



Developing and deploying applications

Note

Before using this information, be sure to read the general information under “Notices” on page 1161.

Compilation date: December 6, 2004

© Copyright International Business Machines Corporation 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

How to send your comments	vii
Chapter 1. Overview and new features for developing and deploying applications	1
Overview of developing and deploying applications	2
Learn about WebSphere applications: Overview and new features	2
What is new for developers	9
Assembly tools	11
Enterprise (J2EE) applications	12
Chapter 2. Designing applications	13
Chapter 3. Obtaining an integrated development environment (IDE)	15
Chapter 4. Developing WebSphere applications	17
Web applications	17
Learn about Web applications	17
Task overview: Developing and deploying Web applications	20
Task overview: Managing HTTP sessions	57
Developing servlets with WebSphere Application Server extensions	69
Developing Web applications	76
Developing session management in servlets	80
Assembling Web applications	82
Assembling so that session data can be shared	84
EJB applications	85
Learn about EJB applications	85
Task overview: Using enterprise beans in applications	88
Developing enterprise beans	92
Using access intent policies	113
Assembling EJB modules	122
Deploying EJB modules	126
Client applications	127
Learn about client applications	127
Using application clients	129
Running application clients	142
Developing application clients	160
Developing ActiveX application client code	161
Developing applet client code	176
Developing J2EE application client code	178
Developing Pluggable application client code	182
Developing Thin application client code	182
Assembling application clients	184
Deploying J2EE application clients on workstation platforms	185
Web services	263
Learn about Web services	263
Implementing Web services applications	265
Deploying Web services	278
Configuring Web service client bindings	280
Configuring the scope of a Web service port	283
Web Services Invocation Framework (WSIF): Enabling Web services	284
Getting started with UDDI Registry	289
Planning to use Web services	290
Setting up and deploying a new UDDI Registry	295
Developing Web services applications	306

Configuring Web services deployment descriptors	358
Developing Web services clients	358
Developing client bindings from a WSDL file	360
UDDI Registry Client Programming	361
Assembling Web services applications	378
Data access resources	390
Learn about data access resources	390
Task overview: Accessing data from applications	392
Developing data access applications	421
Assembling data access applications	511
Deploying data access applications	515
Messaging resources	602
Learn about messaging resources	602
Learning about messaging with WebSphere Application Server	604
Installing and configuring a JMS provider	615
Programming to use asynchronous messaging	616
Mail, URLs, and other J2EE resources	644
Learn about mail, URLs, and other J2EE resources	644
Using mail	645
Using URL resources within an application.	649
Resource environment entries	652
Enabling debugger for a mail session	656
Security	657
Securing applications and their environments.	657
Planning to secure your environment.	658
Implementing security considerations at installation time.	668
Developing secured applications	673
Assembling secured applications	734
Deploying secured applications	744
Testing security.	756
Naming and directory	757
Learn about naming and directory	757
Using naming	758
Developing applications that use JNDI	768
Developing applications that use CosNaming (CORBA Naming interface)	784
Configuring and viewing name space bindings	788
Configuring name servers	792
Object Request Broker	792
Learn about Object Request Brokers (ORB)	792
Managing Object Request Brokers.	793
Transactions	811
Learn about transactions	811
Using the transaction service.	812
Developing components to use transactions	822
Using one-phase and two-phase commit resources in the same transaction	826
WebSphere programming extensions.	830
ActivitySessions	830
Application profiling	851
Asynchronous beans.	869
Dynamic cache	890
Dynamic query	902
Internationalization	932
Object pools	969
Scheduler	975
Startup beans	993
Work area.	995

Chapter 5. Debugging applications	1003
Debugging with the Application Server Toolkit	1004
Chapter 6. Assembling applications	1005
Application assembly and J2EE applications	1006
Starting an assembly tool	1007
Configuring an assembly tool	1007
Archive support in Version 6.0	1009
Migrating code artifacts to an assembly tool	1009
Importing enterprise applications	1010
Importing WAR files	1010
Importing client applications	1011
Importing EJB files	1012
Importing RAR files or connectors	1013
Creating enterprise applications	1014
Creating Web applications	1015
Creating EJB modules	1016
Creating application clients	1018
Creating connector modules	1019
Editing deployment descriptors	1020
Mapping enterprise beans to database tables	1022
Mapping constraints for databases	1023
Verifying archive files	1023
Generating code for Web service deployment	1024
Assembling applications: Resources for learning	1025
Chapter 7. Class loading	1027
Class loaders	1027
Configuring class loaders of a server	1031
Class loader collection	1032
Class loader ID	1032
Class loader mode	1032
Class loader settings	1033
Configuring application class loaders	1033
Configuring Web module class loaders	1034
Configuring class preloading	1035
Class loading: Resources for learning	1036
Chapter 8. Deploying and administering applications	1039
System applications	1039
Installing application files	1039
Installable module versions	1040
Ways to install applications or modules	1041
Installing application files with the console	1042
Example: Installing an EAR file using the default bindings	1053
Installing J2EE modules with JSR-88	1053
Customizing modules using DConfigBeans	1055
Enterprise application collection	1056
Name	1056
Status	1057
Enterprise application settings	1057
Configuring an application	1060
Application bindings	1062
Mapping modules to servers	1066
Starting and stopping applications	1067
Disabling automatic starting of applications	1068

Target mapping collection	1068
Exporting applications	1069
Exporting DDL files	1069
Updating applications	1070
Ways to update application files	1073
Preparing for application update settings	1074
Hot deployment and dynamic reloading	1077
Uninstalling applications	1085
Removing a file	1085
Deploying and administering applications: Resources for learning	1086
Chapter 9. Troubleshooting deployment	1089
Errors or problems deploying, installing, or promoting applications	1089
Troubleshooting testing and first time run problems	1093
Errors starting an application	1094
The application does not start or starts with errors	1098
A web resource does not display	1099
Cannot uninstall an application or remove a node or application server	1101
Chapter 10. Adding logging and tracing to your application	1103
Logging and tracing with Java logging	1103
Loggers	1104
Log handlers	1105
Log levels	1106
Log filters	1106
Log formatters.	1106
Configuring logging properties using the administrative console.	1107
Configuring logging properties for an application	1111
Sample security	1112
Using loggers in an application.	1112
The Common Base Event in WebSphere Application Server	1122
Types of problem determination events	1123
The structure of the Common Base Event	1123
Sample Common Base Event instance.	1133
Sample Common Base Event template	1134
Component Identification for Problem Determination.	1134
Logging Common Base Events in WebSphere Application Server	1135
Programming with the JRas framework	1141
Understanding the JRas facility	1141
JRas Extensions	1143
JRas Messages and Trace event types	1151
Instrumenting an application with JRas extensions	1154
Notices	1161
Trademarks and service marks	1163

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

This topic summarizes the contents and organization of the development documentation, including links to conceptual overviews and descriptions of new features.

- “Overview of developing and deploying applications” on page 2
- “What is new for developers” on page 9
- “Assembly tools” on page 11
- “Enterprise (J2EE) applications” on page 12

Sections in the development documentation:

Chapter 3, “Obtaining an integrated development environment (IDE),” on page 15

This topic describes how to obtain design tools and development environments for use with your WebSphere Application Server product edition.

Chapter 2, “Designing applications,” on page 13

This topic helps you find best practices for developing your Java 2 Platform, Enterprise Edition (J2EE)-compliant applications.

Chapter 4, “Developing WebSphere applications,” on page 17

This topic provides development, assembly, and deployment instructions that are specific to various types of applications. For example, you can focus on developing Web applications or Web services; or extending your applications with the application programming interfaces of the messaging or security subsystems.

Chapter 5, “Debugging applications,” on page 1003

This topic provides information on how to import a WebSphere Application Server program into a Java project for debugging.

Chapter 6, “Assembling applications,” on page 1005

This topic describes how to assemble enterprise application modules (EAR files) from new or existing Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 or 1.3 modules, including these archives: Web application archives (WAR), resource adapter archives (RAR), enterprise bean Java archive (JAR) files, and application client archives. This packaging and configuration of code artifacts into application modules or stand-alone Web modules is necessary for deploying the applications onto the application server.

Chapter 7, “Class loading,” on page 1027

This topic describes how to configure class loaders. It includes both configuration performed during application assembly (packaging) and configuration performed at the server. The product run-time environment uses class loaders to find and load new classes for an application. Class loaders are part of the Java virtual machine (JVM) code and are responsible for finding and loading class files.

Chapter 8, “Deploying and administering applications,” on page 1039

This topic describes how to deploy applications onto application servers, and then how to administer the deployed applications. It includes installing applications, starting applications, exporting application files, updating applications, removing applications, and other common tasks

Chapter 9, “Troubleshooting deployment,” on page 1089

This topic describes how to identify and handle a variety of problems encountered during development, assembly, and deployment activities.

Overview of developing and deploying applications

This topic provides links to conceptual overviews of developing, assembling, and deploying your applications into the application serving environment.

“What is new for developers” on page 9

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)

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

Presentations from Education on Demand

- Deployment and administration - all application types
 - Application management overview
 - Installing and uninstalling applications
 - Managed application resources - Enhanced EAR files
 - Fine grained application update
- Class loading
 - Class loader overview
 - Class loader example
 - Class loader problem determination and best practices
 - Class loader - dynamic application reloading
- Programming model
 - J2EE 1.4 Web services overview
 - Web services in Version 6
 - For more conceptual overviews, see: Web services concepts
 - Naming and directory concepts
 - For more conceptual overviews, see: Naming and directory concepts
 - IBM service integration technologies overview
 - For more conceptual overviews, see: JMS resources concepts
 - JDBC
 - JavaServer Faces (JSF) overview
 - Service Data Objects (SDO) overview
 - For more conceptual overviews, see: Data access concepts
 - WebSphere programming model extensions overview
 - For more conceptual overviews, see: Learn about WebSphere applications

Learn about WebSphere applications: Overview and new features

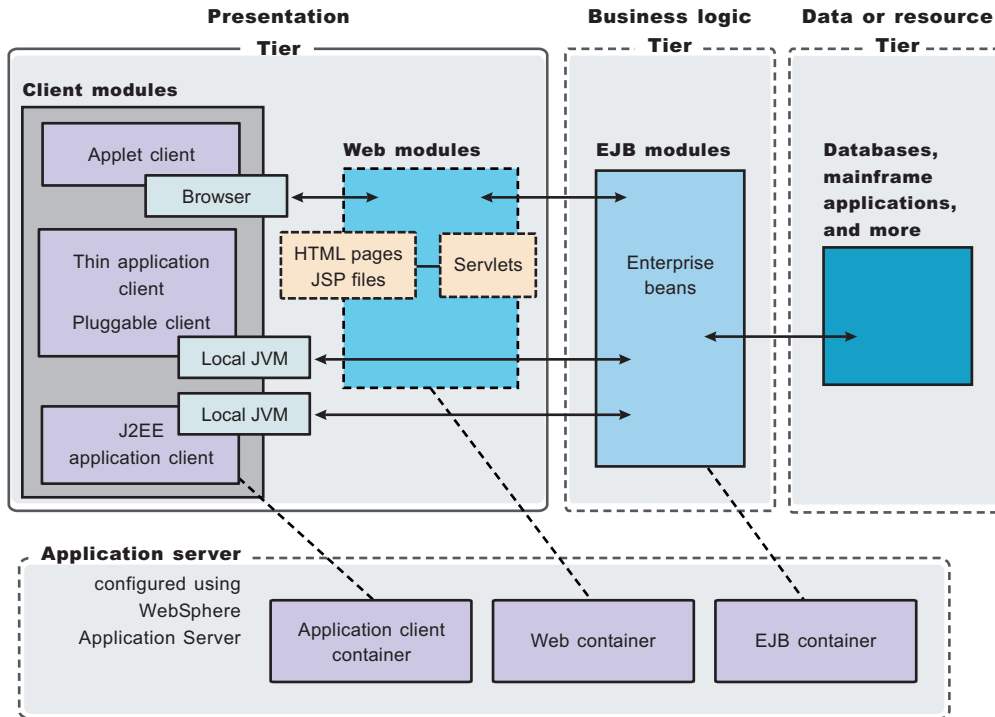
Use the **Learn about WebSphere applications** section as a starting point to study technologies used in and by applications deployed on the application server.

