many scenarios, a variety of extension points are defined. You can write your own implementation classes to obtain the required behavior.

Deprecated: The JRas framework described in this topic is deprecated. However, you can achieve similar results using Java logging.

In general, the JRas extensions require you to call the Manager class to obtain a message logger or trace logger. No provision is made for you to provide your own message or trace logger subclasses. In general, user-provided extensions cannot be used to affect the integrated mode of operation. The behavior of the integrated mode of operation is solely determined by the WebSphere Application Server run time and the JRas extensions classes.

Handlers

The stand-alone JRas logging toolkit defines the RASIHandler interface. All handlers must implement this interface. You can write your own handler classes that implement the RASIHandler interface. Directly create instances of user-defined handlers and add them to the loggers that are obtained from the Manager class.

The stand-alone JRas logging toolkit provides several handler implementation classes. These handler classes are inappropriate for use in the Java 2 Platform, Enterprise Edition (J2EE) environment. You cannot directly use or subclass any of the Handler classes that are provided by the stand-alone JRas logging toolkit. Doing so is a violation of the programming model.

Formatters

The stand-alone JRas logging toolkit defines the RASIFormatter interface. All formatters must implement this interface. You can write your own formatter classes that implement the RASIFormatter interface. You can add these classes to a user-defined handler only. WebSphere Application Server handlers cannot be configured to use user-defined formatters. Instead, directly create instances of your formatters and add them to the your handlers appropriately.

As with handlers, the stand-alone JRas logging toolkit provides several formatter implementation classes. Direct use of these formatter classes is not supported.

Message event types

The stand-alone JRas toolkit defines message event types in the RASIMessageEvent interface. In addition, the WebSphere Application Server reserves a range of message event types for future use. The RASIMessageEvent interface defines three types, with values of $0x01$, $0x02$, and $0x04$. The values $0x08$ through $0x8000$ are reserved for future use. You can provide your own message event types by extending this interface appropriately. User-defined message types must have a value of $0x1000$ or greater.

Message loggers that are retrieved from the Manager class have their message masks set to `pass` or process all message event types defined in the RASIMessageEvent interface. To process user-defined message types, you must manually set the message logger mask to the appropriate state by user code after the message logger is obtained from the Manager class. WebSphere Application Server does not provide any built-in systems management support for managing message types.

Message event objects

The stand-alone JRas toolkit provides a RASMessageEvent implementation class. When a message logging method is called on the message logger, and the message type is currently enabled, the logger creates and distributes an event of this class to all handlers that are currently registered with that logger.

You can provide your own message event classes, but they must implement the RASIEvent interface. You must directly create instances of such user-defined message event classes. When it is created, pass your message event to the message logger by calling the message logger's fireRASEvent method directly. WebSphere Application Server message loggers cannot directly create instances of user-defined types in response to calling a logging method (`msg.message`) on the logger. In addition, instances of user-defined message types are never processed by the WebSphere Application Server handler. You cannot create instances of the RASMessageEvent class directly.

Trace event types

The stand-alone JRas toolkit defines trace event types in the RASITraceEvent interface. You can provide your own trace event types by extending this interface appropriately. In such a case, you must ensure that the values for the user-defined trace event types do not collide with the values of the types that are defined in the RASITraceEvent interface.

Trace loggers that are retrieved from the Manager class typically have their trace masks set to reject all types. A different starting state can be specified by using WebSphere Application Server systems management facilities. In addition, you can change the state of the trace mask for a logger at run-time, using WebSphere Application Server systems management facilities.

To process user-defined trace types, the trace logger mask must be manually set to the appropriate state by user code. WebSphere Application Server systems management facilities cannot be used to manage user-defined trace types, either at start time or run time.

Trace event objects

The stand-alone JRas toolkit provides a RASTraceEvent implementation class. When a trace logging method is called on the WebSphere Application Server trace logger and the type is currently enabled, the logger creates and distributes an event of this class to all the handlers that are currently registered with that logger.

You can provide your own trace event classes. Such trace event classes must implement the RASIEvent interface. You must create instances of such user-defined event classes directly. When it is created, pass the trace event to the trace logger by calling the trace logger's fireRASEvent method directly. WebSphere Application Server trace loggers cannot directly create instances of user-defined types in response to calling a trace method (`entry`, `exit`, `trace`) on the trace logger. In addition, instances of user-defined trace types are never processed by the WebSphere Application Server handler. You cannot create instances of the RASTraceEvent class directly.

User defined types, user defined events and WebSphere Application Server

By definition, the WebSphere Application Server handler processed user-defined message or trace types, or user-defined message or trace event classes. Message and trace entries of either a user-defined type or user-defined event class cannot be written to the WebSphere Application Server run-time logs.

JRas programming interfaces for logging (deprecated):

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

General considerations

You can configure the WebSphere Application Server to use Java 2 security to restrict access to protected resources such as the file system and sockets. Because user-written extensions typically access such protected resources, user-written extensions must contain the appropriate security checking calls, using AccessController doPrivileged calls. In addition, the user-written extensions must contain the appropriate

policy file. In general, locating user-written extensions in a separate package is a good practice. It is your responsibility to restrict access to the user-written extensions appropriately.

Writing a handler

User-written handlers must implement the RASIHandler interface. The RASIHandler interface extends the RASIMaskChangeGenerator interface, which extends the RASIObject interface. A short discussion of the methods that are introduced by each of these interfaces follows, along with implementation pointers. For more in-depth information on any of the particular interfaces or methods, see the corresponding product API documentation.

RASIObject interface

The RASIObject interface is the base interface for stand-alone JRas logging toolkit classes that are stateful or configurable, such as loggers, handlers, and formatters.
* The stand-alone JRas logging tookit supports rudimentary properties-file based configuration. To implement this configuration support, the configuration state is stored as a set of key-value pairs in a properties file. The public Hashtable getConfig and public void setConfig(Hashtable ht) methods are used to get and set the configuration state. The JRas extensions do not support properties-based configuration. Implement these methods as no-operations. You can implement your own properties-based configuration using these methods.
* Loggers, handlers, and formatters can be named objects. For example, the JRas extensions require the user to provide a name for the loggers that are retrieved from the manager. You can name your handlers. The public String getName and public void setName(String name) methods are provided to get or set the name field. The JRas extensions currently do not call these methods on user handlers. You can implement these methods as you want, including as no operations.
* Loggers, handlers, and formatters can also contain a description field. The public String getDescription and public void setDescription(String desc) methods can be used to get or set the description field. The JRas extensions currently do not use the description field. You can implement these methods as you want, including as no operations.
* The public String getGroup method is provided for use by the RASManager interface. Since the JRas extensions provide their own Manager class, this method is never called. Implement this as a no-operation.

RASIMaskChangeGenerator interface

The RASIMaskChangeGenerator interface is the interface that defines the implementation methods for filtering of events based on a mask state. It is currently implemented by both loggers and handlers. By definition, an object that implements this interface contains both a message mask and a trace mask, although both need not be used. For example, message loggers contain a trace mask, but the trace mask is never used because the message logger never generates trace events. Handlers, however, can actively use both mask values. For example, a single handler can handle both message and trace events.
* The public long getMessageMask and public void setMessageMask(long mask) methods are used to get or set the value of the message mask. The public long getTraceMask and public void setTraceMask(long mask) methods are used to get or set the value of the trace mask.

In addition, this interface introduces the concept of *calling back* to interested parties when a mask changes state. The callback object must implement the RASIMaskChangeListener interface.
* The public void addMaskChangeListener(RASIMaskChangeListener listener) and public void removeMaskChangeListener(RASIMaskChangeListener listener) methods are used to add or remove listeners to the handler. The public Enumeration getMaskChangeListeners method returns an enumeration over the list of currently registered listeners. The public void fireMaskChangedEvent(RASMaskChangeEvent mc) method is used to call back all the registered listeners to inform them of a mask change event.

For efficiency reasons, the JRas extensions message and trace loggers implement the RASIMaskChangeListener interface. The logger implementations maintain a composite mask in addition to the logger mask. The logger composite mask is formed by logically *or'ing* the appropriate masks of all handlers that are registered to that logger, then *and'ing* the result with the logger mask. For example, the message logger composite mask is formed by or'ing the message masks of all handlers that are registered with that logger, then and'ing the result with the logger message mask.

All handlers are required to properly implement these methods. In addition, when a user handler is instantiated, the logger that is added must be registered with the handler; use the addMaskChangeListener method. When either the message mask or trace mask of the handler is changed, the logger must be called back to inform it of the mask change. With this process, the logger can dynamically maintain the composite mask.

The RASMaskChangedEvent class is defined by the stand-alone JRas logging toolkit. Direct use of that class by user code is supported in this context.

In addition, the RASIMaskChangeGenerator interface introduces the concept of caching the names of all message and trace event classes that the implementing object process. The intent of these methods is to support a management program such as a graphical user interface to retrieve the list of names, introspect the classes to determine the event types that they might possibly process and display the results. The JRas extensions do not ever call these methods, so they can be implemented as no operations.
- The public void addMessageEventClass(String name) and public void removeMessageEventClass(String name) methodscan be called to add or remove a message event class name from the list. The method public Enumeration getMessageEventClasses returns an enumeration over the list of message event class names. Similarly, the public void addTraceEventClass(String name) and public void removeTraceEventClass(String name) methods can be called to add or remove a trace event class name from the list. The public Enumeration getTraceEventClasses method returns an enumeration over the list of trace event class names.

RASIHandler interface

The RASIHandler interface introduces the methods that are specific to the behavior of a handler.

The RASIHandler interface, as provided by the stand-alone JRas logging toolkit, supports handlers that run in either a synchronous or asynchronous mode. In asynchronous mode, events are typically queued by the calling thread and then written by a worker thread. Because spawning of threads is not supported in the WebSphere Application Server environment, it is expected that handlers do not queue or batch events, although this activity is not expressly prohibited.
- The public int getMaximumQueueSize() and public void setMaximumQueueSize(int size) methods create `IllegalStateException` exceptions to manage the maximum queue size. The public int getQueueSize method is provided to query the actual queue size.
- The public int getRetryInterval and public void setRetryInterval(int interval) methods support the notion of error retry, which implies some type of queueing.
- The public void addFormatter(RASIFormatter formatter), public void removeFormatter(RASIFormatter formatter) and public Enumeration getFormatters methods are provided to manage the list of formatters that the handler can be configured with. Different formatters can be provided for different event classes, if appropriate.
- The public void openDevice, public void closeDevice and public void stop methods are provided to manage the underlying device that the handler abstracts.
- The public void logEvent(RASIEvent event) and public void writeEvent(RASIEvent event) methods are provided to pass events to the handler for processing.

Writing a formatter

User-written formatters must implement the RASIFormatter interface. The RASIFormatter interface extends the RASIObject interface. The implementation of the RASIObject interface is the same for both handlers

and formatters. A short discussion of the methods that are introduced by the RASIFormatter interface follows. For more in-depth information on the methods introduced by this interface, see the corresponding product API documentation.

RASIFormatter interface
- The public void setDefault(boolean flag) and public boolean isDefault methods are used by the concrete RASHandler classes that are provided by the stand-alone JRas logging toolkit to determine if a particular formatter is the default formatter. Because these RASHandler classes must never be used in a WebSphere Application Server environment, the semantic significance of these methods can be determined by the user.
- The public void addEventClass(String name), public void removeEventClass(String name) and public Enumeration getEventClasses methods are provided to determine which event classes a formatter can use to format. You can provide the appropriate implementations.
- The public String format(RASIEvent event) method is called by handler objects and returns a formatted String representation of the event.

Programming model summary

The programming model that is described in this section builds upon and summarizes some of the concepts already introduced. This section also formalizes usage requirements and restrictions. Use of the WebSphere Application Server JRas extensions in a manner that does not conform to the following programming guidelines is prohibited.

Deprecated: The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

You can use the WebSphere Application Server JRas extensions in three distinct operational modes. The programming models concepts and restrictions apply equally across all modes of operation.
- You must not use implementation classes that are provided by the stand-alone JRas logging toolkit directly, unless specifically noted otherwise. Direct usage of those classes is not supported. IBM Support provides no diagnostic aid or bug fixes relating to the direct use of classes that are provided by the stand-alone JRas logging toolkit.
- You must obtain message and trace loggers directly from the Manager class. You cannot directly instantiate loggers.
- You cannot replace the WebSphere Application Server message and trace logger classes.
- You must guarantee that the logger names that are passed to the Manager class are unique, and follow the documented naming constraints. When a logger is obtained from the Manager class, you must not attempt to change the name of the logger by calling the setName method.
- Named loggers can be used more than once. For any given name, the first call to the Manager class results in the Manager class creating a logger that is associated with that name. Subsequent calls to the Manager class that specify the same name result in a returned reference to the existing logger.
- The Manager class maintains a hierarchical namespace for loggers. Use a dot-separated, fully qualified class name to identify any logger. Other than dots or periods, logger names cannot contain any punctuation characters, such as an asterisk (*), a comma (.), an equals sign (=), a colon (:), or quotes.
- Group names must comply with the same naming restrictions as logger names.
- The loggers returned from the Manager class are subclasses of the RASMessageLogger and the RASTraceLogger classes that are provided by the stand-alone JRas logging toolkit. You can call any public method that is defined by the RASMessageLogger and RASTraceLogger classes. You cannot call any public method that is introduced by the provided subclasses.
- If you want to operate in either stand-alone or combined mode, you must provide your own Handler and Formatter subclasses. You cannot use the Handler and Formatter classes that are provided by the stand-alone JRas logging toolkit. User written handlers and formatters must conform to the documented guidelines.
- Loggers that are obtained from the Manager class come with a WebSphere Application Server handler installed. This handler writes message and trace records to logs that are defined by the WebSphere Application Server run time. Manage these logs using the provided systems management interfaces.