WebSphere applications use the following kinds of technologies:

- 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 following diagram shows applications and their clients, and the application server containers and services in a single application server installation. The remainder of this topic introduces each technology, including the many “WebSphere programming extensions” on page 830 (not shown).



Product architecture and programming model, at a glance

Product subsystems	WebSphere applications	WebSphere applications
<p>Servers</p> <ul style="list-style-type: none"> • Application servers • More server types • Core groups • Workload balancing <p>Environment</p> <ul style="list-style-type: none"> • Virtual hosts • Variable settings • Shared libraries <p>System administration</p> <ul style="list-style-type: none"> • Administrative clients • Configuration files • Domains (cells, nodes) <p>Performance</p> <ul style="list-style-type: none"> • Monitoring • Tuning performance <p>Troubleshooting</p> <ul style="list-style-type: none"> • Diagnostic tools • Support and self-help <p>The product subsystems, for the most part, do not depend on the type of applications being deployed</p>	<p>Services</p> <ul style="list-style-type: none"> • Security • Naming • ORB • Transactions <p>J2EE applications</p> <ul style="list-style-type: none"> • Web applications • EJB applications <p>Clients</p> <ul style="list-style-type: none"> • Client applications • Web clients • Web services clients • Administrative clients <p>Web services</p> <ul style="list-style-type: none"> • Web services and Service Oriented Architecture (SOA) 	<p>J2EE resources</p> <ul style="list-style-type: none"> • Data access resources • Messaging resources • Mail, URLs, and more <p>WebSphere extensions</p> <ul style="list-style-type: none"> • ActivitySessions • Application profiling • Asynchronous beans • Dynamic caching • Dynamic and EJB query • Internationalization • Object pools • Scheduler • Startup beans • Work areas

J2EE application components

Web applications

The diagram shows the Web container, 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.

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

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."

EJB (Enterprise JavaBeans) applications

The diagram shows the EJB container, which provides all of the runtime services needed to administer enterprise beans. The container is an interface between EJB components and the

application server. It provides many low-level services, including threading and transaction support. From an administrative perspective, the container handles data access for the contained beans.

Enterprise beans are Java components that typically implement the business logic of J2EE applications, as well as accessing data.

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

The diagram shows a Java client running in the client container, which is installed separately on the client machine. It enables the client to run applications in an EJB-compatible J2EE environment. Depending on the source of technical information, client applications might be called application clients. This documentation tends to use the two terms synonymously

Web clients

The diagram shows a Web browser client making a request to the Web container of the application server, by route of the HTTP server. A Web client or Web browser client runs in a Web browser, and is comprised of Web application components. See the Web applications section for Web client information.

Web services clients

The diagram shows the Web services container, but does not depict a Web services client. Web services clients are yet another kind of client that might exist in your application serving environment. See the Web services section for details about this kind of client.

Administrative clients

The diagram shows the administrative interface (Admin UI) and a scripting client accessing parts of the systems administration infrastructure. Although the current discussion focuses on the technologies in the programming model, administrative clients are mentioned here 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 very flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

Data access, messaging, and J2EE resources

Data access resources

The diagram shows the application using the JCA container to access an application database. Connection management for access to enterprise information systems (EIS) in the application server is based on the J2EE Connector Architecture (JCA) specification. 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 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 documentation often groups data access under the heading of J2EE resources for simplicity.

Messaging resources

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. For compatibility with WebSphere Application Server version 5, you can configure JMS message-driven beans against a listener port.

Mail, URLs, and other J2EE resources

J2EE resources are used by applications deployed on a J2EE-compliant application server. They include:

- JDBC resources and other technology for data access (previously discussed)
- JMS resources and other messaging system support (previously discussed)
- JavaMail support, for applications to send Internet mail
- URLs, for describing logical locations
- Resource environment entries, for mapping logical names to physical names

Security

See the Securing WebSphere applications PDF

The diagram shows the security server. 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.

Additional services for use by applications

Naming and directory

The name server provides a Java Naming and Directory Interface (JNDI) name space. The naming service registers resources hosted on the application server. It 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.

With this product, you receive the following 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

Object Request Broker (ORB)

The product uses an ORB to manage communication between client applications and server applications, as well as among product components.

An ORB manages the interaction between clients and servers, using IIOP. It enables 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.

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.

WebSphere extensions

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

What is new for developers

This topic highlights what is new or changed in Version 6 for users who are going to develop and assemble WebSphere applications.

The biggest improvement in development is support for the J2EE 1.4 specification, enabling you to take advantage of the latest Java technology, as described in "Java 2 Platform, Enterprise Edition (J2EE) specification" in the information center.

In addition, deploying applications has never been easier -- particularly redeploying updated applications or modules. For what is new with deployment, see "New: Application deployment improvements" in the information center.

The remainder of this topic describes smaller but significant changes in the development and assembly infrastructure, and for various types of WebSphere applications.

WebSphere extensions

Several more WebSphere extensions are now available in this product edition, with new features relative to their recent inclusion in the WebSphere Business Integration Server Foundation, Version 5.1 release. As a starting point for learning about each extension, see "WebSphere programming extensions" on page 830. See also the WebSphere extensions section in "Learn about WebSphere applications: Overview and new features" on page 2.

Web services

WebSphere Application Server has been a leader in advocating support for Web services standards that allow more automated, less hand-coded cross-platform computing. New 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 WebSphere Application Server Version 6 supports the WS-I Basic Profile 1.1 for development of interoperable Web services supporting the integration of Web services solutions. WebSphere Application Server also supports 30 operating system platforms, the most in the industry.

Enterprise beans can be invoked from a Web services client using RMI-IIOP

More detail about Web services support is provided in the following items. WebSphere Application Server Version 6 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, 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.

New extensions to the JSR-101 and JSR-109 programming models

For more information, refer to “Using WSDL EJB bindings to invoke an EJB from a Web services client” on page 345.

WebSphere Application Server Version 6.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 318.

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: 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 323 and “WSDL2Java command” on page 327.

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.

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 Web Developer. These tools are referred to in this information center as the *assembly tools*.

Either or both tools are available on separate CD-ROMs in your WebSphere Application Server CD-ROM package.

The assembly feature of the AST and Rational Web 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. Follow instructions available with AST or Rational Web Developer to install an assembly tool.

Although this information center refers to the AST and Rational Web Developer products as the *assembly tools*, you can use AST and Rational Web Developer to do more than assemble modules. Rational Web

Developer is an integrated development environment that provides development, testing, assembly and deployment capabilities. However, articles on application assembly in this information center focus on assembling J2EE modules using the J2EE Perspective of the assembly tools. AST and Rational Web Developer provide extensive online documentation; the articles on application assembly in this information center supplement that documentation. The **Application Server Toolkit** information center is available with this information center.

Note: You can also use Rational Application Developer to assemble application files, though it is available only on a trial basis in the WebSphere Application Server CD-ROM package.

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
- Zero or more Web modules
- Zero or more connector modules (packaged in RAR 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.

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.

- 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.

- Consult the following Web resources for learning.

The top 10 (more or less) J2EE best practices

The authors, which 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.

Best practices for using WebSphere Application Server Web services

The author states: Web services performance comes of age in WebSphere Application Server Version 5.0.2, but just as with more traditional J2EE applications, the performance of Web services applications is largely determined by the design of the application and the database. This article considers the application design factors unique to Web services performance, including the most important: moving to WebSphere Application Server 5.0.2. I will examine the performance of WebSphere Application Server 5.0.2 Web services and establish some best practices for optimizing Web services performance on WebSphere Application Server 5.0.2

WebSphere Application Server V5 TechNote: Separating Static and Dynamic Web Content

This tip is a little dated in its details, but the premise remains true. It describes how to separate static content from a WebSphere application so it can be served by the Web server instead of the application server. Separating your static Web content (HTML, GIF, CSS files, and so on.) and dynamic Web content (servlets and JSP files) allows processing capacity to be split between the Web server and application server. You can then allocate capacity between the two based on the amount of dynamic and static content in your site. If your site serves mostly static content, it is more cost effective to add more Web servers than it is to add more application servers.

WebSphere best practices on developerWorks

The WebSphere Best Practices Zone provides a collection of best practices for administering WebSphere Application Server. Over time this zone will grow to include best practices for using other WebSphere software products, and to cover more topics.

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.

Resource reference list

At the URL <http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&uid=swg27005148>, WebSphere Application Server has a large amount of existing documentation. Use the following information as a guideline to find the documentation that you require.

User communities and other non-IBM sites

These sites gather knowledge about using WebSphere products.

- <http://www.websphere-world.com/>
- <http://www.websphere.org/>
- <http://www.webspherepro.com/wphome/>
- <http://www.sys-con.com/websphere/>
- <http://websphereadvisor.com/>

WebSphere Application Server V5.x best practices for configuration changes

This White paper at the URL http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&uid=swg27005391&loc=en_US&cs=utf-8&lang=en describes how to manually modify certain parts of the WebSphere Application Server Version 5.x configuration that are not available through the administrative tooling that comes with the product. Version 5.x configuration, for all editions of the product, is stored in XML files in a subdirectory under the main product installation root directory.

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.

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" in the information center 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 or Rational Web Developer that is shipped with WebSphere Application Server.
- 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

Use this section as a starting point to investigate the technologies used in and by applications that you deploy on the application server. Also use this section to learn about developing WebSphere applications.

See “Learn about WebSphere applications: Overview and new features” on page 2 for an introduction to each technology.

Web applications	How do I?...	Overview		Samples
EJB applications	How do I?...	Overview	Tutorials	Samples
Client applications	How do I?...	Overview		Samples
Web services	How do I?...	Overview	Tutorials	Samples
Data access resources	How do I?...	Overview	Tutorials	Samples
Messaging resources	How do I?...	Overview	Tutorials	Samples
Mail, URLs, and other J2EE resources	How do I?...	Overview		
Security	See the <i>Securing WebSphere applications</i> PDF	See the <i>Administering applications</i> PDF	See the <i>Securing WebSphere applications</i> PDF	See the <i>Securing WebSphere applications</i> PDF
Naming and directory	How do I?...	Overview		
Object Request Broker	How do I?...	Overview		
Transactions	How do I?...	Overview		Samples
ActivitySessions	How do I?...	Overview		Samples
Application profiling	How do I?...	Overview		Samples
Asynchronous beans	How do I?...	Overview		Samples
Dynamic caching	How do I?...	Overview		
Dynamic query	How do I?...	Overview		Samples
Internationalization	How do I?...	Overview		Samples
Object pools	How do I?...	Overview		
Scheduler	How do I?...	Overview		Samples
Startup beans	How do I?...	Overview		
Work areas	How do I?...	Overview		

Web applications

Learn about Web applications

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and “How do I?...” topics, pending their availability.

How do I?...