- You can programmatically add and remove user-defined handlers from a logger at any time. Multiple additions and removals of user defined handlers are supported. You are responsible for creating an instance of the handler to add, configuring the handler by setting the handler mask value and formatter appropriately, then adding the handler to the logger using the addHandler method. You are responsible for programmatically updating the masks of user-defined handlers, as appropriate.
- You might get a reference to the handler that is installed within a logger by calling the getHandlers method on the logger and processing the results. You must not call any methods on the handler that are obtained in this way. You can remove the WebSphere Application Server handler from the logger by calling the logger removeHandler method, passing in the reference to the WebSphere Application Server handler. When removed, the WebSphere Application Server handler cannot be added again to the logger.
- You can define your own message type. The behavior of user-defined message types and restrictions on their definitions is discussed in Extending the JRas framework.
- You can define your own message event classes. The use of user-defined message event classes is discussed in Extending the JRas framework.
- You can define your own trace types. The behavior of user-defined trace types and restrictions on your definitions is discussed in Extending the JRas framework.
- You can define your own trace event classes. The use of user-defined trace event classes is discussed in Extending the JRas framework.
- You must programmatically maintain the bits in the message and trace logger masks that correspond to any user-defined types. If WebSphere Application Server facilities are used to manage the predefined types, these updates must not modify the state of any of the bits that correspond to those types. If you are assuming ownership responsibility for the predefined types, then you can change all bits of the masks.

JRas messages and trace event types

This topic describes JRas message and trace event types.

Event types

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The base message and trace event types that are defined by the stand-alone JRas logging toolkit are not the same as the native types that are recognized by the WebSphere Application Server run-time. Instead, the basic JRas types are mapped onto the native types. This mapping can vary by platform or edition. The mapping is discussed in the following section.

Platform message event types

The message event types that are recognized and processed by the WebSphere Application Server runtime are defined in the RASIMessageEvent interface that is provided by the stand-alone JRas logging toolkit. These message types are mapped onto the native message types, as follows.

WebSphere Application Server native type	JRas RASIMessageEvent type
Audit	TYPE_INFO, TYPE_INFORMATION
Warning	TYPE_WARN, TYPE_WARNING
Error	TYPE_ERR, TYPE_ERROR

Platform trace event types

The trace event types that are recognized and processed by the WebSphere Application Server run time are defined in the RASITraceEvent interface that is provided by the stand-alone JRas logging toolkit. The RASITraceEvent interface provides a rich and complex set of types. This interface defines both a simple set of levels, as well as a set of enumerated types.

- For a user who prefers a simple set of levels, the RASITraceEvent interface provides `TYPE_LEVEL1`, `TYPE_LEVEL2`, and `TYPE_LEVEL3`. The implementations provide support for this set of levels. The levels are hierarchical, enabling level 2 also enables level 1, enabling level 3 also enables levels 1 and 2.
- For users who prefer a more complex set of values that can be *OR'd* together, the RASITraceEvent interface provides `TYPE_API`, `TYPE_CALLBACK`, `TYPE_ENTRY_EXIT`, `TYPE_ERROR_EXC`, `TYPE_MISC_DATA`, `TYPE_OBJ_CREATE`, `TYPE_OBJ_DELETE`, `TYPE_PRIVATE`, `TYPE_PUBLIC`, `TYPE_STATIC`, and `TYPE_SVC`.

The trace event types are mapped onto the native trace types as follows:

Mapping WebSphere Application Server trace types to the JRas RASITraceEvent level types.

WebSphere Application Server native type	JRas RASITraceEvent level type
Event	TYPE_LEVEL1
EntryExit	TYPE_LEVEL2
Debug	TYPE_LEVEL3

Mapping WebSphere Application Server trace types to the JRas RASITraceEvent enumerated types.

WebSphere Application Server native type	JRas RASITraceEvent enumerated types
Event	TYPE_ERROR_EXC, TYPE_SVC, TYPE_OBJ_CREATE, TYPE_OBJ_DELETE
EntryExit	TYPE_ENTRY_EXIT, TYPE_API, TYPE_CALLBACK, TYPE_PRIVATE, TYPE_PUBLIC, TYPE_STATIC
Debug	TYPE_MISC_DATA

For simplicity, it is recommended that one or the other of the tracing type methodologies is used consistently throughout the application. If you decide to use the non-level types, choose one type from each category and use those types consistently throughout the application, to avoid confusion.

Message and trace parameters

The various message logging and trace method signatures accept the `Object`, `Object[]` and `Throwable` parameter types. WebSphere Application Server processes and formats the various parameter types as follows:

Primitives
> Primitives, such as int and long are not recognized as subclasses of Object type and cannot be directly passed to one of these methods. A primitive value must be transformed to a proper Object type (Integer, Long) before passing as a parameter.

Object
> The toString method is called on the object and the resulting String is displayed. Implement the toString method appropriately for any object that is passed to a message logging or trace method. It is the responsibility of the caller to guarantee that the toString method does not display confidential data such as passwords in clear text, and does not cause infinite recursion.

Object[]
> The Object[] type is provided for the case when more than one parameter is passed to a message logging or trace method. The toString method is called on each Object in the array. Nested arrays are not handled, that is none of the elements in the Object array belong in an array.

Throwable

The stack trace of the Throwable type is retrieved and displayed.

Array of primitives

An array of primitive, for example, `byte[]`, `int[]`, is recognized as an Object, but is treated somewhat as a second cousin of Object by Java code. In general, avoid arrays of primitives, if possible. If arrays of primitives are passed, the results are indeterminate and can change, depending on the type of array passed, the API used to pass the array, and the release of the product. For consistent results, user code needs to preprocess and format the primitive array into some type of String form before passing it to the method. If such preprocessing is not performed, the following problems can result:

- [B@924586a0b - This message is deciphered as a byte array at location X. This message is typically returned when an array is passed as a member of an Object[] type and results from calling the toString method on the byte[] type.
- Illegal trace argument : array of long. This response is typically returned when an array of primitives is passed to a method taking an Object.
- 01040703: The hex representation of an array of bytes. Typically this problem can occur when a byte array is passed to a method taking a single Object. This behavior is subject to change and cannot be relied on.
- ″1″ ″2″: The String representation of the members of an int[] type formed by converting each element to an integer and calling the toString method on the integers. This behavior is subject to change and cannot be relied on.
- [Ljava.lang.Object;@9136fa0b : An array of objects. Typically this response is seen when an array containing nested arrays is passed.

Controlling message logging

Writing a message to a WebSphere Application Server log requires that the message type passes three levels of filtering or screening:

1. The message event type must be one of the message event types that is defined in the RASIMessageEvent interface.
2. Logging of that message event type must be enabled by the state of the message logger mask.
3. The message event type must pass any filtering criteria that is established by the WebSphere Application Server run-time.

When a WebSphere Application Server logger is obtained from the Manager class, the initial setting of the mask forwards all native message event types to the WebSphere Application Server handler. It is possible to control what messages get logged by programmatically setting the state of the message logger mask.

Some editions of the product support user specified message filter levels for a server process. When such a filter level is set, only messages at the specified severity levels are written to WebSphere Application Server. Message types that pass the mask check of the message logger can be filtered out by WebSphere Application Server.

Control tracing

Each edition of the product provides a mechanism for enabling or disabling trace. The various editions can support static trace enablement (trace settings are specified before the server is started), dynamic trace enablement (trace settings for a running server process can be dynamically modified), or both.

Writing a trace record to a WebSphere Application Server requires that the trace type passes three levels of filtering or screening:

1. The trace event type must be one of the trace event types that is defined in the RASITraceEvent interface.
2. Logging of that trace event type must be enabled by the state of the trace logger mask.
3. The trace event type must pass any filtering criteria that is established by the WebSphere Application Server run-time.

When a logger is obtained from the Manager class, the initial setting of the mask is to suppress all trace types. The exception to this rule is the case where the WebSphere Application Server run time supports static trace enablement and a non-default startup trace state for that trace logger is specified. Unlike message loggers, the WebSphere Application Server can dynamically modify the trace mask state of a trace logger. WebSphere Application Server only modifies the portion of the trace logger mask that corresponds to the values that are defined in the RASITraceEvent interface. WebSphere Application Server does not modify undefined bits of the mask that might be in use for user-defined types.

When the dynamic trace enablement feature that is available on some platforms is used, the trace state change is reflected both in the application server run time and the trace mask of the trace logger. If user code programmatically changes the bits in the trace mask corresponding to the values that are defined by in the RASITraceEvent interface, the mask state of the trace logger and the run time state become unsynchronized and unexpected results occur. Therefore, programmatically changing the bits of the mask corresponding to the values that are defined in the RASITraceEvent interface is not supported.

Instrumenting an application with JRas extensions

You can create an application using JRas extensions.

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

To create an application using the WebSphere Application Server JRas extensions, perform the following steps:
1. Determine the mode for the extensions: integrated, stand-alone, or combined.
2. If the extensions are used in either stand-alone or combined mode, create the necessary handler and formatter classes.
3. If localized messages are used by the application, create a resource bundle.
4. In the application code, get a reference to the Manager class and create the manager and logger instances.
5. Insert the appropriate message and trace logging statements in the application.

Creating JRas resource bundles and message files

The WebSphere Application Server message logger provides the message and msg methods so the user can log localized messages. In addition, the message logger provides the textMessage method to log messages that are not localized. Applications can use either or both, as appropriate.

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The mechanism for providing localized messages is the resource bundle support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.ResourceBundle, java.util.ListResourceBundle and java.util.PropertyResourceBundle classes, as well as the java.text.MessageFormat class.

The PropertyResourceBundle class is the preferred mechanism to use. In addition, note that the JRas extensions do not support the extended formatting options such as {1, date} or {0, number, integer} that are provided by the MessageFormat class.

You can forward messages that are written to the internal WebSphere Application Server logs to other processes for display. For example, messages that are displayed on the administrative console, which can be running in a different location than the server process, can be localized using the *late binding* process. Late binding means that WebSphere Application Server does not localize messages when they are logged, but defers localization to the process that displays the message.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. You must package the resource bundle separately from the application, and install it in a location where the viewing process can access it. If you do not want to take these steps, you can use the early binding technique to localize messages as they are logged.

The two techniques are described as follows:

Early binding

The application must localize the message before logging it. The application looks up the localized text in the resource bundle and formats the message. When formatting is complete, the application logs the message using the textMessage method. Use this technique to package the application resource bundles with the application.

Late binding

The application can choose to have the WebSphere Application Server run time localize the message in the process where it displays. Using this technique, the resource bundles are packaged in a stand-alone `.jar` file, separately from the application. You must then install the resource bundle `.jar` file on every machine in the installation from which an administrative console or log viewing program might be run. You must install the `.jar` file in a directory that is part of the extensions class path. In addition, if you forward logs to IBM service, you must also forward the `.jar` file that contains the resource bundles.

To create a resource bundle, perform the following steps.

1. Create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:
   - Each property in the file is terminated with a line-termination character.
   - If a line contains only white space, or if the first non-white space character of the line is the number sign symbol (#) or exclamation mark (!), the line is ignored. The # and ! characters can therefore be used to put comments into the file.
   - Each line in the file, unless it is a comment or consists only of white space, denotes a single property. A backslash (\) is treated as the line-continuation character.
   - The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (=), colon (:), and white space ( ).
   - The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but using this approach is not recommended because escaping characters is error prone and confusing. Instead, use a valid separator character that does not display in any keys in the properties file.
   - White space after the key and separator is ignored until the first non-white space character is encountered. All characters that remain before the line-termination character define the element.

   See the Java documentation for the java.util.Properties class for a full description of the syntax and construction of properties files.

2. Translate the file into localized versions of the file with language-specific file names for example, the DefaultMessages.properties file can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese.

3. When the translated resource bundles are available, write them to a system-managed persistent storage medium. Resource bundles are used to convert the messages into the requested national language and locale.

4. When a message logger is obtained from the JRas manager, configure the logger to use a particular resource bundle. Messages logged through the `message` API use this resource bundle when message localization is performed. At run time, the user's locale setting is used to determine the properties file from which to extract the message that is specified by a message key, ensuring that the message is delivered in the correct language.

5. If the message loggers msg method is called, explicitly identify a resource bundle name.

The application locates the resource bundle based on the file location relative to any directory in the class path. For instance, if the DefaultMessages.properties property resource bundle is in the `baseDir/subDir1/subDir2/resources` directory and `baseDir` is in the class path, the name *subdir1.subdir2*.resources.DefaultMessage is passed to the message logger to identify the resource bundle.

JRas resource bundles:

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a PropertiesResourceBundle resource bundle. This sample shows how to create such a properties file.

Resource bundle sample

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs are inserted into it. All the normal properties files conventions and rules apply to this file. In addition, the creator must be aware of other restrictions that are imposed on the values by the Java MessageFormat class. For example, apostrophes must be escaped or they cause a problem. Avoid the use of non-portable characters. WebSphere Application Server does not support the use of extended formatting conventions that the MessageFormat class supports, such as {1, date} or {0, number, integer}.

Assume that the base directory for the application that uses this resource bundle is `baseDir` and that this directory is in the class path. Assume that the properties file is stored in the subdirectory `baseDir` that is not in the class path (*baseDir/subDir1/subDir2/resources*). To allow the messages file to resolve, the subDir1.subDir2.resources.DefaultMessage name is used to identify the PropertyResourceBundle resource bundle and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`:

```
# Contents of the DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three substitution parameters: parm1={0}, parm2 = {1}, parm3={2}
```

When the `DefaultMessages.properties` file is created, the file can be sent to a translation center where the localized versions are generated.

JRas manager and logger instances

You can use the JRas extensions in integrated, stand-alone, or combined mode. Configuration of the application varies depending on the mode of operation, but use of the loggers to log message or trace entries is identical in all modes of operation.

Deprecated: The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Integrated mode is the default mode of operation. In this mode, message and trace events are sent to the WebSphere Application Server logs.

In the combined mode, message and trace events are logged to both WebSphere Application Server and user-defined logs.

In the stand-alone mode, message and trace events are logged only to user-defined logs.

Using the message and trace loggers

Regardless of the mode of operation, the use of message and trace loggers is the same.

Using a message logger

The message logger is configured to use the DefaultMessages resource bundle. Message keys must be passed to the message loggers if the loggers are using the message API.

```
msgLogger.message(RASIMessageEvent.TYPE_WARNING, this,
    methodName, "MSG_KEY_00");
... msgLogger.message(RASIMessageEvent.TYPE_WARN, this,
    methodName, "MSG_KEY_01", "some string");
```

If message loggers use the msg API, you can specify a new resource bundle name.

```
msgLogger.msg(RASIMessageEvent.TYPE_ERR, this, methodName,
                "ALT_MSG_KEY_00", "alternateMessageFile");
```

You can also log a text message. If you are using the textMessage API, no message formatting is done.

```
msgLogger.textMessage(RASIMessageEvent.TYPE_INFO, this, methodName,"String and Integer",
"A String", new Integer(5));
```

Using a trace logger

Because trace is normally disabled, guard trace methods for performance reasons.

```
private void methodX(int x, String y, Foo z)
{
   // trace an entry point. Use the guard to make sure tracing is enabled.
Do this checking before you gather parameters to  trace.
   if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT) {
        // I want to trace three parameters, package them up in an Object[]
        Object[] parms = {new Integer(x), y, z};
        trcLogger.entry(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX", parms);
  }
... logic
  // a debug or verbose trace point
  if (trcLogger.isLoggable(RASITraceEvent.TYPE_MISC_DATA) {
        trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA, this, "methodX" "reached here");
  }
  ...
  // Another classification of trace event. An important state change is
 detected, so a different trace type is used.
  if (trcLogger.isLoggable(RASITraceEvent.TYPE_SVC) {
     trcLogger.trace(RASITraceEvent.TYPE_SVC, this, "methodX", "an important event");
  }
 ...
  // ready to exit method, trace. No return value to trace
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
        trcLogger.exit(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX");
    }
}
```

Setting up for integrated JRas operation

Use JRas operations in integrated mode to send trace events and logging messages to only WebSphere Application Server logs.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

In the integrated mode of operation, message and trace events are sent to WebSphere Application Server logs. This approach is the default mode of operation.

1. Import the requisite JRas extensions classes:

```
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;
```

2. Declare logger references:

```
private RASMessageLogger msgLogger = null;
private RASTraceLogger trcLogger = null;
```

3. Obtain a reference to the Manager class and create the loggers. Because loggers are named singletons, you can do this activity in a variety of places. One logical candidate for enterprise beans is the ejbCreate method. For example, for the myTestBean enterprise bean, place the following code in the ejbCreate method:

```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
        myTestBean.class.getName());



// Configure the message logger to use the message file that is created
// for this application.
msgLogger.setMessageFile("acme.widgets.DefaultMessages");
trcLogger = mgr.createRASTraceLogger("Acme", "Widgets", "RasTest",
        myTestBean.class.getName());
mgr.addLoggerToGroup(trcLogger, groupName);
```

Setting up for combined JRas operation

Use JRas operation in combined mode to output trace data and logging messages to both WebSphere Application Server and user-defined logs.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

In combined mode, messages and trace are logged to both WebSphere Application Server logs and user-defined logs. The following sample assumes that:

- You wrote a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter.
- You are not using user-defined types or events.

1. Import the requisite JRas extensions classes:

```
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;
```

2. Import the user handler and formatter:

```
import com.ibm.ws.ras.test.user.*;
```

3. Declare the logger references:

```
private RASMessageLogger msgLogger = null;
  private RASTraceLogger trcLogger = null;
```

4. Obtain a reference to the Manager class, create the loggers, and add the user handlers. Because loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for enterprise beans is the ejbCreate method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must support this approach. The following sample is a message logger sample. The procedure for a trace logger is similar.

```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
    msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
            myTestBean.class.getName());
    // Configure the message logger to use the message file defined
    // in the ResourceBundle sample.
    msgLogger.setMessageFile("acme.widgets.DefaultMessages");

    // Create the user handler and formatter. Configure the formatter,
```

```
      // then add it to the handler.
      RASIHandler handler = new SimpleFileHandler("myHandler", "FileName");
      RASIFormatter formatter = new SimpleFormatter("simple formatter");
       formatter.addEventClass("com.ibm.ras.RASMessageEvent");
       handler.addFormatter(formatter);

      // Add the Handler to the logger. Add the logger to the list of the
      //handlers listeners, then set the handlers
      // mask, which updates the loggers composite mask appropriately.
      // WARNING - there is an order dependency here that must be followed.
      msgLogger.addHandler(handler);
      handler.addMaskChangeListener(msgLogger);
      handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

Setting up for stand-alone JRas operation

You can configure JRas operations to output trace data and logging messages to only user-defined locations.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

In stand-alone mode, messages and traces are logged only to user-defined logs. The following sample assumes that:

- You have a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter.
- You are not using user-defined types of events.

1. Import the requisite JRas extensions classes:

   ```
   import com.ibm.ras.*;
   import com.ibm.websphere.ras.*;
   ```

2. Import the user handler and formatter:

   ```
   import com.ibm.ws.ras.test.user.*;
   ```

3. Declare the logger references:

   ```
   private RASMessageLogger msgLogger = null;
     private RASTraceLogger trcLogger = null;
   ```

4. Obtain a reference to the Manager class, create the loggers, and add the user handlers. Because loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for enterprise beans is the ejbCreate method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must support this approach. The following sample is a message logger sample. The procedure for a trace logger is similar.

   ```
   com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
       msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
               myTestBean.class.getName());
       // Configure the message logger to use the message file that is defined in
       //the ResourceBundle sample.
       msgLogger.setMessageFile("acme.widgets.DefaultMessages");

       // Get a reference to the Handler and remove it from the logger.
       RASIHandler aHandler = null;
       Enumeration enum = msgLogger.getHandlers();
       while (enum.hasMoreElements()) {
           aHandler = (RASIHandler)enum.nextElement();
           if (aHandler instanceof WsHandler)
               msgLogger.removeHandler(wsHandler);
       }

       // Create the user handler and formatter. Configure the formatter,
       // then add it to the handler.
       RASIHandler handler = new SimpleFileHandler("myHandler", "FileName");
   ```

```
    RASIFormatter formatter = new SimpleFormatter("simple formatter");
    formatter.addEventClass("com.ibm.ras.RASMessageEvent");
    handler.addFormatter(formatter);

    // Add the Handler to the logger. Add the logger to the list of the
    // handlers listeners, then set the handlers
    // mask, which will update the loggers composite mask appropriately.
    // WARNING - there is an order dependency here that must be followed.
    msgLogger.addHandler(handler);
    handler.addMaskChangeListener(msgLogger);
    handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

Configuring logging properties using the administrative console

Use this task to browse or change the properties of Java logging.

Before applications can log diagnostic information, you need to specify how you want the server to handle log output, and what level of logging you require. Using the administrative console, you can:

- Enable or disable a particular log, specify where log files are stored and how many log files are kept.
- Specify the level of detail in a log, and specify a format for log output.
- Set a log level for each logger.

You can change the log configuration statically or dynamically. Static configuration changes affect applications when you start or restart the application server. Dynamic or run time configuration changes apply immediately.

When a log is created, the level value for that log is set from the configuration data. If no configuration data is available for a particular log name, the level for that log is obtained from the parent of the log. If no configuration data exists for the parent log, the parent of that log is checked, and so on up the tree, until a log with a non-null level value is found. When you change the level of a log, the change is propagated to the children of the log, which recursively propagates the change to their children, as necessary.

To configure loggers and log handlers for Java logging, use the administrative console to complete the following steps:

1. Set the output properties for a log:
   a. In the navigation pane, click **Servers > Application Servers**.
   b. Click the name of the server that you want to work with.
   c. Under Troubleshooting, click **Logging and tracing**.
   d. Click the name of a system log to configure (Diagnostic Trace, JVM Logs, Process Logs or IBM Service Logs).
   e. To make a static change to the system log configuration, click the **Configuration** tab. To change the configuration dynamically, click the **Runtime** tab.
   f. Change the properties for the selected log according to your needs.
   g. Click **Apply**.
   h. Click **OK**.
2. Set the logging levels for your logs:
   a. In the navigation pane, click **Servers > Application Servers**.
   b. Click the name of the server that you want to work with.
   c. Under Troubleshooting, click **Logging and tracing**.
   d. Click **Change Log Detail levels**.

e. To make a static change to the configuration, click the **Configuration** tab. A list of well-known components, packages, and groups is displayed. To change the configuration dynamically, click the **Runtime** tab. The list of components, packages, and groups displays all the components that are currently registered on the running server.

f. Select a component, package, or group to set a logging level.

g. Click **Apply**.

h. Click **OK**.

3. To have static configuration changes take effect, stop then restart the application server.

Log level settings

Use this topic to configure and manage log level settings.

Using log levels you can control which events are processed by Java logging. When you change the level for a logger, the change is propagated to the children of the logger.

Change Log Detail Levels

Enter a log detail level that specifies the components, packages, or groups to trace. The log detail level string must conform to the specific grammar described in this topic. You can enter the log detail level string directly, or generate it using the graphical trace interface.

If you select the Configuration tab, a static list of well-known components, packages, and groups is displayed. This list might not be exhaustive.

If you select the Runtime tab, the list of components, packages, and group are displayed with all the components that are registered on the running application server and in the static list.

The format of the log detail level specification is:

```
<component> = <level>
```

where <component> is the component for which to set a log detail level, and <level> is one of the valid logger levels (off, fatal, severe, warning, audit, info, config, detail, fine, finer, finest, all). Separate multiple log detail level specifications with colons (:).

Components correspond to Java packages and classes, or to collections of Java packages. Use an asterisk (*) as a wildcard to indicate components that include all the classes in all the packages that are contained by the specified component. For example:

* Specifies all traceable code running in the application server, including the product system code and customer code.

com.ibm.ws.*
Specifies all classes with the package name beginning with com.ibm.ws.

com.ibm.ws.classloader.JarClassLoader
Specifies the JarClassLoader class only.

An error can occur when setting a log detail level specification from the administrative console if selections are made from both the Groups and Components lists. In some cases, the selection made from one list is lost when adding a selection from the other list. To work around this problem, enter the log detail level specification directly into the log detail level entry field.