Deploy and administer Web applications

- Deploy and administer Web applications (same as any application type)
- Deploy applications (Education on Demand)
- Administer applications (Education on Demand)
- Troubleshoot Web application deployment
- Modify the default Web container configuration
- Configure session management
- Configure session management (Education on Demand)
- Disable run-time compilation of JSP files
- Tune session management

Secure Web applications

- Develop Web applications that use declarative security
- Use programmatic security when declarative security is not enough
- Secure Web applications during assembly

Develop Web applications

- Develop servlets that filter requests or responses
- Develop servlets that notify listeners of context or session changes
- Develop servlets using page lists to avoid hardcoding URLs
- Develop servlets that manage HTTP sessions
- Have the product automatically set encoding values and content types
- Share session data among various Web modules in the same application
- Develop JSP files with WebSphere extensions

Refer to the *Migrating, coexisting, and interoperating* PDF.

Migrate Web applications Assemble Web applications

- Configure Web application deployment descriptors
- Create Web application archive (WAR) files for deployment

Conceptual overviews

Documentation

Refer to the article *Introduction: Web applications* in the information center.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

• **Plants by WebSphere**

Using the Plants by WebSphere storefront, customers can open accounts, browse for items to purchase, view product details, and place orders. The Plants by WebSphere application uses container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, a stateful session bean, JSP pages, and servlets.

When the Greenhouse Supplier Sample application is installed and configured, an administrator can order additional inventory from the Greenhouse Supplier. See the Samples Gallery for more information on the Greenhouse Supplier application. The Greenhouse Supplier is used with Plants By WebSphere to demonstrate Web services.

- **WebSphere Bank**

Using the WebSphere Bank online bank, customers can open accounts, get account balances, and transfer funds between accounts. The WebSphere Bank application uses Web services, Java Message Service (JMS) API, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, Message-Driven Beans (MDB), JSP pages, and servlets.

- **Greenhouse by WebSphere**

Using the Greenhouse by WebSphere online supplier, customers can open accounts, select items and amounts to order, and check their order status. The Greenhouse by WebSphere application uses Web services, the Java message service (JMS) API, scheduler, asynchronous beans, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, message-driven beans (MDB), Java server pages (JSP)s, and the struts framework.

- **Java Adventure Builder**

The Adventure Builder customer Web site resides on the Web tier and is designed using a Web application architecture. This Web site communicates to the order processing backend module using Web services interactions. Adventure Builder, a basic Web site travel application built on the J2EE 1.4 platform, is a simple shopping application.

The customer can browse and select from a catalog of products, in this case vacation packages, assemble or build an entire vacation from different components, principally lodging and activities. The parts of a particular vacation package are determined by user responses given on a sequence of forms. You can maintain vacation package options in a virtual shopping cart, perform sign on and sign off procedures, create user accounts, and purchase a trip package, sending a purchase order to the order fulfillment system. The Adventure Builder application uses several J2EE 1.4 technologies.

- **Simple Servlet - Greeting**

A greeting with connection information displays when Simple Servlet runs. A servlet and JSP page implement this greeting.

- **PageList Servlet - Page Returner**

PageList, an IBM WebSphere Application Server extension of the servlet API, coordinates some typical servlet functions, such as forwarding pages, and hides details of their implementation from the user. The PageList Sample enables users to choose a page to return, and automatically forwards the specified page from the servlet.

- **Filter Servlet - Output Trail**

Servlet filters can pre-process and post-process request data and information. Text is displayed in the HTML browser by the simple Filter Sample, before and after the servlet is processed.

- **JSP - Calendar Creator**

Select attributes and create a calendar using the most up to date functionality available with JavaServer Pages technology.

- **Simple JSP - Date**

A greeting with the current date and time displays when this Sample runs. This greeting implements in a JSP page.

- **Tag Library - Random Quote**

Tag libraries consolidate coding functionality into easy-to-use markup tags in JSP pages. This Sample generates a random number, accesses the corresponding element in a resource bundle, and displays a quote.

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.
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 "Resources for learning" article for links to design documentation.
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 the application is accessed.

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.

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

The IBM 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 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.

The URL for accessing Samples is: <http://localhost:9080/WSsamples/>

Snoop

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 execute the servlet.

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

HelloHTML

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`.

HitCount

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 6.0 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 engine:

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

WebSphere Application Server 6.0 supports the JSP 2.0 specification.

The JSP engine

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

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

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

- **compileWithAssert**

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

- **classdebuginfo**

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

- **deprecation**

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

- **disableJspRuntimeCompilation**

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

- **extendedDocumentRoot**

To allow a JSP file resource to be shared across Web application archives, specify a comma delimited list of directories and/or Java Archive (JAR) files as search paths to be used if the requested resource cannot be located in the Web application archive's public document tree. If the JSP file is located inside a JAR file and `reloadEnabled` is true, the timestamp of the JAR file is used for `isOutDated` checks for recompile purposes. The default for this parameter is `null`.

- **ieClassID**

Indicates the Java plug-in COM class ID for Internet Explorer. The `<jsp:plugin>` tags use this value. The default classid is `clsid:8AD9C840-044E-11D1-B3E9-00805F499D93`. This parameter requires regeneration of Java source.

- **javaEncoding**

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

- **jspCompileClasspath**

This parameter tells the JSP engine to use a small class path for the Java compilation phase. The small class path speeds up the compilation process. This small class path is not used by default because it includes only a subset of WebSphere JAR files and excludes many WebSphere JAR files that contain WebSphere public APIs.

If your JSP files do not use WebSphere public APIs within scriptlets, you can enable the small class path by using the `jspCompileClasspath` parameter with no value. If your JSP files do use WebSphere public APIs within scriptlets, then add those additional JAR files to the `jspCompileClasspath` option. The entries are separated by spaces, and are assumed to be relative to the WebSphere Application Server installation root.

1. A space-separated list of JAR files, relative to the WebSphere installation directory. This instructs the JSP engine to use the small class path, with the additional named JAR files.

2. No value at all. This instructs the JSP engine to use the small class path, without any additional user-defined JAR files. See the example in "Configuring JSP engine parameters" in the information center.

The entire WebSphere class path is used by default. This parameter requires regeneration of Java source.

- **jsp.file.extensions**

For JSP files with extensions other than the four standard extensions, `*.jsp`, `*.jspx`, `*.jsw`, and `*.jsv`, you can configure these extensions using this parameter. These extensions are added to the standard extensions. The preferred method for doing this is to create a `<jsp-property-group>` in `web.xml`, and add a `<url-pattern>` tag for each extension.

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

- **keepgenerated**

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

- **keepGeneratedclassfiles**

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

- **reloadEnabled**

Determines whether or not a JSP file is translated and compiled at runtime if the JSP file or its dependencies (see `trackDependencies`) are modified. If `reloadEnabled` is `false`, a JSP file is still compiled, if necessary, on the first request to it unless the parameter `disableJspRuntimeCompilation` is `true`. The default for this parameter is `false`.

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

- **reloadInterval**

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

- **scratchdir**

Specifies the directory where the generated class files are created. The system property `com.ibm.websphere.servlet.temp.dir` is used to set the `scratchdir` option on a server-wide basis. The JSP engine `scratchdir` parameter takes precedence over the system property `com.ibm.websphere.servlet.temp.dir`. The default for this parameter is `{WAS_ROOT}/profiles/profilename/temp`. This parameter requires regeneration of Java source.

- **trackDependencies**

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

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

The default is `false`.

- **useFullPackageNames**

If `useFullPackageNames` is `true`, the JSP engine generates and loads JSP classes using full package names. The default is to generate all JSP classes in the same package. (For more information, see `Packages and directories for generated files`). The JSP engine's class loader knows how to load JSP classes when they are all in the same package.

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

When `useFullPackageNames` is set to `true`, the batch compiler generates a file called `generated_web.xml` in the Web module's `WEB-INF` directory. This file contains servlet configuration

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

- **useImplicitTagLibs**

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

- **useJikes**

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

- **usePageTagPool**

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

- **useThreadTagPool**

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

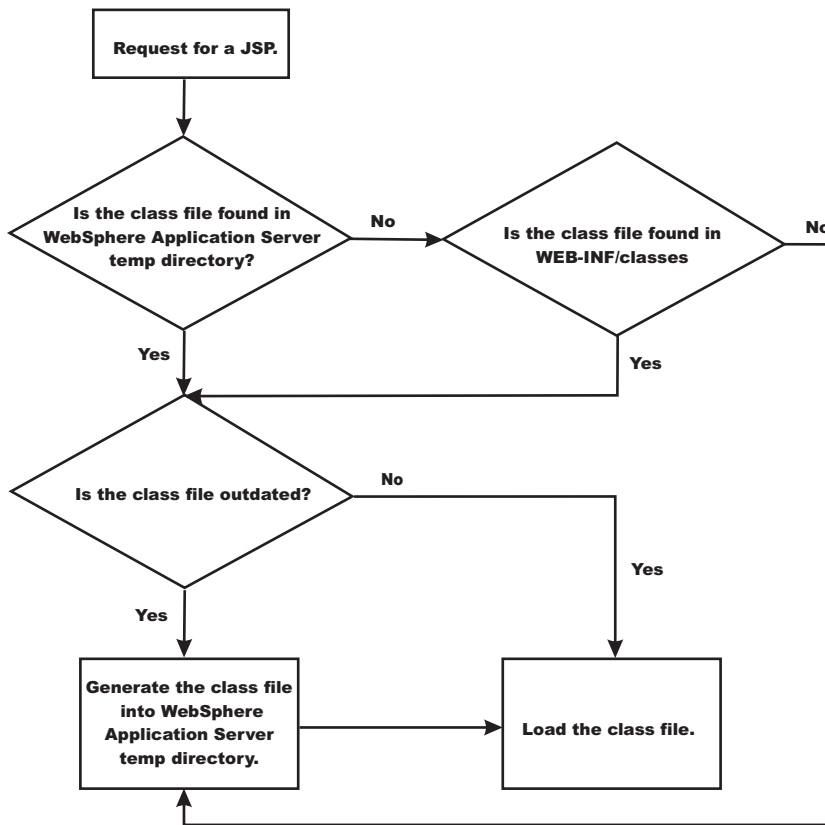
- **verbose**

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

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

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

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_ROOT}/profiles/profilename/temp`. 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.



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
<i>compileToDir</i> not supplied; <i>compileToWebInf</i> 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.
<i>compileToDir</i> not supplied; <i>compileToWebInf</i> is false	The class files are compiled into the Web module's WEB-INF/classes directory.	The class files are compiled into the WebSphere temp directory, usually {WAS_ROOT}/profiles/profilename/temp.
<i>compileToDir</i> is supplied; <i>compileToWebInf</i> not supplied, or is either true or false	The class files are compiled into the directory indicated by <i>compileToDir</i> .	The class files are compiled into the directory indicated by <i>compileToDir</i> .

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 30 for more information. The table below gives examples of packages and directory structures for generated .java and .class files.