Select a component or group to set a log detail level. The table following lists the valid levels for application servers at WebSphere Application Server Version 6 and later, and the valid logging and trace levels for earlier versions:

Version 6 logging level	Logging level before Version 6	Trace level before Version 6	Content / Significance

Off	Off	All disabled*	Logging is turned off. * In Version 6, a trace level of All disabled turns off trace, but does not turn off logging. Logging is enabled from the Info level.
Fatal	Fatal	-	Task cannot continue and component, application, and server cannot function.
Severe	Error	-	Task cannot continue but component, application, and server can still function. This level can also indicate an impending fatal error.
Warning	Warning	-	Potential error or impending error. This level can also indicate a progressive failure (for example, the potential leaking of resources).
Audit	Audit	-	Significant event affecting server state or resources
Info	Info	-	General information outlining overall task progress
Config	-	-	Configuration change or status
Detail	-	-	General information detailing subtask progress
Fine	-	Event	Trace information - General trace + method entry, exit, and return values
Finer	-	Entry/Exit	Trace information - Detailed trace
Finest	-	Debug	Trace information - A more detailed trace that includes all the detail that is needed to debug problems
All		All enabled	All events are logged. If you create custom levels, All includes those levels, and can provide a more detailed trace than finest.

When you enable a logging level in Version 6.0 or above, you are also enabling all of the levels with higher severity. For example, if you set the logging level to warning on your Version 6.x application server, then warning, severe and fatal events are processed.

Trace information, which are events at the Fine, Finer and Finest levels, can be written only to the trace log. Therefore, if you do not enable diagnostic trace, setting the log detail level to Fine, Finer, or Finest will not have an effect on the data that is logged.

HTTP error and NCSA access log settings

Use this page to configure an HTTP error log and National Center for Supercomputing Applications (NCSA) access logs for an HTTP transport channel. The HTTP error log contains HTTP errors. The level of error logging that occurs is dependent on the value that is selected for the Error log level field.

To view this administrative console page, click **Application servers** > *server name* > **HTTP error and NCSA access logging**.

The NCSA access log contains a record of all inbound client requests that the HTTP transport channel handles. All of the messages that are contained in these logs are in NCSA format.

After you configure the HTTP error log and the NCSA access logs, make sure that the Enable NCSA access logging field is selected for the HTTP channels for which you want logging to occur. To view the settings for an HTTP channel, click **Servers** > **Application Servers** > *server* > **Web Container Transport Chains** > **HTTP Inbound Channel**.

Enable service at server startup

When selected, either an NCSA access log or an HTTP error log, or both are initialized when the server starts.

Enable access logging

When selected, a record of inbound client requests that the HTTP transport channel handles is kept in the NCSA access log.

Access log file path

Indicates the directory path and name of the NCSA access log. Standard variable substitutions, such as $(SERVER_LOG_ROOT), can be used when specifying the directory path.

Access log maximum size

Indicates the maximum size, in megabytes, of the NCSA access log file. When this size is reached, the *logfile_name*.1 archive log is created. However, every time that the original log file overflows this archive file, the file is overwritten with the most current version of the original log file.

NCSA access log format

Indicates that the NCSA format is used when logging client access information. If Common is selected, the log entries contain the requested resource and a few other pieces of information, but does not contain referral, user agent, or cookie information. If Combined is selected, referral, user agent, and or cookie information is included.

Enable error logging

When selected, HTTP errors that occur while the HTTP channel processes client requests are recorded in the HTTP error log.

Error log file path

Indicates the directory path and the name of the HTTP error log. Standard variable substitutions, such as $(SERVER_LOG_ROOT), can be used when specifying the directory path.

Error log maximum size

Indicates the maximum size, in megabytes, of the HTTP error log file. When this size is reached, the *logfile_name*.1 archive log is created. However, every time that the original log file overflows this archive file, this file is overwritten with the most current version of the original log file.

Error log level

Indicates the type of error messages that are included in the HTTP error log.

You can select:

Critical

Only critical failures that stop the Application Server from functioning properly are logged.

Error The errors that occur in response to clients are logged. These errors require Application Server administrator intervention if they result from server configuration settings.

Warning

Information on general errors, such as socket exceptions that occur while handling client requests, are logged. These errors do not typically require Application Server administrator intervention.

Information

The status of the various tasks that are performed while handling client requests is logged.

Debug

More verbose task status information is logged. This level of logging is not intended to replace RAS logging for debugging problems, but does provide a steady status report on the progress of individual client requests. If this level of logging is selected, you must specify a large enough log file size in the Error log maximum size field to contain all of the information that is logged.

The Common Base Event in WebSphere Application Server

This topic describes how WebSphere Application Server takes advantage of the Common Base Events.

An application creates an event object whenever something happens that either needs to be recorded for later analysis or which might require the trigger of additional work. An *event* is a structured notification that reports information that is related to a situation. An event reports three kinds of information:
* The situation: What happened
* The identity of the affected component: For example, the server that shut down
* The identity of the component that is reporting the situation, which might be the same as the affected component

The application that creates the event object is called the *event source*. Event sources can use a common structure for the event. The accepted standard for such a structure is called the *Common Base Event*. The Common Base Event is an XML document that is defined as part of the autonomic computing initiative. The Common Base Event defines common fields, the values they can take, and the exact meanings of these values.

The Common Base Event model is a standard that defines a common representation of events that is intended for use by enterprise management and business applications. This standard, which is developed by the IBM Autonomic Computing Architecture Board, supports encoding of logging, tracing, management, and business events using a common XML-based format. This format makes it possible to correlate different types of events that originate from different applications. For more information about the Common Base Event model, see the Common Base Event specification (*Canonical Situation Data Format: The Common Base Event V1.0.1*). The common event infrastructure currently supports Version 1.0.1 of the specification.

The basic concept behind the Common Base Event model is the *situation*. A situation can be anything that happens anywhere in the computing infrastructure, such as a server shutdown, a disk-drive failure, or a failed user login. The Common Base Event model defines a set of standard situation types that accommodate most of the situations that might arise (for example, StartSituation and CreateSituation).

The Common Base Event contains all of the information that is needed by the consumers to understand the event. This information includes data about the runtime environment, the business environment, and the instance of the application object that created the event.

For complete details on the Common Base Event format, see the XML schema that is included in the Common Base Event specification document, at ftp://www6.software.ibm.com/software/developer/library/ac-toolkitdg.pdf .

Types of problem determination events

Problem determination involves multiple types of data, including at least two different classes of event data, log events, and diagnostic events.

Log events, which are also referred to as *message events*, are typically emitted by components of a business application during normal deployment and operations. Log events might identify problems, but these events are also normally available and emitted while an application and its components are in production mode. The target audience for log and message events is users and administrators of the application and the components that make up the application. Log events are normally the only events available when a problem is first detected, and are typically used during both problem recovery and problem resolution.

Diagnostic events, which are commonly referred to as *trace events*, are used to capture internal diagnostic information about a component, and are usually not emitted or available during normal deployment and operation. The target audience for diagnostic events is the developers of the components that make up the business application. Diagnostic events are typically used when trying to resolve problems within a component, such as a software failure, but are sometimes used to diagnose other problems, especially when the information provided by the log events is not sufficient to resolve the problem. Diagnostic events are typically used when trying to resolve a problem.

A *Common Base Event* is a common structure for an event. It defines common fields, the values that these fields can take, and the exact meanings of these values for an event. Common Base Events are primarily used to represent log events.

The structure of the Common Base Event

A *Common Base Event* is a common structure for an event. It defines common fields, the values that these fields can take, and the exact meanings of these values for an event.

The Common Base Event contains several structural elements. These elements include:
- Common header information
- Component identification, both source and reporter
- Situation information
- Message data
- Extended data
- Context data
- Associated events and association engine

Each of these structural elements has its own embedded elements and attributes.

The following table presents a summary of all the fields in the Common Base Event and their usage requirements for problem determination events. This table shows whether a particular element or attribute is required, recommended, optional, prohibited, or discouraged for log events, and the base specification.

Field name	Log events	Base specification
Version	Required	Required
creationTime	Required	Required
severity	Required	Optional
Msg	Required	Optional
sourceComponentId*	Required	Required
sourceComponentId.location	Required	Required
sourceComponentId.locationType	Required	Required

sourceComponentId.component	Required	Required
sourceComponentId.subComponent	Required	Required
sourceComponentId.componentIdType	Required	Required
sourceComponentId.componentType	Required	Required
sourceComponentId.application	Recommended	Optional
sourceComponentId.instanceId	Recommended	Optional
sourceComponentId.processId	Recommended	Optional
sourceComponentId.threadId	Recommended	Optional
sourceComponentId.executionEnvironment	Optional	Optional
situation*	Required	Required
situation.categoryName	Required	Required
situation.situationType*	Required	Required
situation.situationType.reasoningScope	Required	Required
situation.situationType.(specific Situation Type elements)	Required	Required
msgDataElement*	Recommended	Optional
msgDataElement .msgId	Recommended	Optional
msgDataElement .msgIdType	Recommended	Optional
msgDataElement .msgCatalogId	Recommended	Optional
msgDataElement .msgCatalogTokens	Recommended	Optional
msgDataElement .msgCatalog	Recommended	Optional
msgDataElement .msgCatalogType	Recommended	Optional
msgDataElement .msgLocale	Recommended	Optional
extensionName	Recommended	Optional
localInstanceId	Optional	Optional
globalInstanceId	Optional	Optional
priority	Discouraged	Optional
repeatCount	Optional	Optional
elapsedTime	Optional	Optional
sequenceNumber	Optional	Optional
reporterComponentId*	Optional	Optional
reporterComponentId.location	Required (2)	Required (2)
reporterComponentId.locationType	Required (2)	Required (2)
reporterComponentId.component	Required (2)	Required (2)
reporterComponentId.subComponent	Required (2)	Required (2)
reporterComponentId.componentIdType	Required (2)	Required (2)
reporterComponentId.componentType	Required (2)	Required (2)
reporterComponentId.instanceId	Optional	Optional
reporterComponentId.processId	Optional	Optional
reporterComponentId.threadId	Optional	Optional
reporterComponentId.application	Optional	Optional
reporterComponentId.executionEnvironment	Optional	Optional
extendedDataElements*	Note 3	Optional

contextDataElements*	Note 4	Optional
associatedEvents*	Note 5	Optional

Notes:

- Items followed by an asterisk (*) are elements that consist of sub elements and attributes. The fields in those elements are listed in the table directly following the parent element name.
- Some of the elements are optional, but when included, they include sub elements and attributes that are required. For example, the reporterComponentId element has a ComponentIdentification type. The component attribute in ComponentIdentification is required. Therefore, the reporterComponentId.component attribute is required, but only when the reporterComponentId parent element is included.
- The extendedDataElements element can be included multiple times to supply extended data information. See the Extended data section for more information on required and recommended extended data element values.
- The contextDataElements element can be included multiple times to supply context data information.
- The associatedEvents element can be included multiple times to supply correlation data. No recommended uses of this element exist for the producers of problem determination data, and the use of this element is discouraged.

Common header information

This topic provides additional information about how to format and use these fields for problem determination events, which can be used to clarify and extend the information provided in the other documents.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

The common header information in the Common Base Event includes the following information about an event:

- Version: The version of this Common Base Event
- creationTime: The date and time when the event generated
- Severity and priority: The severity of the condition (situation) that is identified by the event
- extensionName: The type of event that was captured
- localInstanceId and globalInstanceId: Identifiers that can be used to quickly identify a specific event within a set of events
- repeatCount and elapsedTime: Information that supports a system to efficiently report multiple events of the same type, by consolidating those events into a single event
- sequenceNumber: Sequence information that supports a system to order a set of events in other ways than time of capture

severity

All problem determination events must provide an indication as to the relative severity of the condition (situation) being reported by providing appropriate values for the severity field in the Common Base Event. The severity field is required for problem determination events. This field is more restrictive than the base specification for the Common Base Event, which lists this field as optional because effective and efficient problem determination requires the ability to quickly identify the information that is needed to resolve a problem as well as prioritize the problems that need addressing. Typically, the following values are used for problem determination events:

10	Information	Log information events, normal conditions, and events that are supplied to clarify operations, for example, state transitions, operational changes. These events typically do not require administrator action or intervention.
20	Harmless	Similar to information events, but are used to capture audit items, such as state transitions or operational changes. These events typically do not require administrator action or intervention.
30	Warning	Warnings typically represent recoverable errors, for example a failure that the system can correct. These events can require administrator action or intervention.
40	Minor	Minor errors describe events that represent an unrecoverable error within a component. The failure affects the component ability to service some requests. The business application can continue to perform its normal functions, but its overall operation might be degraded. These events require administrator action or intervention to address the condition.
50	Critical	Critical errors describe events that represent an unrecoverable error within a component. The failure significantly affects the component ability to service most requests. The business application can continue most, but not all of its normal functions and its overall operation might be degraded. These events require administrator action or intervention to address the condition.
60	Fatal	Fatal errors describe events that represent an unrecoverable error within a component. The failure usually results in the complete failure of the component. The business application can continue some normal functions, but its overall operation might be degraded. These events require administrator action or intervention to address the condition.

msg

Refer to "Message data" on page 1512 for information on this attribute.

priority

The use of the priority field is discouraged for problem determination events. The severity field is typically used to communicate and evaluate the importance of problem determination events. Use the priority field to enhance the information that is provided in the severity field, that is. prioritize events of the same severity.

extensionName

The extensionName field is used to communicate the type of event that is reported, for example, what general class of events is being reported. In many cases this field provides an indication of what additional data you can expect with the event, for example, optional data values.

repeatCount

The repeatCount field is valid for problem determination events, but is not typically used or supplied by the event producers. This field is used for data reduction and consolidation by event management and analysis systems.

elapsedTime

The elapsedTime field is valid for problem determination events, but is not typically used or supplied by the event producers. This field is used for data reduction and consolidation by event management and analysis systems.

sequenceNumber

The sequenceNumber field is valid for problem determination events. It is typically used only by event producers when the granularity of the event time stamp (the creationTime field) is not sufficient in ordering events. The sequenceNumber field is typically used to sequence events that have the same time stamp value.

Event management and analysis systems can use the sequenceNumber field for a number of reasons, including providing alternative sequencing, not necessarily based on a time stamp. The recommendations here are provided primarily for event producers.

Component identification for source and reporter

The component identification fields in the Common Base Event are used to indicate which component in the system is experiencing the condition that is described by the event (the sourceComponentID) and which component emitted the event (the reporterComponentID). Typically, these components are the same, in which case only the sourceComponentID is supplied. Some notes and scenarios on when to use these two elements in the Common Base Event:

- The sourceComponentID is always used to identify the component experiencing the condition that is described by the event.

- The reporterComponentID is used to identify the component that actually produced and emitted the event. This element is typically used only within events that are emitted by a component that is monitoring another component and providing operational information regarding that component. The monitoring component (for example, a Tivoli agent or hardware device driver) is identified by the reporterComponentID and the component being monitored (for example, a monitored server or hardware device) is identified by the sourceComponentID.

  A potential misuse of the reporterComponentID is to identify a component that provides event conversion or management services for a component, for example, identifying an adapter that transforms the events that are captured by a component into Common Base Event format. The event conversion function is considered an extension of the component and not identified separately.

The information that is used to identify a component in the system is the same, regardless of whether it is the source component or reporter component:

location locationType	Component location	Identifies the location of the component.
component componentType	Component name	Identifies the asset name of the component, as well as the type of component.
subcomponent	Subcomponent name	Identifies a specific part or subcomponent of a component, for example a software module or hardware part.

application	Business application name	Identifies the business application or process the component is a part of and provides services for.
instanceId	Operational instance	Identifies the operational instance of a component, that is the actual running instance of the component.
processId threadId	Operational instance	Identifies the operational instance of a component within the context of a software operating system, that is he operating system process and thread running when the event was produced.
executionEnvironment	Operational instance Component location	Provides additional information about the operational instance of a component or its location by identifying the name of the environment hosting the operational instance of the component, for example the operating system name for a software application, the application server name for a Java 2 Platform, Enterprise Edition (J2EE) application, or the hardware server type for a hardware part.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines. This section provides additional information about how to format and use some of these fields for problem determination events, which can be used to clarify and extend the information that is provided in the other documents.

Component
    The Component field in a problem determination event is used to identify the manageable asset that is associated with the event. A manageable asset is open for interpretation, but a good working definition is a manageable asset represents a hardware or software component that can be separately obtained or developed, deployed, managed, and serviced. Examples of typical component names are:

- IBM eServer xSeries model x330
- IBM WebSphere Application Server version 5.1 (5.1 is the version number)
- Microsoft Windows 2000
- The name of an internally developed software application for a component

subComponent
    The Subcomponent field in a problem determination event identifies the specific part of a component that is associated with the event. The subcomponent name is typically not a manageable asset, but provides internal diagnostic information when diagnosing an internal defect within a component, that is What part failed? Examples of typical subcomponents and their names are:

- Intel Pentium processor within a server system (Intel Pentium IV Processor)
- the enterprise bean container within a Web application server (enterprise bean container)
- the task manager within an operating system (Linux Kernel Task Manager)
- the name of a Java class and method (myclass.mycompany.com or myclass.mycompany.com.methodname).

The format of a subcomponent name is determined by the component, but use the convention shown previously for naming a Java class or the combination of a Java class and method is followed. The subcomponent field is required in the Common Base Event.

componentIdType

The componentIdType field is required by the Common Base Event specification, but provides minimal value for problem determination events. For problem determination events, the use of the application value is discouraged. The componentIdType field identifies the type of component; the application is identified by the application field.

application

The application field is listed as an optional value within the Common Base Event specification, but provide it within problem determination events whenever it this value is available. The only reason this field is not required for problem determination events is that instances exist where the issuing component might not be aware of the overall business application.

instanceId

The instanceId field is listed as an optional value within the Common Base Event specification, but provide this value within problem determination events whenever it is available.

Always provide the instanceID when a software component is identified and identify the operational instance of the component (for example, which operation instance of an installed software image is actually associated with the event). Provide this value for hardware components when these components support the concept of operational instances.

The format of the supplied value is defined by the component, but must be a value that an analysis system can use (either human or programmatic) to identify the specific running instance of the identified component. Examples include:

- **cell, node, server** name for the IBM WebSphere Application Server
- **deployed EAR file name** for a Java enterprise bean
- **serial number** for a hardware processor

processId

The processId field is listed as an optional value within the Common Base Event specification, but provide this value for problem determination events whenever it is available and applicable. Always provide this value for software-generated events, and identify the operating system process that is associated with the component that is identified in the event. Match the format of the thread ID with the format of the operating system (or other running environment, such as a Java virtual machine). This field is typically not applicable or used for events that are emitted by hardware (for example, firmware).

threadId

The threadId field is listed as an optional value within the Common Base Event specification, but provide this value for problem determination events whenever it is available and applicable. Always provide for software-generated events, and identify the active operating system thread when the event was detected or issued. A notable exception to this recommendation is some operating systems or running environments do not support threads. Match the format of the thread ID with the format of the operating system (or other running environment, such as a Java virtual machine). This field is typically not applicable or used for events that are emitted by hardware (for example, firmware).

executionEnvironment

The executionEnvironment field, when used, identifies the immediate running environment that is used by the component being identified. Some examples are:

- the operating system name when the component is a native software application.
- the operating system/Java virtual machine name when the component is a Java 2 Platform, Standard Edition (J2SE) application.
- the Web server name when the component is a servlet.
- the portal server name when the component is a portlet.
- the application server name when the component is an enterprise bean.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

Situation information

The situation information is used to classify the condition that is reported by an event into a common set of situations.

The Common Base Event specification [CBE101] provides information on the set of situations defined for the Common Base Event, with the values and formats that are used to describe these situations. The Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

Consider the following points regarding situation information for problem determination events:
- Whenever possible, use the situation categorizations and qualifiers that are described in the base Common Base Event specification. Avoid using your own situation definitions as much as possible.
- Not all messages and logs can be classified using the situation definitions that are supplied in the base Common Base Event specification. You can use the OtherSituation categorization to provide your own situation information, but the recommended course of action for problem determination events is to use the ReportSituation categorization, with reportCategory=Log.
- Warning events can be confusing. A warning event (that is an event with severity=warning) typically indicates a recoverable failure, but the situation settings can be interpreted as unrecoverable failures (for example ConnectSituation, successDisposition=UNSUCCESSFUL). Use the appropriate situation categorization so the severity setting indicates the severity of the situation, that is whether the component recovered from the failure.
- The recommended setting for the reasoningScope value is EXTERNAL for all message events.

Message data

All problem determination Common Base Events must provide human readable text that describes the specific reported event within the msg field of the Common Base Event.

The text that is associated with events representing actual messages or log entries is expected to be translated and localized. Include the msgDataElement element in the Common Base Event whenever internationalized text is provided in the event. This element provides information about how the message text is created and how to interpret it. This information is particularly invaluable when trying to interpret the event programmatically or when trying to interpret the message independent of the locale or language that is used to format the message text.

Prerequisite: Understand the concepts that are associated with creating internationalized messages. A good source of education on these concepts is provided by the documentation that is associated with internationalization of Java information and the usage of resource bundles within the Java language.

The msgDataElement element in the Common Base Event includes the following information about the value of the msg field that is provided with an event:
- The locale of the supplied message text, which identifies how the locale-independent fields within the message are formatted, as well as the language of the message (msgLocale).
- A locale-independent identifier that is associated with the message that can be used to interpret the message independent of the message language, message locale, and message format (msgId and msgIdType).
- Information on how a translated message is created, including:
  - The identifier that is used to retrieve the message template (msgCatalogId).
  - The name and type of message catalog that are used to retrieve the message template (msgCatalog and msgCatalogType).
  - Any locale-independent information that is inserted into the message template to create the final message (msgCatalogTokens).

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines. This section provides additional information about how to format and use these fields for problem determination events.

msg

All message, log, and trace events must provide a human-readable message in the msg field of the Common Base Event. The msg field is required for problem determination events, both log events and diagnostic events. This field is more restrictive than the base specification for the Common Base Event, which lists this field as optional; effective and efficient problem determination requires the ability to quickly identify the reported condition. The format and usage of this message is component-specific, but use the following general guidelines:

- Expect the message text that is supplied with messages and log events to be internationalized.
- Provide the locale of the supplied message text with the msgLocale field in the msgDataElement element of the Common Base Event.
- Provide additional information regarding the format and construction of internationalized messages whenever possible, using the msgDataElement element of the Common Base Event.

msgLocale

Provide the message locale whenever message text is provided within the Common Base Event, as is the case with all problem determination events. The msgLocale field is listed as an optional value within the Common Base Event specification, but provide this information within problem determination events whenever possible. The reason this field is not required for problem determination events is that instances exist where the locale information is not provided or available when formatting the Common Base Event.

msgId and msgIdType

Several companies include a locale-independent identifier within internationalized message text that you can use to interpret the described condition by the message text, independent of the message. For example, most messages issued by IBM software look like `IEE890I WTO Buffers in console backup storage = 1024`, where a unique, locale-independent identifier `IEE890I` precedes the translated message text. This identifier provides a way to uniquely detect and identify a message independent of location and language. This detection is invaluable for locale-independent and programmatic analysis.

The msgId field is listed as an optional value within the Common Base Event specification, but it must be provided within problem determination events whenever this identifier is included in the message text. Likewise, the msgIdType field is listed as an optional value within the Common Base Event specification, but it must be provided within problem determination events whenever a value is supplied for msgId. Do not supply these fields when the message text is not translated or localized, for example, for trace events.

msgCatalogId

The msgCatalogId field is listed as an optional value within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. This field is not required for problem determination events because not all problem determination events include translated message text Some cases exist where the value is not provided or available when formatting the Common Base Event. Do not supply this field when the message text is not translated or localized, for example, for trace events.

msgCatalogTokens

The msgCatalogTokens field is listed as an optional value within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. This field is not required for problem determination events because not all problem determination events include translated message text, and cases exist where the value is not

provided or available when formatting the Common Base Event. This value contains the list of locale-independent values or message tokens that are inserted into the localized message text when creating a translated message.

These values are difficult to extract from a translated message without knowing the translated message template that is used to create the message. Do not supply this field when the message text is not translated or localized

The Common Base Event provides several mechanisms for providing additional data about an event, including this field, extended data elements, and extensions to the schema. Always use the msgCatalogTokens field to supply the list of message tokens that is included in the message text associated with an event. These values can also be supplied in other parts of the Common Base Event, but they must be included in this field.

msgCatalog and msgCatalogType

The msgCatalog and msgCatalogType fields are listed as optional values within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. These fields are not required for problem determination events because not all problem determination events include translated message text, and cases exist where the values are not provided or available when formatting the Common Base Event. Do not complete these fields when the message text has is not translated or localized, for example, for trace events.

Extended data

The Common Base Event provides several methods for including this additional data, including extending the Common Base Event schema or supplying one or more ExtendedDataElement elements within the Common Base Event, which is the preferred approach.

The base information that is included in a Common Base Event might not be sufficient to represent all of the information captured by a component when creating a problem determination event.

Use an ExtendedDataElement element to represent a single data item. A Common Base Event can contain more than one of these elements, essentially one for each additional data item. A hint to the number and type of ExtendedDataElement elements is supplied by the extensionName value, but this information is only a hint. The usage of the attributes in the ExtendedDataElement element for problem determination events is the same as those for any other Common Base Event.

Sample Common Base Event instance

This XML document is an example of a Common Base Event instance that is generated by a WebSphere Application Server application.