JSP file	Java package		Location of .java or .class files in file system	
	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.

```
/*
C:/WebSphere_6.0/AppServer/profiles/AppSrv01/installedApps/MyCell/sampleApp.ear/examples.war
/WEB-INF/classes/_ibmjsp/_jsp1.java was generated @ Thu Oct 14 10:05:56 EDT 2004
IBM WebSphere Application Server - ND, 6.0.0.0
  Build Number: o0441.04
  Build Date: 10/12/04
```

```
*****
The JSP engine configuration parameters were set as follows:
```

```
classDebugInfo = [false]
debugEnabled = [false]
deprecation = [false]
compileWithAssert = [false]
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]
jspCompileClasspath = []
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 runtime reloading 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
<code><jsp-file></code>	<pre> <servlet> <servlet-name>jspOne</servlet-name> <jsp-file>jspOne.jsp</jsp-file> </servlet> </pre>	Yes	Yes	Can be true or false
<code><servlet-class></code>	<pre> <servlet> <servlet-name>jspTwo</servlet-name> <servlet-class>_ibmjsp.jspTwo</servlet-class> </servlet> </pre>	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. In WebSphere Application Server 6.0, 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.

1. An example of a configuration repository directory is
`{WAS_ROOT}/profiles/profilename/config/cells/cellname/applications/enterpriseappname/deployments/deployedname/webmodulename`
2. 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 runtime reloading: JSP files can be translated and compiled at runtime 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 JSPs 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 runtime 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 be 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 runtime or is processed by the batch compiler.

Disabling compilation

Disablement of runtime 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 runtime does not translate and compile JSP files; the JSP engine loads only precompiled class files. JSP source files do not need to be present in order 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.

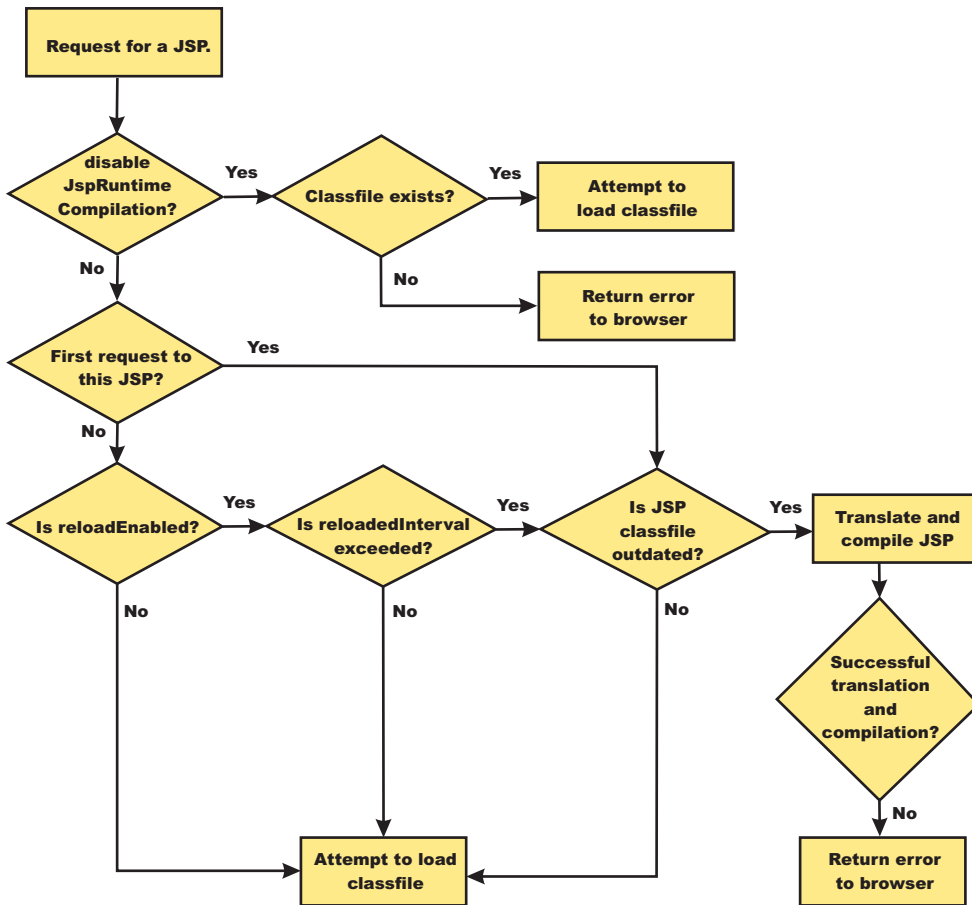


Figure 1. Reload processing sequence when **trackDependencies** is false.

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.

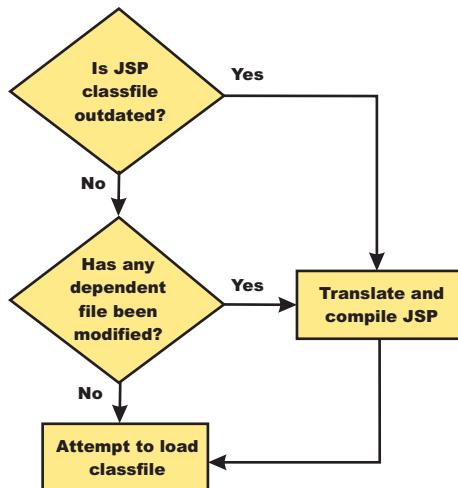


Figure 2. Additional reload processing performed when **trackDependencies** is true.

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 6, "Assembling applications," on page 1005.

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.

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.

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 server 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 .jspxf 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 (described in the topic Batch Compiler Ant Task) is also available for executing the batch compiler using Ant.

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>]
[-jspCompileClasspath <classpath to Websphere Application Server public API JAR files;
    or no value at all>]
[-verbose <true | false>]
[-deprecation <true | false>]
[-javaEncoding <encoding>]
[-compileWithAssert <true | false>]
[-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” on page 24.
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 runtime, 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:

where:

- **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. A template response file, `batchcompiler.properties.default`, is found in `{WAS_ROOT}/bin`. 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.

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 will be 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 compiled. 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 WebSphere 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.[bat/sh] in {WAS_ROOT}/bin. 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 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 runtime 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 runtime 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 runtime, these same JAR files and classes have to be made available through the standard WebSphere Application Server configuration mechanisms.

Default: null

- **jspCompileClasspath**

This option instructs the batch compiler to use a small class path for the Java compilation phase. The small class path greatly speeds up the compilation process. This small class path is not used by default because it includes only a subset of WebSphere Application Server JAR files. The small class path excludes many WebSphere Application Server JAR files, among which are those that contain WebSphere public APIs.

If your JSP pages do not use any WebSphere public APIs within scriptlets you can enable the small class path by using the `jspCompileClasspath` parameter with no value, as in Example 1 below.

If your JSP pages do use WebSphere public APIs within scriptlets, then add those additional JAR files to the `jspCompileClasspath` option, as in Example 2 below.

The entries are separated by spaces, and are assumed to be relative to the WebSphere Application Server installation root.

Example 1: If no public APIs are needed in JSP pages: `JspBatchCompiler -enterpriseApp.name sampleApp -jspCompileClasspath`

Example 2: public APIs from the `admin.jar` file needed in JSP pages: `JspBatchCompiler -enterpriseApp.name sampleApp -jspCompileClasspath "lib/admin.jar"`

Use-case: Use this parameter to speed up the Java compilation step of JSP page processing.

Default: The entire WebSphere Application Server class path is used by default.

- **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.

- **compileWithAssert**

Tells the batch compiler to enable assertions. If `compileWithAssert` is true, the batch compiler will pass the `-source 1.4` option to the `javac` compiler. If `compileWithAssert` is false, no option is sent to the `javac` compiler. The default behavior of `javac` is to compile code normally even if the word `assert` is used as regular identifier.

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

Use-case: Set this parameter to true when you want you use the assertion facility in your JSP pages and you want to be able to turn on assertions at runtime.

Default value: false

- **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 the section, "JSP batch compiler tool" on page 36, 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 compiler parameter
earPath	-ear.path

warPath	-war.path
src	-war.path
Same as warPath, for backward compatibility	
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	-compileToDir -compileToDir
same as compileToDir, for backward compatibility	
compileToWebInf	-compileToWebInf
jspCompileClasspath	-jspCompileClasspath
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	-additional.classpath
same as additionalClasspath, for backward compatibility	
verbose	-verbose
deprecation	-deprecation
javaEncoding	-javaEncoding
compileWithAssert	-compileWithAssert
useJikes	-useJikes
jspFileExtensions	-jsp.file.extensions

logLevel	-log.level
wasHome	none
Classpathref	none

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:

```
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="wsjpc" 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">
    <wsjpc wasHome="${was.home}"
      earpath="${ear.path}"
      forcecompilation="true"
      extractToDir="${extract.dir}"
      useThreadTagPool="true"
      keepgenerated="true"
      jspCompileClasspath=""
    />
  </target>
  <target name="jspc2" description="example using an enterprise app and webmodule">
    <wsjpc wasHome="${was.home}"
      enterpriseAppName="${app.name}"
      webmoduleName="${webmodule.name}"
      removeTempDir="true"
      forcecompilation="true"
      keepgenerated="true"
      jspCompileClasspath=""
    />
  </target>
  <target name="jspc3" description="example using an enterprise app, webmodule and specific directory">
    <wsjpc wasHome="${was.home}"
      enterpriseAppName="${app.name}"
      webmoduleName="${webmodule.name}"
      fileName="${dir.name}"
      recurse="false"
      forcecompilation="true"
      keepgenerated="true"
      jspCompileClasspath=""
    />
  </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.

Location added to class path	Batch compiler target		
	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 <code>-extractToDir</code> is not used; otherwise, no.
Shared libraries	yes	no	no
Web module JAR files and classes	yes	yes	yes
<code>additional.classpath</code> parameter to batch compiler	no	no	yes
<code>jspCompileClassPath</code> parameter	When this parameter is used, the only change to the information above is that a subset of WebSphere Application Server JAR files and classes is used for Java compilation. All JAR files and classes, that are given in the value for the <code>jspCompileClassPath</code> parameter are also added to the class path for Java compilation.		

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: 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.

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**

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

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

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

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: 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.

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

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

```


Example: tsx:dbmodify tag syntax: 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: 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
<p>myBean.class</p> <pre>// Code to get a connection // Code to get the data Select * from myTable; // Code to close the connection</pre> <p>JSP file</p> <pre><tsx:repeat index=abc> <tsx:getProperty name="myBean" property="coll(abc)" /> </tsx:repeat></pre> <p>Notes:</p> <ul style="list-style-type: none"> • The bean (myBean.class) is a static bean. • The method to access the bean properties is myBean.get(<i>property(index)</i>). • 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. 	<p>JSP file</p> <pre><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="coll(abc)" /> </tsx:repeat></pre> <p>Notes:</p> <ul style="list-style-type: none"> • 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(<i>"property", index</i>). • 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>
```

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 or 6.

Symptom Your Web module does not run when you migrate it to Version 5 or 6

Problem In Version 4.x, the classpath setting that affected visibility was *Module Visibility Mode*. In Versions 5 and 6, 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.

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 82 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.

View links to additional information about:

- “Web applications: Resources for learning” on page 56
- “Web applications: Resources for learning” on page 56
- “Web applications: Resources for learning” on page 56

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

- Redbook on Servlet and JSP file Programming
- 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

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.

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.

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 <code>anonymous</code>	A new session is created. The user name is <code>FRED</code>
A session ID for a valid session is passed in. The current session user name is <code>"anonymous"</code>	The session is returned.	The session is returned. session management changes the user name to <code>FRED</code>
A session ID for a valid session is passed in. The current session user name is <code>FRED</code>	The session is not returned. An <code>UnauthorizedSessionRequestException</code> error is created*	The session is returned.

	Unauthenticated HTTP request is used to retrieve a session	HTTP request is authenticated, with an identity of "FRED" used to retrieve a session
A session ID for a valid session is passed in. The current session user name is BOB	The session is not returned. An <code>UnauthorizedSessionRequestException</code> error is created*	The session is not returned. An <code>UnauthorizedSessionRequestException</code> 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 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. 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.