Use the following example for reference:

```
<CommonBaseEvent  creationTime="2004-09-18T04:03:28.484Z"
   globalInstanceId="myhost:1095479647062:1899"
   msg="WSVR0024I: Server server1 stopped"
   severity="10"
   version="1.0.1">

 ... several extendedDataElements for WebSphere Application Server internal use only ...

<sourceComponentId  component="com.ibm.ws.runtime.component.ServerCollaborator"
   componentIdType="Unknown"
   executionEnvironment="Windows 2000[x86]#5.0"
   instanceId="myhost\myhost\server1"
   location="myhost"
   locationType="Hostname"
   processId="1095479647062"
   subComponent="Unknown"
   threadId="Alarm : 0"
   componentType="http://www.ibm.com/namespaces/autonomic/WebSphereApplicationServer"/>
```

```
<msgDataElement msgLocale="en_US">
 <msgCatalogTokens value="server1"/>
 <msgId>WSVR0024I< /msgId>
 <msgCatalogId>WSVR0024I< /msgCatalogId>
 <msgCatalog>com.ibm.ws.runtime.runtime< /msgCatalog>
</msgDataElement>

<situation categoryName="ReportSituation">
 <situationType xsi:type="ReportSituation" reasoningScope="EXTERNAL" reportCategory="LOG"/>
</situation>

</CommonBaseEvent>
```

A number of extendedDataElement elements in the XML are used by WebSphere Application Server, but are not for application use because these elements might change.

The CommonBaseEvent element defines the Common Base Event instance. This element has a set of attributes that are common for all Common Base Events. This set includes the extensionName attribute, which defines the type or class of the Common Base Event instance, the creation time, severity, and priority.

Nested within the CommonBaseEvent element are elements giving more detail about the situation. The first of these elements is the situation element. This classification is standardized.

The CommonBaseEvent element also includes the sourceComponentId and the (optional) reporterComponentId elements. The sourceComponentId element describes where the situation occurred; the reporterComponentId describes where the situation is detected. If the sourceComponentId and the reporterComponentId elements are the same, the reporterComponentId element is omitted.

The attributes of both the sourceComponentId and the reporterComponentId elements are the same. They identify the component type, name, operating system, and network location. The content of these attributes provides vertical correlation of the stack of IT resources that are active when the Common Base Event is created.

Also included in the CommonBaseEvent element are contextDataElements elements that describe the context in which the situation occurred. This context correlates Common Base Event instances that are part of the same work. This correlation is called *horizontal correlation* because an instance of a particular context type correlates events at the same level of abstraction, for example at the business level, the application level, or at the middleware level.

FExtended data elements contain additional data that is used to describe a situation. In this example, an extended data element is added by WebSphere Application Server to describe the Java 2 Platform, Enterprise Edition (J2EE) component that generated the Common Base Event instance and some application data.

Sample Common Base Event template

The content handler uses template information to fill in blanks in the Common Base Event when the Common Base Event complete method is called.

Components that use the WebSphere Application Server event factory home can include a Common Base Event template XML file to provide data to populate Common Base Events. Information that is already supplied in the event is not overridden if the same field is supplied in the template.

The following example illustrates a Common Base Event template:

```
<?xml version="1.0" encoding="UTF-8"?>

<TemplateEvent
```

```
  version="1.0.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

 <CommonBaseEvent
  <sourceComponentId application="My Application" component="com.ibm.componentX"/>
  <extendedDataElements name="Sample ExtendedDataElement name" type="string">
   <values>Sample ExtendedDataElement value</values>
  </extendedDataElements>
 </CommonBaseEvent>

</TemplateEvent>
```

Component identification for problem determination

This topic describes types of problem determination events.

A business application is made up of multiple components. A component can be made up of several internal subcomponents. Consistent application of these concepts is critical for effective problem determination of a business application; all of the parts of the application must use the same concepts and assumptions when creating and formatting events. Use the following definitions and examples when creating Common Base Events for problem determination.

Business application
> A business application is the business logic and business data that is used to address a set of specific business requirements. A business application consists of several components of multiple types, combined in a unique manner by an enterprise, to provide the functions and resources that are needed to address those requirements. The primary creator and manager of a business application is the enterprise, and each enterprise or company creates unique business applications. Examples of business applications are the Payroll Application for the ACME Corporation and the Inventory Application for Spacely Sprockets.

Components
> A business application is created and managed by the enterprise as a set of components. Components are deployable assets, which are developed either by the enterprise or a vendor, and managed by the enterprise. A component might be created by the enterprise, typically for use within a specific business application. For example, the ACME Corporation might create a set of enterprise beans to represent the business logic that is required by their Payroll Application. A component might also be an asset that is produced by a vendor and acquired by an enterprise. Examples of these components are hardware products, such as IBM eServers or Sun Solaris systems, or software products, such as IBM WebSphere Application Server, Oracle Database Servers.

Subcomponents
> A specific component, depending on its complexity, might consist of several subcomponents. For example, the IBM WebSphere Application Server consists of many subcomponents, such as the enterprise bean container and the servlet engine. Subcomponent information is typically used only by the creator of the component to service the component, and as such are not separately deployable or manageable resources in the enterprise. The enterprise might deploy a change or update to a subcomponent, but only upon guidance from the component vendor and as part of the vendor's component. For example, a software fix for the enterprise bean container of the IBM WebSphere Application Server is packaged and deployed as a software update to the IBM WebSphere Application Server. Replacement of the processor in an IBM eServer is deployed as a physical part, but only as a part of the original deployed component, the IBM eServer.

Logging Common Base Events in WebSphere Application Server

This topic describes how WebSphere Application Server takes advantage of the Common Base Events.

WebSphere Application Server uses Common Base Events within its logging framework. Common Base Events can be created explicitly and then logged through the Java logging API, or can be created implicitly

by using the Java logging API directly. For Common Base Event creation, the application server environment provides a Common Base Event factory with a content handler that provides both runtime data and template data for Common Base Events.

Logging with Common Base Event API and the Java logging API

In cases where the events that are generated by the Java logging API are insufficient to describe the event that needs capturing, you can create Common Base Events with the Common Base Event factory APIs.

When you create a Common Base Event, you can add data to the Common Base Event before it is logged. The following diagram illustrates how application code can create and log Common Base Events:
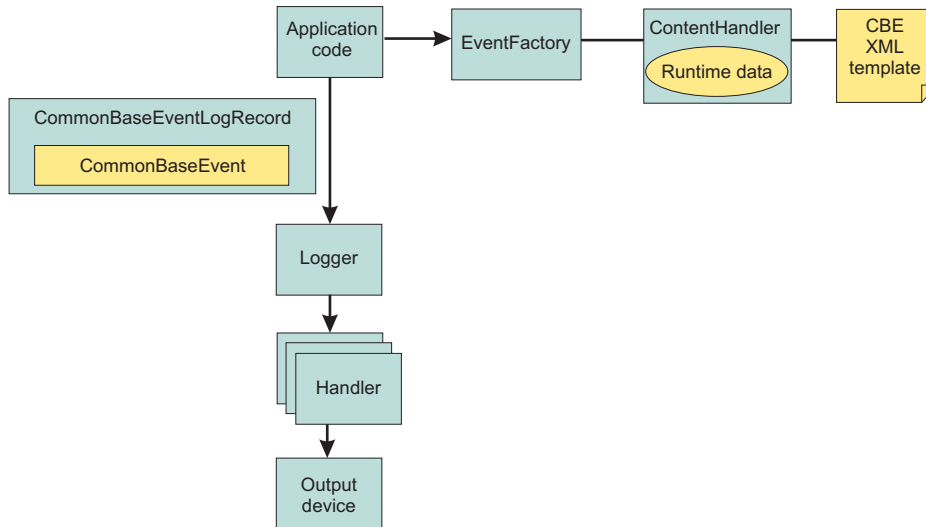
WebSphere Application Server is configured to use an event factory that automatically populates WebSphere Application Server-specific information into the Common Base Events that it generates. In general, it is good practice to create events using the WebSphere Application Server default Common Base Event factory because this approach ensures consistency of Common Base Event content across events. However, you can create and use other Common Base Event factories.

- Application code invokes the createCommonBaseEvent method on the EventFactory class to create a CommonBaseEvent.
- Application code wraps CommonBaseEvent event in a CommonBaseEventLogRecord record, and adds event-specific data.
- Application code calls the CommonBaseEvent event complete method.
- The CommonBaseEvent event invokes the ContentHandler completeEvent method.
- The ContentHandler handler adds XML template data to the CommonBaseEvent event. Not all ContentHandler handlers support templates.
- The ContentHandler handler adds runtime data to the CommonBaseEvent event.
- Application code passes the CommonBaseEventLogRecord record to the logger using the Logger.log method.
- Logger passes CommonBaseEventLogRecord record to Handlers.
- Handlers format data and write to the output device.

After completing all the above steps you will have a Common Base event based on your configuration settings.

Common Base Event content handler:

Content handlers populate data into Common Base Events when the Common Base Event complete method is invoked. You can associate content handlers with Common Base Event templates, which provide default information to transfer into each Common Base Event. Content handlers might also provide any other information that is relevant to completing the population of the Common Base Event, such as appropriate runtime defaults.

The use of content handlers ensures consistency of field use in the Common Base Event within a component or within a set of components that share the same runtime. For example, some content handlers support the specification of a template. If used consistently across a component, this template ensures that all events for that component have the same template information filled in. Similarly, some content handlers can also supply runtime information to their associated Common Base Events. If consistently used throughout the entire runtime, runtime information ensures that all events use runtime data in a similar way.

The event factory home that is used in the WebSphere Application Server runtime is associated with a content handler that both reads from a template, and supplies runtime data. Have components use Event Factories that are obtained from this event factory home with their own templates, to produce consistency between application events and server events.

More details can be found in"Creating custom Common Base Event content handlers" or the API documentation for org.eclipse.hyades.logging.events.cbe.ContentHandler at www.eclipse.org/hyades.

Creating custom Common Base Event content handlers:

Create a custom Common Base Event content handler or template to automate configuration or values for specific events.

A *content handler* is an object that automatically sets the property values of each event based on any arbitrary policies that you want to use.

The following content handler classes were added to WebSphere Application Server to facilitate the use of the Common Base Event infrastructure:

Class Name	Description
WsContentHandlerImpl	This provides an implementation of org.eclipse.hyades.logging.events.cbe.ContentHandler specifically for use in the WebSphere Application Server environment. This content handler completes Common Base Events using information from the WebSphere Application Server runtime, and it uses the same content handler as is used internally by the WebSphere Application Server when completing Common Base Events for logging.
WsTemplateContentHandlerImpl	This provides the same function as WsContentHandlerImpl, but it extends the org.eclipse.hyades.logging.events.cbe.impl.TemplateContentHandlerImpl class to enable the use of a Common Base Event template. Template content takes precedence in cases where the template data specifies values for the same Common Base Event fields as does the WsContentHandlerImpl.

In some situations, you might want some event property data set automatically for every event that you create. This automation is a way to fill in certain standard values that do not change, such as the application name, or to set some properties based on information that is available from the runtime environment, like creation time or thread information. You can set property data automatically by creating a content handler.

- Use the following code sample to implement the CustomContentHandler class:

```
public class CustomContentHandler extends WsContentHandlerImpl {

 public CustomContentHandler() {
  super();
  // TODO Custom initialization code goes here
 }

 public void completeEvent(CommonBaseEvent cbe) throws CompletionException {
  // following code will add WAS content to the Content Base Event
  super.completeEvent(cbe);
  // TODO Custom content can be added to the Content Base Event here
 }
}
```

- The following shows how to implement the CustomTemplateContentHandler class:

```
public class CustomTemplateContentHandler extends WsTemplateContentHandlerImpl {

 public CustomTemplateContentHandler() {
  super();
  // TODO Custom initialization code goes here
 }

 public void completeEvent(CommonBaseEvent cbe) throws CompletionException {
  // following code will add WAS content to the Content Base Event
  super.completeEvent(cbe);
  // TODO Custom content can be added to the Content Base Event here
 }
}
```

You now have a content handler or a custom content handler template based on the settings that you specified.

Common Base Event factory home:

Event Factory homes provide Event Factory instantiation that is based on a unique factory name.

Event Factory home implementations are tightly coupled with content handlers that are used to populate Common Base Events with template or default data. Event Factory instances are maintained by the associated Event Factory home, based on their unique name. For example, when application code requests a named Event Factory, the newly created Event Factory instance is returned and persisted for future requests for that named Event Factory. An abstract Event Factory home class provides the implementation for the APIs in the Event Factory home interface. Implementers extend the abstract Event Factory home class and implement the createContentHandler API to create a typed content handler that is based on the type of Event Factory home implementation.

In WebSphere Application Server, the default Event Factory home that is obtained with a call to EventFactoryContext.getInstance.getEventFactoryHome method is associated with a ContentHandler handler capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactoryHome at www.eclipse.org/hyades.

Creating custom Common Base Event factory homes:

Use custom Common Base Event factory homes to control configuration and implementation of unique Event Factories.

Event Factory Homes create and provide homes for Event Factory instances. Each Event Factory Home has a Content Handler. This Content Handler is assigned to every Event Factory the Event Factory Home creates. In turn, when a Common Base Event is created, the Content Handler from the Event Factory is

assigned to it. Event Factory instances are maintained by the associated Event Factory Home, based on their unique name. For example, when application code requests a named Event Factory, the newly created Event Factory instance is returned and persisted for future requests for that named Event Factory.