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.

Tuning session management

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_nameWeb 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

- **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.**

This action allows the object to properly serialize when using distributed sessions. Without this extension, the object cannot serialize correctly and throws an error. 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.

- **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 user 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.)

Managing 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

- Best practices
- HTTP Session Persistence Best Practices
- 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 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 Javadoc for the application server APIs in the product `install_root\web\apidocs` directory.

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 lifecycle 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 (in this case, 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.

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

```

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

```

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

Filter, FilterChain, FilterConfig classes for servlet filtering

The following interfaces are defined as part of the `javax.servlet` package:

- `Filter` interface - methods: `doFilter()`, `getFilterConfig()`, `setFilterConfig()`
- `FilterChain` interface - methods: `doFilter()`
- `FilterConfig` interface - methods: `getFilterName()`, `getInitParameter()`, `getInitParameterNames()`, `getServletContext()`

The following classes are defined as part of the `javax.servlet.http` package:

- HttpServletRequestWrapper - methods: See the Servlet 2.4 Specification
- HttpServletResponseWrapper - methods: See the Servlet 2.4 Specification

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

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 mimeType, String pageName, HttpServletRequest request, HttpServletResponse response)`

where the method arguments are:

mimeType 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.

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.

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.

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 `PageListServlet` code example to see how the `callPage()` method is invoked by a servlet.

In the example, the client type detection method, `getMLTypeFromRequest(HttpServletRequest request)`, 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, extending `PageListServlet`, 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: 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(m1Name, 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 `autoResponseEncoding` 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 `<install_root>\lib` directory.

For example, to compile a servlet running on the Windows NT version of WebSphere Application Server, specify:

```
javac -classpath D:\Program Files\WebSphere\AppServer\lib\j2ee.jar MyServlet.java
```

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 6.0 supports JavaServer Faces 1.0 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.

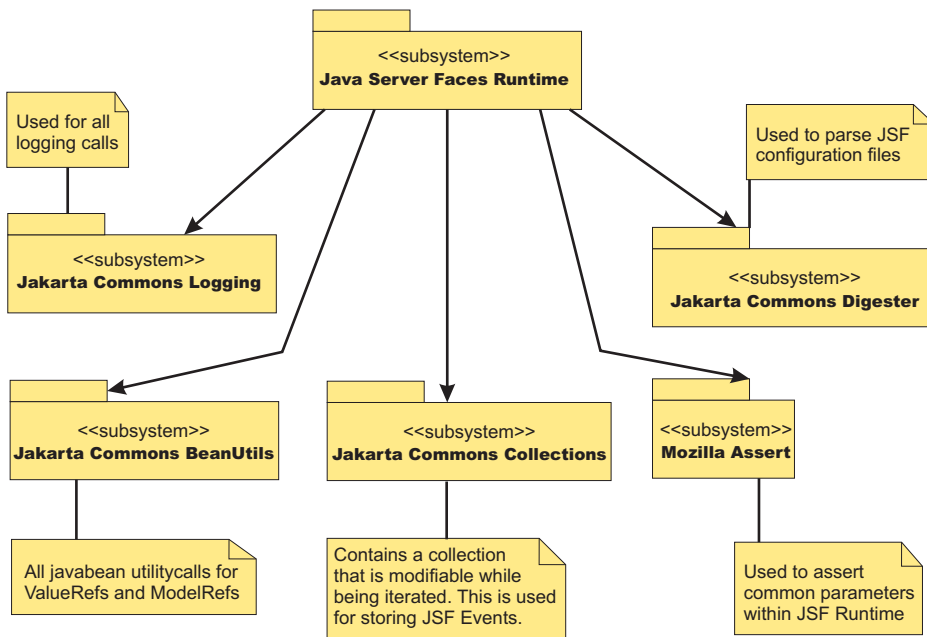
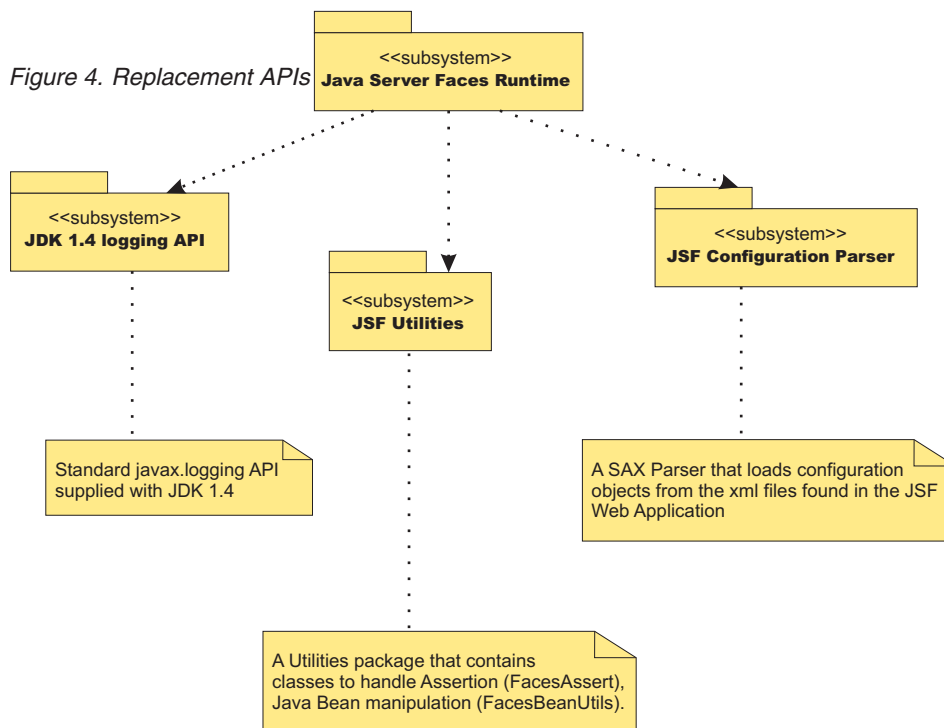


Figure 3. Current external API dependencies from the Sun based JSF runtime



The specification related classes (javax.faces.*) for JSF is packaged in a JAR file called jsf-api.jar. The IBM modified version of the JSF Sun reference implementation is packaged in a JAR file called ws-jsf.jar. The dependency on JavaServer Pages Standard Tag Library (JSTL) requires that its JAR files, (jstl.jar and standard.jar), are available, therefore they need to be available in the Application Server runtime. The

jstl.jar and standard.jar files reside in the lib directory of the WebSphere Application Server install image. You must copy the JSF JAR files to the standard Application Server runtime directory.

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 6.0. 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 6.0 version.

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 FacesAssert 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

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 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 article 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 Web 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 AST or Rational Web Developer 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.

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 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).

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 used for specifying supplied to the server component that allows serving of servlets by name or class name

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.

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 the assembly tool.

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.

EJB applications

Learn about EJB applications

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Deploy and administer EJB applications

- Deploy EJB applications
- Deploy applications (Education on Demand)
- Administer applications (Education on Demand)
- Modify the default EJB container configuration

Secure EJB applications

- Develop EJB applications that use declarative security
- Use programmatic security, when declarative security is not enough
- Secure EJB applications during assembly
- Migrate EJB application components

Assemble EJB applications

- Choose an access intent policy for EJB 2.0.x applications
- Troubleshoot access intent
- Assemble EJB applications for deployment

Conceptual overviews

Documentation

- "Enterprise beans" on page 88
- See Chapter 9 of the IBM Redbook EJB 2.0 Development with WebSphere Application Studio Developer (sg246819)

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

developerWorks offers these tutorials that accompanied the WebSphere Application Server Technology for Developers Version 6 release. They provide a solid understanding of the J2EE technologies.

- **Tutorial 1 - JSP, Servlets, EJB**

This tutorial provides an understanding on the EJB Query Language, JSP Expression Language, building your own custom tags, and also about the new deployment descriptors which uses the XML Schema instead of the DTD. The zip file comes with all sample code required to run this tutorial.

- **Tutorial 2- EJB Timer**

This tutorial provides an understanding on the EJB Query Language, JSP Expression Language, building your own custom tags, and also about the new deployment descriptors which uses the XML Schema instead of the DTD. The zip file comes with all sample code required to run this tutorial.

- **Tutorial 3- Message Driven Timer**

This tutorial makes full use of the MyBank sample codes. The sample consists of 2 entity beans, CustomerBean and AccountBean, whose abstract schema types are Customer and Account, respectively. Each entity beans has remote/local interfaces and remote/local home interfaces. The entity bean CustomerBean has one-to-many relationships with AccountBean. The SenderBean.java session bean is responsible of sending message to the destination, and the MDB MyBankListenerBean.java is the consumer for the message. The zip file comes with all sample code required to run this tutorial.

Samples

The Samples Gallery offers:

- **Plants by WebSphere**

Using the Plants by WebSphere storefront, customers can open accounts, browse for items to purchase, view product details, and place orders. The Plants by WebSphere application uses container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, a stateful session bean, JSP pages, and servlets.

When the Greenhouse Supplier Sample application is installed and configured, an administrator can order additional inventory from the Greenhouse Supplier. See the Samples Gallery for more information on the Greenhouse Supplier application. The Greenhouse Supplier is used with Plants By WebSphere to demonstrate Web services.

- **WebSphere Bank**

Using the WebSphere Bank online bank, customers can open accounts, get account balances, and transfer funds between accounts. The WebSphere Bank application uses Web services, Java Message Service (JMS) API, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, Message-Driven Beans (MDB), JSP pages, and servlets.

- **Greenhouse by WebSphere**

Using the Greenhouse by WebSphere online supplier, customers can open accounts, select items and amounts to order, and check their order status. The Greenhouse by WebSphere application uses Web services, the Java message service (JMS) API, scheduler, asynchronous beans, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, message-driven beans (MDB), Java server pages (JSP) files, and the struts framework.

- **BMP - Address Book**

A basic address book application that creates, updates, finds, and removes address book entries. The Address Book application uses Bean Managed Persistence (BMP).

- **CMP 1.1 - Movie Review**

A movie review application that creates and finds movie reviews. This Movie Review application uses Container-Managed Persistence (EJB 1.1 CMP).

- **CMP 2.1 - Movie Review**

A movie review application that creates and finds movie reviews. This Movie Review application uses Container-Managed Persistence (EJB 2.1 CMP).

- **CMR - Subscription**

A subscription service, where you can enter an e-mail address and select programming topics. This application uses Container-Managed Relationships (EJB 2.0 CMR).

- **Stateful Session - Reading List**

View library-like resources. Add and remove resources from reading lists, using a stateful session bean.

- **Stateless Session - Basic Calculator**

A basic calculator application that performs addition, subtraction, multiplication and division, using a stateless session bean

- **Timer Service - Bulletin Board**

A bulletin board application, where you can enter a message and post it for a specified length of time. This application uses the EJB Timer Service.

Task overview: Using enterprise beans in applications

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 420 and “Messaging: Resources for learning” on page 614.
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 the assembly tool. 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 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 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 "Resources for learning."

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 "Resources for learning."

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.

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" in the information center.

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 "Resources for learning."

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.