The following classes were added to facilitate the use of Event Factory homes for logging Common Base Events:

Class Name	Description
WsEventFactoryHomeImpl	This class extends the org.eclipse.hyades.logging.events.cbe.impl.AbstractEventFactoryHome class. This Event Factory Home returns Event Factory instances associated with the WsContentHandlerImpl Content Handler. The WsContentHandlerImpl is the Content Handler used by the WebSphere Application Server by default when no Event Factory template is in use.
WsTemplateEventFactory HomeImpl	This class extends the org.eclipse.hyades.logging.events.cbe.impl.EventXMLFileEventFactoryHomeImpl class. This Event Factory Home returns Event Factory instances associated with the WsTemplateContentHandlerImpl Content Handler. The WsTemplateContentHandlerImpl is the Content Handler used by the WebSphere Application Server when an Event Factory template is required.

Custom event factory homes support the use of Common Base Event for logging in WebSphere Application Server and make logging easy and consistent between the WebSphere Application Server runtime and the exploiters of this API. The CustomEventFactoryHome and CustomTemplateEventFactoryHome classes will be used to obtain an event factory. These classes are there to make sure the correct content handler is being used with a particular event factory. The CustomEventFactoryHelper class is an example of how the infrastructure provider can hide the factory selection details from infrastructure users, using their own set of parameters to decide which the appropriate event factory is.

- The following code samples provide examples of how to implement and use the CustomEventFactoryHome class.

  1. Implementation of the CustomEventFactoryHome class is as follows:

```
public class CustomEventFactoryHome extends AbstractEventFactoryHome {

 public CustomEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomContentHandler();
 }
}
```

  2. The following is an example of how to use the CustomEventFactoryHome class:

```
// get the event factory
 EventFactory eventFactory=(new CustomEventFactoryHome()).getEventFactory("XYZ");
 // create an event - call appropriate method
 eventFactory.createCommonBaseEvent();
 // log event ...
```

- For the CustomTemplateEventFactoryHome class you can use the following code for implementation and use:

1. Implement the CustomTemplateEventFactoryHome class by using this code:

```
public class CustomTemplateEventFactoryHome extends
  EventXMLFileEventFactoryHomeImpl {

 public CustomTemplateEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomTemplateContentHandler();
 }
}
```

2. Use the CustomTemplateEventFactoryHome class by following this sample code:

```
// get the event factory
  EventFactory eventFactory=(new
    CustomTemplateEventFactoryHome()).getEventFactory("XYZ");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

- The CustomEventFactoryHelper class can be implemented and used by following the code below:

1. Implement the custom CustomEventFactoryHelper class using this code:

```
public class CustomTemplateEventFactoryHome extends
  EventXMLFileEventFactoryHomeImpl {

 public CustomTemplateEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomTemplateContentHandler();
 }
}
Figure 4     CustomTemplateEventFactoryHome class
public class CustomEventFactoryHelper {
 // name of the event factory to use
 public static final String FACTORY_NAME="XYZ";

 public static EventFactory getEventFactory(String param1, String param2) {
  EventFactory factory=null;
  switch (resolveFactory(param1,param2)) {
  case 1:
   factory=(new CustomEventFactoryHome()).getEventFactory(FACTORY_NAME);
   break;
  case 2:
   factory=(new
    CustomTemplateEventFactoryHome()).getEventFactory(FACTORY_NAME);
   break;

  default:
   // Add default for event factory
```

```
   break;
  }
  return factory;
 }

 private static int resolveFactory(String param1, String param2) {
  int factory=0;
  // Add code here to resolve which factory to use
  return factory;
 }
}
```

2. To use the CustomEventFactoryHelper class, use the following code:

```
// get the event factory
  EventFactory eventFactory=
   CustomEventFactoryHelper.getEventFactory("param1","param2","param3");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

Use the information provided here to implement a custom content factory home and the associated classes based on the settings that you specify.

Common Base Event factory context:

The event factory context provides a service to look up event factory homes. Retrieve the event factory context using a call to the EventFactoryContext.getInstance method.

Using this class, you can look up the event factory homes by name, and avoid the need to include the typed home in code. The EventFactoryHome name must be located on the class path to be found. The EventFactoryContext context also stores an EventFactoryHome name as a default, which can be obtained with a call to the EventFactoryContext.getInstance.getEventFactoryHome method.

In WebSphere Application Server, the EventFactoryContext context is configured with a default EventFactoryHome name which is associated to a ContentHandler handler that is capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactory at www.eclipse.org/hyades.

Common Base Event factory:

Use event factories to create Common Base Events and complete event properties with associated content handlers.

Content handlers populate data into Common Base Events when the Common Base Event invokes the complete method. All event properties set by the application code have priority over all properties that are specified by the content handler. Event factory implementations are tightly coupled with the content handler instance, which is associated with the event factory when the event factory is instantiated. Factory instances can be retrieved only from their associated event factory home. Event factory instances are retrieved and maintained based on unique names. Event factory names are hierarchical; they are represented using the standard Java dot-delimited, name-space naming conventions.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactory at www.eclipse.org/hyades.

java.util.logging -- Java logging programming interface
The java.util.logging.Logger class provides a variety of methods with which data can be logged.

In the WebSphere Application Server, the Java logging API (java.util.logging) automatically creates Common Base Events for events that are logged at the WsLevel.DETAIL level or above (including WsLevel.DETAIL, Level.CONFIG, Level.INFO, WsLevel.AUDIT, Level.WARNING, Level.SEVERE, and WsLevel.FATAL). These Common Base Events are created using the event factory that is associated with the logger to which the message is logged. If no event factory is specified, WebSphere Application Server uses a default event factory which automatically fills in WebSphere Application Server-specific information.

The WebSphere Application Server uses a special implementation of the java.util.logging.Logger class that automatically creates Common Base Events for the following methods:

- config
- info
- warning
- severe
- log: All variants except log(LogRecord) when used with the WsLevel.DETAIL level or more severe levels
- logp: When used with the WsLevel.DETAIL level or more severe levels
- logrb: When used with the WsLevel.DETAIL level or more severe levels

The WebSphere Application Server logger implementation is used only for named loggers for example, loggers that are instantiated with calls, such as Logger.getLogger(″com.xyz.SomeLoggerName″). Loggers instantiated with calls to the Logger.getAnonymousLogger and Logger.getLogger, or Logger.global methods do not use the WebSphere Application Server implementation, and do not automatically create Common Base Events for logging requests made to them. Log records that are logged directly with the Logger.log(LogRecord) method are not automatically converted by WebSphere Application Server loggers into Common Base Events.

The following diagram illustrates how application code can log Common Base Events:

The Java logging API processing of named loggers and message-level events proceeds as follows:

1. Application code invokes the named logger (WsLevel.DETAIL or above) with event-specific data.
2. The logger creates a Common Base Event using the createCommonBaseEvent method on the event factory that is associated with the logger.
3. The logger creates a Common Base Event using the event factory associated to the logger.
4. The logger wraps the common base event in a CommonBaseEventLogRecord record, and adds event-specific data.
5. The logger calls the Common Base Event complete method.
6. The Common Base Event invokes the ContentHandler completeEvent method.

7. The content handler adds XML template data to the Common Base Event (including for example, the component name). Not all content handlers support templates.
8. The content handler adds runtime data to the Common Base Event (including for example, the current thread name).
9. The logger passes the CommonBaseEventLogRecord record to the handlers.
10. The handlers format data and write to the output device.

Logger.properties file
Use the Logger.properties file to set logger attributes for your component.

The properties file is loaded the first time the Logger.getLogger(loggername) method is called within an application. The Logger.properties file must be either on the WebSphere Application Server class path, or the context class path.

The logging subsystem uses Common Base Events to represent all the messages in the WebSphere Application Server activity.log file. You can specify your own event factory template to be used with your loggers. Use the eventfactory property in your Logger.properties file. See "Sample Common Base Event template" on page 1515 for details on the Common Base Event template.

By convention, the name of the event factory template file should be the fully qualified package name of the package using the template. The name of the file must end with the `.event.xml` extension. For example, a valid event factory template file name for the com.abc.somepackage package is:

`com.abc.somepackage.event.xml`

When you specify the property value for the eventfactory property in the Logger.properties file, include the full path name with no leading slash relative to the root of your class path entry. Do not include the `.event.xml` extension.

For example, if the template files from the example above are located in the `com/abc/templates` directory, the valid value for the eventfactory property is:

`com/abc/templates/com.abc.somepackage`

Finally, if this event factory template file is used by the com.abc.somepackage.SomeClass logger, then the following entry will appear in the Logger.properties file:

`com.abc.somepackage.SomeClass.eventfactory=com/abc/templates/com.abc.somepackage`

Generate Common Base Event content with the default event factory
A default Common Base Event content handler populates Common Base Events with WebSphere Application Server runtime information. This content handler can also use a Common Base Event template to populate Common Base Events.

The default content handler is used when the server creates CommonBaseEventLogRecords as would be the case in the following example:

```
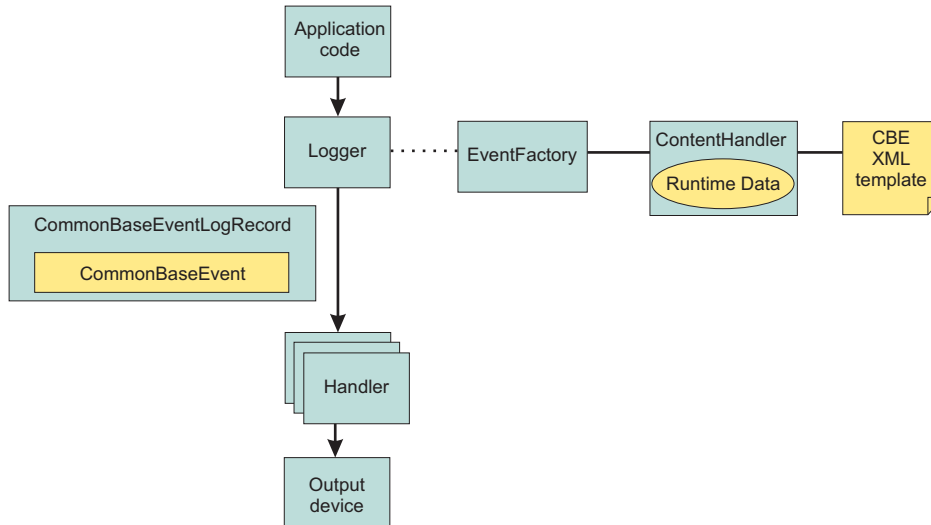// Get a named logger
Logger logger = Logger.getLogger("com.ibm.someLogger");
// Log to the logger -- implicitly the default content handler
// will be associated with the CommonBaseEvent contained in the
// CommonBaseEventLogRecord. logger.warning("MSG_KEY_001");
```

To specify a Common Base Event template in the above case, a Logger.properties file would need to be provided with an eventfactory entry for *com.ibm.someLogger*. If a valid template is found on the classpath, then the Logger's event factory will use the specified template's content in addition to the WebSphere Application Server runtime information when populating Common Base Events. If the template is not found on the classpath, or is invalid, then the Logger's event factory will only use the WebSphere Application Server runtime information when populating Common Base Events.

The default content handler is also associated with the event factory home supplied in the global event factory context. This is convenient for creating Common Base Events that need to be populated with content similar to that generated from the WebSphere Application Server:

```
// Request the event factory from the global event factory home
EventFactory eventFactory = EventFactoryContext.getInstance().getEventFactoryHome().getEventFactory(templateName);

// Create a Common Base Event
CommonBaseEvent commonBaseEvent = eventFactory.createCommonBaseEvent();

// Complete the Common Base Event using content from the template (if specified above)
// and the server runtime information.
eventFactory.getContentHandler().completeEvent(commonBaseEvent);
```

In the above example, if the template referenced by *templateName* is found on the classpath, and the template is valid, then the event factory home will return an event factory which uses a content handler that combines the template's content with the WebSphere Application Server runtime information when populating Common Base Events. If the template is not found on the classpath, or is invalid, then the event factory home will return an event factory which uses a content handler that uses only the WebSphere Application Server runtime information when populating Common Base Events.

The default content handler populates Common Base Events in the server environment with the following runtime information:

CommonBaseEvent.globalInstanceId
> **Value:** The *unique_record_id*

> Set this value only if the CommonBaseEvent.globalInstanceId value is null before the completeEvent method is called.

CommonBaseEvent.msg
> **Value:** A localized message that is based on the MsgDataElement element.

> Set this value only if the CommonBaseEvent.msg message is null before the completeEvent method is called.

CommonBaseEvent.severity
> **Value:** Set based on the value of level set on the CommonBaseEventLogRecord record, if level >= Level.SEVERE, set to 50; if level >= Level.WARNING, set to 30; the default is set to 10.

> Set this value only if the CommonBaseEvent.severity value is null before the completeEvent method is called.