View links to additional information about:

- Planning, business scenarios, and IT architecture
- Programming model and decisions
- Programming instructions and examples
- Programming specifications

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

- Rules and Patterns for Session Facades
EJB programming practice: Fronting entity beans with a session-bean facade.
- 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.
- Java™ 2 Platform: Compatibility with Previous Releases

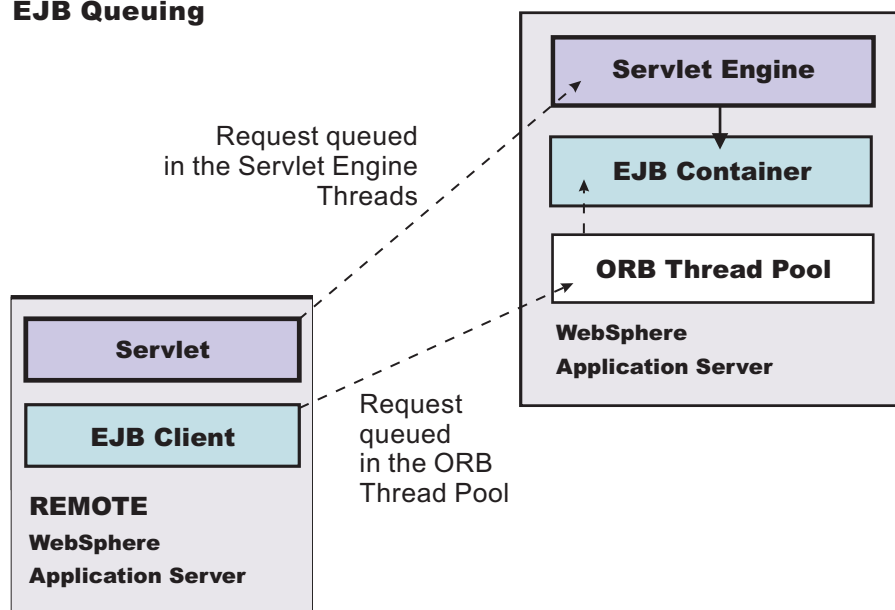
This Sun Microsystems article includes both source and binary compatibility issues.

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



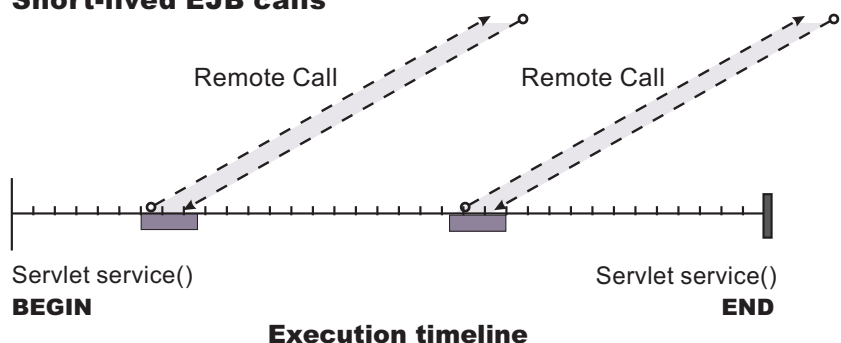
The following are two tips for queueing enterprise beans:

- **Analyze the calling patterns of the EJB client.**

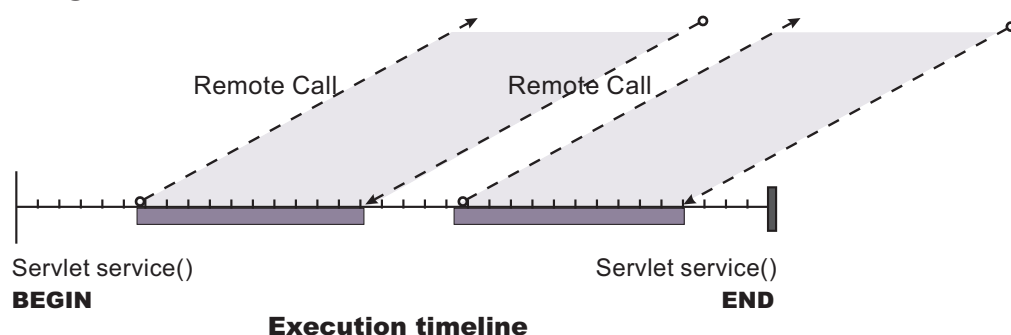
When configuring the thread pool, it is important to understand the calling patterns of the EJB client. If a servlet is making a small number of calls to remote enterprise beans and each method call is relatively quick, consider setting the number of threads in the ORB thread pool to a value lower than the Web

container thread pool size value.

Short-lived EJB calls



Longer-lived EJB calls



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 at the ORB. Increase the number of threads.

See also "Queuing network" in the information center.

Developing enterprise beans

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

This function is an addition to the existing EJB caching options. 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:
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 accesses this data in a read-only fashion. 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 `findByMethod` 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.

For detailed information about source and binary compatibility between deployed versions, see "Resources for learning."

3. 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 `findByMethod` 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 `setField(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 spec 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.

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 114 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 125 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.

Batch 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 can be only one database round trip for all the batched persistence requests.

The WebSphere Application Server versions 5.0.2 and later enable you to take advantage of this. You can turn this option on from the EJB CMP side. When you choose this option, 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 version 6.0 enables you to make the same settings using the Application Server Toolkit (AST).

Setting 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:

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 **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**.

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 version 6.0 enables you to make the same settings using the Application Server Toolkit (AST).

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 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).

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:

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**.

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:

Usage Scenario

The scenario of use for this feature begins with configuring one or more bean types to be long-lifetime beans (see Explicit Invalidation in the Persistence Manager Cache, and configuring the necessary Java Message Service (JMS) resources (described below). Once this is done, the server is started. 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*.
- 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" in the information center).

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):

```
package com.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 PMCacheInvalidationRequest implements Serializable {
    . . .

    /**
     * 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 choose whichever JMS provider they prefer (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 yourself. 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 Persistence Manager Cache invalidation:

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.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

1. Open the administrative console.
2. Click **Servers > Application Servers > servername > EJB Container Settings > EJB timer service settings**. The Timer Service settings panel appears.
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 106

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 wish 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 977 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 a couple of ways to configure the Timer Service to share the same database across multiple servers. Steps 1 and 2 below are mutually exclusive. Choose one or the other.

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 106
 - 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.TimerService;
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 = theTimerService.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	<i>jdbc/DefaultEJBTimerDataSource</i>

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	<i>EJBTIMER_</i>

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

Timer service for Enterprise JavaBeans 2.1

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 104.

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.

Enterprise bean development:

1. Write your enterprise bean to implement the *javax.ejb.TimedObject* interface, including the *ejbTimeout()* method.
2. The bean calls the *EJBContext.getTimerService()* method to get an instance of the **TimerService** object.
3. The bean calls the *TimerService* method to create a Timer. This Timer is now associated with that bean.
4. After you create it, the Timer instance can be passed 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 “Editing deployment descriptors” on page 1020 and “EJB Timer Service settings” on page 106 for more information.

Clustered environment considerations: In a single server environment, there is no question 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 104 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” in the information center.

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 111.

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 112.

Example: FindEJBTimers command: 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 be 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: 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

The 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 a web service. You 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 pre-existing 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 351.

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 Assembly Service Toolkit.

Defining data sources for entity beans

Create a data source or JDBC resource and give it a Java Naming and Directory Interface (JNDI) name.

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. For more information, see "Application bindings" on page 1062

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.

Using access intent policies

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.

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. Nullable columns are automatically excluded from overqualified updates at deployment time; concurrent changes to a nullable field might result in lost updates. When used with the IBM Rational Application Developer product, this product provides support for selecting a subset of the non-nullable columns that are to be reflected in the overqualified update statement that is generated in the deployment code to support optimistic policies.

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.

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 are most likely to be needed next by an application. A *read-ahead hint* is a representation of the related beans that are to be read. It 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 CMRs 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 their 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*), a bean's data columns appear only once, at the position it first appears 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 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 a relationship *employees* with the Employee bean and also has a relationship *manager* 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.

Some things to consider

When developing your read-ahead hints, you should keep the following in mind:

- Read ahead on long or complex paths can result in a query that is too complex to be useful. Read ahead on root or leaf inheritance mappings need particular care. You should add up the number of tables that are involved in the preload and then consider whether a join that complex is really a reasonable query on your target database.
- Read ahead does NOT work in the following cases:
 - preload paths across M:N relationships
 - preload paths across recursive enterprise bean relationships or recursive fk relationships
 - where multiple instances of the same table occur on the same path (whether through a recursive relationship or not).
 - when readAhead 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.

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 has not changed since the last load (with proper locking based on access intent's concurrency control attribute.)

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: Usage Scenario

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.

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.
- User1 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.
- User3 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.
- User1 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.
- User3 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.
- User1 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 run-time 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.

Although it is recommended that you always configure bean level access intent for your applications, if you find it necessary you can apply access intent policies to methods within the scope of an EJB module. In such cases the policy becomes the default access intent for all requests upon the configured methods.

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.

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 tools.

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

Access intent exceptions

The following exceptions are thrown in response to the application of access intent policies:

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 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

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.

Additional information about valid settings follows:

Profile name	Concurrency control	Access type	Transaction isolation
<code>wsPessimisticRead</code> (Note 1)	pessimistic	read	For Oracle, read committed. Otherwise, repeatable read
<code>wsPessimisticUpdate</code> (Note 2)	pessimistic	update	For Oracle, read committed. Otherwise, repeatable read
<code>wsPessimisticUpdate-Exclusive</code> (Note 3)	pessimistic	update	serializable
<code>wsPessimisticUpdate-NoCollision</code> (Note 4)	pessimistic	update	read committed
<code>wsPessimisticUpdate-WeakestLockAtLoad</code> (Note 5)	pessimistic	update	Repeatable read
<code>wsOptimisticRead</code>	optimistic	read	read committed
<code>wsOptimisticUpdate</code> (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. Generated overqualified-update query forces failure if CMP column values have changed since the beginning of the transaction.

Access intent best practices

This topic outlines issues to consider when applying access intent policies to Enterprise JavaBeans (EJB) methods.

- **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

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 databases, the weakest lock is a shared one, and the query runs without a `USE AND KEEP UPDATE LOCKS` clause. On Oracle databases, however, the weakest lock is an update lock; this means that the SQL query must contain a `USE AND KEEP UPDATE LOCKS` clause. However, not every SQL statement necessarily supports `USE AND KEEP UPDATE LOCKS`; for example, if the query is being run against multiple tables in a join, `USE AND KEEP UPDATE LOCKS` is not supported. To avoid this problem, try either of the following:

- Modify your SQL query or reconfigure your application so that an update lock is supported
- 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 article 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.

To increase performance, break container-managed persistence (CMP) enterprise beans into several enterprise bean modules during assembly. The load time for hundreds of beans is improved by distributing the beans across several JAR files and packaging them to an EAR file. Load time is faster when the administrative server attempts to start the beans, for example, 8-10 minutes versus more than one hour when one JAR file is used.

Use an assembly tool such as the Application Server Toolkit (AST) or Rational Web 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.
1. Start an assembly tool.
 2. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that the **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.

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.

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” on page 125 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 the assembly tool.
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.
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.

EJB module collection

Use this page to manage the EJB modules deployed in a specific application.

To view this administrative console page, click **Applications > Enterprise Applications > *applicationName* > EJB modules**. Click the check boxes to select one or more of the EJB modules in your collection.

Remove: Removes a module from the deployed application. The module is deleted from the application in the WebSphere Application Server configuration repository and also from all 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 then 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 then the application is restarted.

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 then the application is restarted.

URI:

When resolved relative to the application URL, this specifies the location of the module's archive contents on a file system. The URI matches the `<ejb>` or `<web>` tag in the `<module>` tag of the application deployment descriptor.

There are three buttons on this panel.

Remove

removes the EJB Module

Remove File

removes the specified file from the EJB Module

Update

updates the module or the application. With Update, you can add, remove, or replace modules

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* > EJB modules > *moduleName***.

URI:

When resolved relative to the application URL, this specifies the location of the module archive contents on a file system. The URI must match the URI of a ModuleRef URI in the deployment descriptor of the deployed application (EAR).

Alternate DD:

Specifies a deployment descriptor to be used at run time instead of the one installed in the module.

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

Client applications

Learn about client applications

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...**Deploy and administer client applications**

- Deploy client applications on Windows operating systems
- Run J2EE client applications
- Troubleshoot client applications

Secure client applications

Refer to the *Securing applications and their environment* PDF.

Develop client applications - basics

- Decide on a type of client application
- Develop J2EE client applications
- Develop pluggable client applications
- Develop thin client applications
- Develop ActiveX client applications
- Troubleshoot client applications (refer to the *Troubleshooting and support PDF*)

Develop client applications - additional aspects

- Obtain internationalization contexts in clients
- Use the internationalization service
- Involve your clients in transactions
- Use the name space
- Pass client information to a database
- Have J2EE client applications access databases directly

Develop other kinds of clients

- Develop Web services clients
- Develop JMS clients

Migrate client applications

- Migrate client applications

Assemble client applications

- Assemble client applications
- Configuring resources for client applications
- Configuring database resources, in particular

Conceptual overviews**Documentation**

Refer to the article *Introduction: Client modules* in the information center.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery and the Client Samples Gallery offer the following client related Samples. Their availability varies by operating system. In general, Applet clients and ActiveX to EJB Bridge clients are not available on Unix and Linux operating systems.

- **Basic Calculator Client**

A basic calculator application that performs addition, subtraction, multiplication and division, using the BasicCalculator stateless session bean.

- **Client Samples in the Samples Gallery**

- Plants by WebSphere Catalog Manager, as:

- a J2EE application client
- an ActiveX to EJB Bridge client

- Client Technology Samples

- J2EE application client
 - Basic Calculator Client
 - Class Loader
 - Launch API

- Java thin client

- Basic Calculator Client

- Applet client

- Basic Calculator Client

- ActiveX to EJB Bridge client

- Basic Calculator Client

- ActiveX to EJB Bridge

- XJB Examples

- **Client Samples in the Client Samples Gallery**

The Client Samples Gallery demonstrates the following clients.

- J2EE application client
- Java thin client
- Applet client
- ActiveX to EJB Bridge client

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.

View the WebSphere Application Server Clients Samples Gallery for more information. To access these samples, install WebSphere Application Server Clients, and retrieve the samples from your local file system as the following command indicates:

```
<install_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 run-time environment.

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

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.
- An ActiveX to EJB Bridge run time for ActiveX to EJB Bridge application client applications (only for 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: 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

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 Web 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 exist 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.

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, 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 run-time 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 section provides some 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	<ul style="list-style-type: none">• 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 -CClasspath 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 -CClasspath 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 UNIX platforms) 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.getAddress(), InetAddress.getByAddress[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 bootstrap 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

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.

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

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).

This task only applies to J2EE application clients.

1. Open a command window and invoke the following script to launch J2EE application clients using the `launchClient` shell:

```
install_root/bin/launchClient.bat
```

The `launchClient` batch command starts the application client run time, which:

- Initializes the client run time.
- Loads the class that you designated as the main class with an assembly tool.
- Runs the main method of the application client program.

When your program terminates, the application client run time cleans up the environment and the Java virtual machine (JVM) code ends.

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 (shown here on 2 lines for publication):

```
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 section describes the Java 2 Platform Enterprise Edition (J2EE) command line syntax for the launchClient tool for WebSphere Application Server.

The following example illustrates the command line invocation syntax for the launchClient tool (shown here on 2 lines for publication):

```
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:

-CCsoapConnectorPort

The Simple Object Access Protocol (SOAP) connector port. If you do not specify this argument, the WebSphere Application Server default value is used.

-CCverbose

This option displays additional information messages. The default is `false`.

-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.

-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.

-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.

-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.

-CCinitonly

Use this option to initialize application client run time for ActiveX application clients without launching the client application. The default is `false`.

-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" in the *Troubleshooting and support* PDF.

-CCtracefile

Indicates the name of the file to which trace information is written. The default is to write output to the console.

-CCpropfile

Indicates the name of a properties file that contains `launchClient` properties. Specify the properties without the `-CC` prefix in the file. For example: `verbose=true`.

-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 `<install_root>/properties/client.policy`.

-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>`.

-CCexitVM

Use this option to have the WebSphere Application Server call the `System.exit()` method after the client application completes. The default is `false`.

-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`.

-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, read the topic "Interpreting trace output" in the *Troubleshooting and support PDF*.

-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.

The following examples demonstrate correct syntax.

On the Windows operating system:

```
launchClient c:\earfiles\myapp.ear -CCBootstrapHost=myWASServer -CCverbose=true  
app_parm1 app_parm2
```

On the UNIX operating system:

```
./launchClient.sh /usr/earfiles/myapp.ear -CCBootstrapHost=myWASServer -CCverbose=true  
app_parm1 app_parm2
```

Specifying the directory for an expanded EAR file:

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:

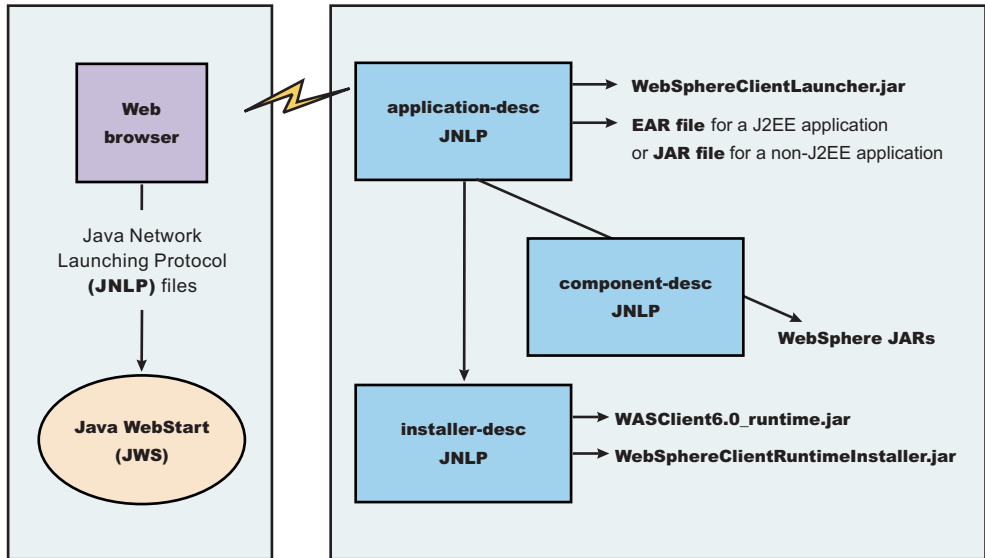
1. Specify a directory to extract the EAR file by setting the `com.ibm.websphere.client.applicationclient.archivedir` Java system property. If the directory does not exist or is empty, the EAR file is extracted normally. If the EAR file was previously extracted, the launchClient tool reuses the directory.
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. 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). Based on the JNLP descriptor, it downloads appropriate pieces of a client application and any of its dependencies. If any of the pieces of the application are already cached on the client machine, then those components are not downloaded again, unless they have been updated on the server machine. After you download and cache the client application, JWS launches it natively on the client machine.

The following diagram shows an overview of launching a client application, include the Application Client for WebSphere Application Server, 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 product version 1.4.2 or later is highly recommended. The following operating systems support running J2EE application client applications and or Thin application client applications using Java Web Start:

- Red Hat Enterprise Linux for Intel, Version 3.0
- SuSE Linux Enterprise Server, Versions 8 and 9
- Windows 2000 Professional, Windows 2000 Professional, Windows Advance Server, and Windows 2000 Advance Server
- AIX, Versions 5.1, 5.2, and 5.3
- Solaris, Versions 8 and 9
- HP-UX 11i

You can use Java Web Start on the Java 2 Standard Edition Developer Kits that IBM provides, packaged in 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.

Using 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 147
- “Client application Java Network Launcher Protocol deployment descriptor file”
- “ClientLauncher class” on page 152

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.

1. Prepare the Application Clients run-time dependency component for JWS.
2. Prepare the Application Clients run-time library component for JWS.
3. **Optional:** Run the Java Web Start sample.

Note: 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 (shown here on 2 lines for publication))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:

```
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)
```

Solution: Update the jnlp.jsp file to remove the quotation marks (" ") from the argument. Use the following example argument (shown here on 2 lines for publication) to correct the errors:

```
<argument>-url=wsejb:/com.ibm.wssvt.tc.pli.ejb.WSMultiProtocolHome?jndiName=
com/ibm/wssvt/tc/pli/ejb/WSMultiProtocolHome&</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. The client application has an Application Clients run-time dependency that provides the Java 2 Runtime Environment from IBM, Application Clients run-time properties, the SSL KeyStore and TrustStore file, and

the 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="WASclient6.0" href="/WebSphereClientRuntimeWeb/Runtime/jnlp.jsp"/>
```

It 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 which 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 the following example for details:

```
<resources>
<j2se version="WASclient6.0" href="/WebSphereClientRuntimeWeb/Runtime/jnlp.jsp"/>
<jar href="Launcher/WebSphereClientLauncher.jar" main="true"/>
<jar href="lib/j2eeclient.ear"/>
<property name="com.ibm.websphere.client.launcher.ear" value="j2eeclient.ear"/>
</resources>
```

- 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 download to JWS and the name of the class containing main method entry point is specified in the system property, `com.ibm.websphere.launcher.main`.

```
<resources>
<j2se version="WASclient6.0" href="/WebSphereClientRuntimeWeb/Runtime/jnlp.jsp"/>
  <extension name="WebSphere Runtime"
    href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJars/jnlp.jsp"/>
  <jar href="Launcher/WebSphereClientLauncher.jar" main="true"/>
  <jar href="lib/thinclient.jar"/>
  <property name="com.ibm.websphere.client.launcher.main"
    value="myapp.sample.thinclient.ThinClientMain"/>
</resources>
```

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="WebSphere Runtime"
  href="/WebSphereClientRuntimeWeb/Runtime/WebSphereJars/jnlp.jsp"/>
```

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=${WAS_ROOT}/etc/DummyClientKeyFile.jks </argument>
<argument>-CCDcom.ibm.ssl.trustStore=${WAS_ROOT}/etc/DummyClientTrustFile.jks </argument>
```

- 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` file 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>
```

If any of the default settings in the `sas.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 `/${install_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/DummyClientKeyFile.jks </argument>
<argument>-CCDcom.ibm.ssl.trustStore=\${WAS_ROOT}/etc/DummyClientTrustFile.jks </argument>
```