CommonBaseEvent.ComponentIdentification.component
> **Value:** Set based on the LoggerName value that is set on the CommonBaseEventLogRecord record.

> Set this value only if the CommonBaseEvent.ComponentIdentification.component is null before the completeEvent method is called.

CommonBaseEvent.ComponentIdentification.componentIdType
> **Value:** ″Unknown″

> Set this value only if the CommonBaseEvent.ComponentIdentification.componentIdType value is null before the completeEvent method is called.

CommonBaseEvent.ComponentIdentification.executionEnvironment
> **Value:** OSname[OSarch]#OSversion

> Set this value only if the CommonBaseEvent.ComponentIdentification.executionEnvironment value is null before the completeEvent method is called.

CommonBaseEvent.ComponentIdentification.instanceId
> **Value:** cellName\nodeName\serverName

Set this value only if the CommonBaseEvent.ComponentIdentification.instanceId value is null before the completeEvent method is called. Set only in a server environment because this value is ignored in a client application.

CommonBaseEvent.ComponentIdentification.location
> **Value:**The host name

> Set this value only if both the CommonBaseEvent.ComponentIdentification.location and the CommonBaseEvent.ComponentIdentification.locationType values are null before the completeEvent method is called.

CommonBaseEvent.ComponentIdentification.locationType
> **Value:** The host name

> Set this value only if both the CommonBaseEvent.ComponentIdentification.location and the CommonBaseEvent.ComponentIdentification.locationType values are null before the completeEvent method is called.

CommonBaseEvent.ComponentIdentification.processId
> **Value:** An internally generated representation of the process number.

> Set this value only if the CommonBaseEvent.ComponentIdentification.processId value is null before the completeEvent method is called

CommonBaseEvent.ComponentIdentification.subComponent
> **Value:** Set based on values of the sourceClassName and the sourceMethodName names that are set on the sourceClassName.sourceMethodName name of the CommonBaseEventLogRecord record.

> Set this value only if the CommonBaseEvent.ComponentIdentification.subComponent values is null before the completeEvent method is called and both the sourceClassName and the sourceMethodName names are set.

CommonBaseEvent.ComponentIdentification.threadId
> **Value:** Set to the value of the Java Virtual Machine (JVM) thread name.

> Set this value only if the CommonBaseEvent.ComponentIdentification.threadId values is null before the completeEvent value is called.

CommonBaseEvent.ComponentIdentification.componentType
> **Value:** http://www.ibm.com/namespaces/autonomic/WebSphereApplicationServer

> Set this value only if the CommonBaseEvent.ComponentIdentification.componentType values is null before the completeEvent method is called.

CommonBaseEvent.MsgDataElement.msgLocale
> **Value:** Set based on the default locale of the JVM.

> Set this value only if the CommonBaseEvent.msg value is null before the completeEvent method is called.

CommonBaseEvent.Situation.categoryName
> **Value:** `ReportSituation`

> Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

CommonBaseEvent.Situation.situationType.type
> **Value:** `ReportSituation`

> Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

CommonBaseEvent.Situation.situationType.reasoningScope
> **Value:** EXTERNAL

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

CommonBaseEvent.Situation.situationType.reportCategory
> **Value:** LOG

> Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

The sourceComponentIdentification value is populated if no reporterComponentIdentification ID exists when the completeEvent method is invoked on the content handler. Otherwise, the reporterComponentIdentification ID is populated instead.

Best practices for logging Common Base Events in WebSphere Application Server

The following practices ensure consistent use of Common Base Events within your components, and between your components and WebSphere Application Server components.

Follow these guidelines:
- Use a different logger for each component. Sharing loggers across components gets in the way of associating loggers with component-specific information.
- Associate loggers with event templates that specify source component identification. This association ensures that the source of all events created with the logger is properly identified.
- Use the same template for directly created Common Base Events (events created using the Common Base Event factories) and indirectly created Common Base Events (events created using the Java logging API) within the same component.
- Avoid calling the complete method on Common Base Events until you are finished adding data to the Common Base Event and are ready to log it. This approach ensures that any decisions made by the content handler based on data already in the event are made using the final data.

The following sample `Logger.properties` file entry demonstrates how to associate the com.ibm.componentX logger with the com.ibm.componentX event factory:

```
com.ibm.componentX.eventfactory=com.ibm.componentX
```

The following sample code demonstrates the use of the same event factory setting for direct (Part 1) and indirect (Part 2) Common Base Event logging:

```
<?xml version="1.0" encoding="UTF-8"?>

<TemplateEvent
  version="1.0.1"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

 <CommonBaseEvent
   <sourceComponentId application="My application" component="com.ibm.componentX"/>
   <extendedDataElements CommonBaseEventname="Sample ExtendedDataElement name" type="string">
   <values>Sample ExtendedDataElement value</values>
  </extendedDataElements>
 < /CommonBaseEvent>

< /TemplateEvent>
```

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 Network Deployment.

Default product locations when the root user or an administrator user installs the product

The root user or administrator user (on a Windows system) is capable of registering shared products and installing into system-owned directories. The following default directories are system-owned directories.

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server Network Deployment products or components, of course, require multiple locations.

app_server_root - the install_root for WebSphere Application Server
>
> The following list shows default installation root directories for WebSphere Application Server Network Deployment:
>
> | AIX | /usr/IBM/WebSphere/AppServer
>
> | HP-UX | /opt/IBM/WebSphere/AppServer
>
> | Linux | /opt/IBM/WebSphere/AppServer
>
> | Solaris | /opt/IBM/WebSphere/AppServer
>
> | Windows | C:\Program Files\IBM\WebSphere\AppServer

profile_root
>
> The following list shows the default directory for a profile named *profile_name* on each distributed operating system:
>
> | AIX | /usr/IBM/WebSphere/AppServer/profiles/*profile_name*
>
> | HP-UX | /opt/IBM/WebSphere/AppServer/profiles/*profile_name*
>
> | Linux | /opt/IBM/WebSphere/AppServer/profiles/*profile_name*
>
> | Solaris | /opt/IBM/WebSphere/AppServer/profiles/*profile_name*
>
> | Windows | C:\Program Files\IBM\WebSphere\AppServer\profiles*profile_name*

plugins_root
>
> The following default installation root is for the Web server plug-ins for WebSphere Application Server:
>
> | AIX | /usr/IBM/HTTPServer/Plugins
>
> | HP-UX | /opt/IBM/HTTPServer/Plugins
>
> | Linux | /opt/ibm/HTTPServer/Plugins
>
> | Solaris | /opt/IBM/HTTPServer/Plugins
>
> | Windows | C:\Program Files\IBM\HTTPServer\Plugins

web_server_root
>
> The following default installation root directories are for the IBM HTTP Server:

> **AIX**   /usr/IBM/HTTPServer

> **HP-UX**   /opt/IBM/HTTPServer

> **Linux**   /opt/ibm/HTTPServer

> **Solaris**   /opt/IBM/HTTPServer

> **Windows**   C:\Program Files\IBM\HTTPServer

gskit_root

The following list shows the default installation root directories for Version 7 of the IBM Global Security Kit (GSKit):

> **AIX**   /usr/ibm/gsk7

> **HP-UX**   /opt/ibm/gsk7

> **Linux**   /opt/ibm/gsk7

> **Solaris**   /opt/ibm/gsk7

> **Windows**   C:\Program Files\IBM\GSK7

app_client_root

The following default installation root directories are for the WebSphere Application Client:

> **AIX**   /usr/IBM/WebSphere/AppClient (J2EE Application client only)

> **HP-UX**   /opt/IBM/WebSphere/AppClient (J2EE Application client only)

> **Linux**   /opt/IBM/WebSphere/AppClient (J2EE Application client only)

> **Solaris**   /opt/IBM/WebSphere/AppClient (J2EE Application client only)

> **Windows**   C:\Program Files\IBM\WebSphere\AppClient

updi_root

The following list shows the default installation root directories for the Update Installer for WebSphere Software:

> **AIX**   /usr/IBM/WebSphere/UpdateInstaller

> **HP-UX**   /opt/IBM/WebSphere/UpdateInstaller

> **Linux**   /opt/IBM/WebSphere/UpdateInstaller

> **Solaris**   /opt/IBM/WebSphere/UpdateInstaller

> **Windows**   C:\Program Files\IBM\WebSphere\UpdateInstaller

cip_app_server_root

The following list shows the default installation root directories for a customized installation package (CIP) produced by the Installation Factory.

A CIP is a WebSphere Application Server Network Deployment product bundled with one or more maintenance packages, an optional configuration archive, one or more optional enterprise archive files, and other optional files and scripts:

> **AIX**   /usr/IBM/WebSphere/AppServer/cip/*cip_uid*

> **HP-UX**   /opt/IBM/WebSphere/AppServer/cip/*cip_uid*

> **Linux**   /opt/IBM/WebSphere/AppServer/cip/*cip_uid*

/opt/IBM/WebSphere/AppServer/cip/*cip_uid*

C:\Program Files\IBM\WebSphere\AppServer\cip*cip_uid*

The *cip_uid* variable is the CIP unique ID generated during creation of the build definition file. You can override the generated value in the Build definition wizard. Use a unique value to allow multiple CIPs to install on the system.

component_root

The component installation root directory is any installation root directory described in this topic. Some programs are for use across multiple components. In particular, the Update Installer for WebSphere Software is for use with WebSphere Application Server Network Deployment, Web server plug-ins, the Application Client, and the IBM HTTP Server. All of these components are part of the product package.

Default product locations when a non-root user or a non-administrator user installs the product

The non-root user or non-administrator user (on a Windows system) is not capable of registering shared products and installing into system-owned directories. The following default directories are user-owned directories in the home directory of the non-root installer as opposed to being globally shared resources that are available to all users.

app_server_root

The following list shows the default installation directories for non-root installation of WebSphere Application Server Network Deployment:

AIX *user_home*/IBM/WebSphere/AppServer

HP-UX *user_home*/IBM/WebSphere/AppServer

Linux *user_home*/IBM/WebSphere/AppServer

Solaris *user_home*/IBM/WebSphere/AppServer

Windows C:\IBM\WebSphere\AppServer

profile_root

The following list shows the default directories for creating profiles:

AIX *user_home*/IBM/WebSphere/AppServer/profiles/

HP-UX *user_home*/IBM/WebSphere/AppServer/profiles/

Linux *user_home*/IBM/WebSphere/AppServer/profiles/

Solaris *user_home*/IBM/WebSphere/AppServer/profiles/

Windows C:\IBM\WebSphere\AppServer\profiles\

web_server_root

The following default installation root directories are for the IBM HTTP Server:

AIX *user_home*/IBM/HTTPServer

HP-UX *user_home*/IBM/HTTPServer

Linux *user_home*/ibm/HTTPServer

Solaris *user_home*/IBM/HTTPServer

Windows C:\IBM\HTTPServer

plugins_root

The following list shows the default installation root directories for the Web server plug-ins for WebSphere Application Server:

`AIX`   *user_home*/IBM/HTTPServer/Plugins

`HP-UX`   *user_home*/IBM/HTTPServer/Plugins

`Linux`   *user_home*/ibm/HTTPServer/Plugins

`Solaris`   *user_home*/IBM/HTTPServer/Plugins

`Windows`   C:\IBM\HTTPServer\Plugins

app_client_root

The following list shows the default installation root directories for the WebSphere Application Client:

`AIX`   *user_home*/IBM/WebSphere/AppServer/AppClient (J2EE Application client only)

`HP-UX`   *user_home*/IBM/WebSphere/AppClient (J2EE Application client only)

`Linux`   *user_home*/IBM/WebSphere/AppClient (J2EE Application client only)

`Solaris`   *user_home*/IBM/WebSphere/AppClient (J2EE Application client only)

`Windows`   C:\IBM\WebSphere\AppClient

updi_root

The following list shows the default installation directories for non-root installation of WebSphere Application Server Network Deployment:

`AIX`   *user_home*/IBM/WebSphere/UpdateInstaller

`HP-UX`   *user_home*/IBM/WebSphere/UpdateInstaller

`Linux`   *user_home*/IBM/WebSphere/UpdateInstaller

`Solaris`   *user_home*/IBM/WebSphere/UpdateInstaller

`Windows`   C:\Program Files\IBM\WebSphere\UpdateInstaller

cip_app_server_root

The following list shows the default installation root directories for a WebSphere Application Server Network Deployment product CIP:

`AIX`   *user_home*/IBM/WebSphere/AppServer/cip/*cip_uid*

`HP-UX`   *user_home*/IBM/WebSphere/AppServer/cip/*cip_uid*

`Linux`   *user_home*/IBM/WebSphere/AppServer/cip/*cip_uid*

`Solaris`   *user_home*/IBM/WebSphere/AppServer/cip/*cip_uid*

`Windows`   C:\IBM\WebSphere\AppServer\cip*cip_uid*

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

Trademarks and service marks

For trademark attribution, visit the IBM Terms of Use Web site (http://www.ibm.com/legal/us/).