Here is an example of the client application JNLP descriptor file for a J2EE Application client application.

```
<%-- This is a generic jnlp for a client app. It will specify the WAS JRE
as a dependency as well as the client launcher
-->
<%! private final String description="J2EE Client Example"; private final
String earName="J2EEWebStart.ear";
%>
<% // locally declared variable

String urlSt = request.getRequestURL().toString();
String jnlpCodeBase=urlSt.substring(0,urlSt.lastIndexOf('/'));
String jnlpRefURL=urlSt.substring(urlSt.lastIndexOf('/')+1,urlSt.length());
// The client application descriptor noted a resource reference to be resolved
// at deploy time as following
%>
<%--
Need to set a JNLP mime type - if Web Start is installed on the client,
this header will induce the browser to drive the Web Start Client

--%><%
response.setContentType("application/x-java-jnlp-file"); 1
response.setHeader("Cache-Control", null);
response.setHeader("Set-Cookie", null);
response.setHeader("Vary", null);
%>
<?xml version="1.0" encoding="utf-8"?
<!-- JNLP File for <%=description %> -->
<jnlp
spec="1.0+"
<%-- Automate the code base response
-->% codebase="<%=jnlpCodeBase%>"
href="<%=jnlpRefURL%>"
<information>
<title><%=description %></title>
<description kind="short"><%=description %></description>
<description kind="tooltip"><%=description %></description>
<offline-allowed></offline-allowed>
</information>
<security>
<all-permissions></all-permissions>
```



```

</security>
<resources>
  <!-- The URL for the Client JRE installer -->
  WASClient6.0"
href="/WebSphereClientRuntimeWeb/Runtime/jnlp.jsp"></j2se> 2

  <!-- Specify the client launcher -->
  <jar href=" ../Launcher/WebSphereClientLauncher.jar" main="true"> </jar> 3

  <!-- Ear we want to download to the client -->

  <jar href="<%=earName%>"></jar> 4
  <!-- The launcher depends on this property to be set -->
  <property name="com.ibm.websphere.client.launcher.ear"
value="<%=earName%>"></property> 5

  <resources>
<!-- Web Start will consider the Launcher as the application to run -->
<application-desc> 6
<argument>-CCproviderURL=corbaloc:iiop:your_server_hostname </argument> 7
<
  <argument>-
CCDcom.ibm.ssl.keyStore=\${install_root}/etc/DummyClientKeyFile.jks</argument> 8

<argument>-
CCDcom.ibm.ssl.trustStore=
  \${install_root}/etc/DummyClientTrustFile.jksCCDcom.ibm.ssl.trustStore=
  \${install_root}/etc/DummyClientTrustFile.jks</argument> 9
</application-desc>
</jnlp>

```

- **1**--Specifies the mime type of the file must be JNLP so that the browser will know what to do with the file.
- **2**--Specifies that the application is depending on the WASClient6.0 Java Runtime Environment and specifies the URL of the JNLP for the Application Clients run-time dependency.
- **3**--Specifies the JAR file containing the launcher class. This should be the first jar specified and must contain the URL of the JAR file.
- **4**--Specifies the EAR file to be downloaded, which is similar to the one you run on an Application Client for WebSphere Application Server installation.
- **5**--Specifies the value of the EAR file name of the J2EE application.
- **6**--Specifies an application descriptor.
- **7**--Specifies the arguments for the J2EE Application client application as they are specified on the launchClient call.
- **8, 9**--Overrides the values in the sas.client.props file. They are needed because the installation location of the Application Clients run-time dependency component is unknown before it is actually installed. By default, security is turned on for the client application, and these values are required. The `\${install_root}` directory name is substituted with the Application Clients run-time dependency component installation location at run time.

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. It is packaged in the WebSphereClientLauncher.jar file that is located in a WebSphere Application Server clients installation under the `<install_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, they 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.

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 client application. This JAR file is a JAR file that is already defined as a resource for the client 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 client application.

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\WASClient6.0_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\WASClient6.0_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
WASClient6.0_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. For more information, see the topic, "Creating self-signed personal certificates" in the *Securing applications and their environment* PDF.
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 is an Installer JNLP
It will download two .jars:
WebSphereClientRuntimeInstaller.jar - includes the installer utility
WASClient6.0_<platform>.jar - the client runtime JRE image
```

```
The installer will unzip the client runtime jar on the client machine, and register
it with Java Web Start
```

```
--%>
```

```
<%! private final String description="WebSphere Client 6.0 Runtime JRE";
// The version here is (WAS based) JRE version - as to be managed on the client
private final String JREversion="WASclient6.0";%>
```

```
<%
```

```
// locally declared variable
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 Web Start is installed on the client,
// this header will induce the browser to drive the Web Start Client
response.setContentType("application/x-java-jnlp-file"); 1
response.setHeader("Cache-Control", null);
response.setHeader("Set-Cookie", null);
response.setHeader("Vary", null);
```

```
// 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", JREversion);      2
    else
        response.addHeader("x-java-jnlp-version-id", JREversion);

%>
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for <%=description %> -->
<jnlp
    spec="1.0+" <!--
        Automate the code base response --%>
        codebase="<%=jnlpCodeBase%>"
        href="<%=jnlpRefURL%>"
<information>
    <title><%=description%></title>
    <vendor>IBM</vendor>
    <icon href="icon.gif">
    <description><%=description%></description>
    <description kind="short"><%=description%></description>
    <description kind="tooltip"><%=description%></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="<%=JREversion%>"/> 5

</resources>
<resources os="Windows"> 6
    <jar href="windows/WASClient6.0_windows.jar"/> 7

    <!-- relative path of the jre executable --%>
    <property name="com.ibm.websphere.client.jre.launch.java"
value="java\jre\bin\java.exe"/> 8
<resources os="Linux">
    <jar href="linux/WASClient6.0_linux.jar"/>

    <property name="com.ibm.websphere.client.jre.launch.java" value="java/jre/bin/java"/>
</resources>
<installer-desc />
</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: 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 two configuration files, sas.client.props and soap.client.props.

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.
  type        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 <install_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/WASClient6.0_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="WASClient6.0"/>
```

Preparing Application Clients run-time library component for Java Web Start:

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 directory to the installation bin directory. See the following example for help:

```
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 keystore file, such as myKeystore.
 - b. You must also create a self-signed certificate for the myKeystore file. For more information, see the topic, "Creating self-signed personal certificates" in the *Securing applications and their environment* PDF.
4. Create an Application Clients 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.
 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 be deployed to an Application server.

```
<!--
```

```
"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
```

```
-->
```

```
<%! private final String description="WebSphere Jars";
```

```
%>
```

```
<% // locally declared variable
```

```
String urlSt = request.getRequestURL().toString();
```

```
String jnlpCodeBase=urlSt.substring(0,urlSt.lastIndexOf('/'));
```

```
String jnlpRefURL=urlSt.substring(urlSt.lastIndexOf('/')+1,urlSt.length());
```

```
// The client application descriptor noted a resource reference to be resolved at deploy time as following
```

```
%>
```

```
<%--
```

```
Need to set a JNLP mime type - if Web Start is installed on the client,  
this header will induce the browser to drive the Web Start 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);  2
%>
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for <%=description %> -->
<jnlp
  spec="1.0+"
  <!-- Automate the code base response
--> codebase="<%=jnlpcodeBase%>"
href="<%=jnlprefURL%>">
  <information>
    <title><%=description %></title>
    <description kind="short"><%=description %></description>
    <description kind="tooltip"><%=description %></description>
    <offline-allowed></offline-allowed>
  </information>
  <security>
    <all-permissions></all-permissions>
  </security>
  <resources>
    <jar href="activation-impl.jar"/>  3
    <jar href="activity.jar"/>
    <jar href="activityImpl.jar"/>
    <jar href="activitySession.jar"/>
    <jar href="activitySessionPrivate.jar"/>
    <jar href="acwa.jar"/>
    <jar href="admin.jar"/>
    <jar href="annotations-core.jar"/>
    <jar href="appprofile-impl.jar"/>
    <jar href="appprofile.jar"/>
    <jar href="b2bjaxp.jar"/>

    <!-- ===== -->
    <!-- -->
    <!-- specify all the signed jars created by -->
    <!-- buildClientLibJars tool -->
    <!-- -->
    <!-- ===== -->

    <jar href="wsif-j2c.jar"/>
    <jar href="wsif.jar"/>
    <jar href="wssec.jar"/>
    <jar href="wtp-util.jar"/>
    <jar href="wtpemf.jar"/>
    <jar href="xsd.jar"/>
    <jar href="xsd.resources.jar"/>
    <jar href="xss4j-dsig.jar"/>
    <jar href="xss4j-enc.jar"/>
  </resources>
  <component-desc/>
</jnlp>

```

- **1--**Specifies that the file is a JNLP mime type so that the browser can process the JNLP file.
- **2--**Specifies the Last-Modified header so that any changes to this JSP file are downloaded to the JNLP client.
- **3--**Specifies all the JAR files in the Application Clients run-time library component that are generated by the buildClientLibJars tool.

buildClientLibJars tool: 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\Windows
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\WASClient6.0_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
```

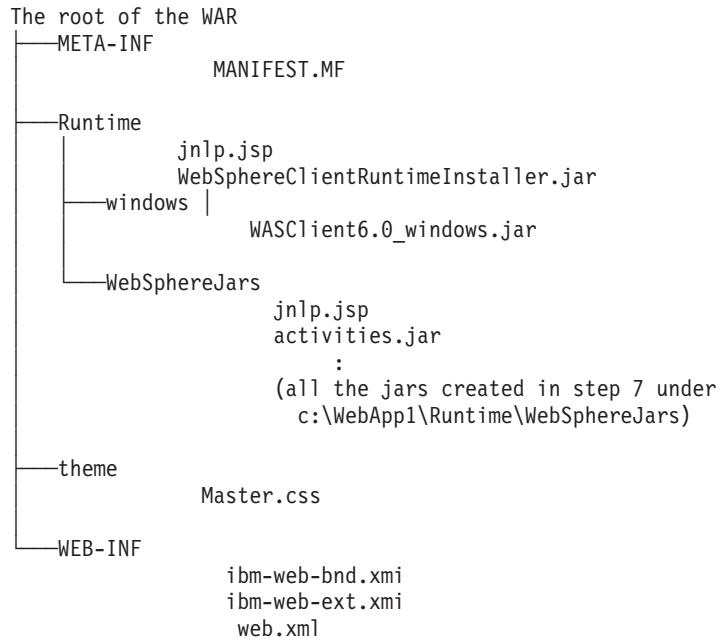
```
jarsigner -keystore myKeystore -storepass myPassword
WASClient6.0_windows.jar myKeyAliasName
```

```
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. For more information, see the topic, "Creating self-signed personal certificates" in the *Securing applications and their environment* PDF.
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 keystore file, such as myKeystore.
 - b. You must also create a self-signed certificate for the myKeystore file. For more information, see the topic, "Creating self-signed personal certificates" in the *Securing applications and their environment* PDF.
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:



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).

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 Web 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 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. 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, 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 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 ="/WSASPIIncludes/setupASPXJB.inc" -->
```

This syntax assumes that you have created a virtual directory in Internet Information Server called WSASPIIncludes 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

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). The majority of tasks for accessing your Java classes and objects are handled with the JClassProxy and JObjectProxy 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 "XJBINI::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 strObjectName, 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 strObjectName, 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

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.
- Take care when invoking Java methods through ActiveX Automation. If you use the wrong case on a method call or use the wrong parameter type, you get an Automation Error 438 "Object doesn't support this property or method" thrown.
- 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.util.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 1. ActiveX to Java primitive data type conversion

Visual Basic Type	Variant Type	Java Type	Notes
Byte	VT_I1	byte	Byte in Visual Basic is unsigned, but is signed in Java data type.
Boolean	VT_BOOL	boolean	
Integer	VT_I2	short	
Long	VT_I4	int	
Currency	VT_CY	long	
Single	VT_R4	float	
Double	VT_R8	double	
String	VT_BSTR	java.lang.String	
String	VT_BSTR	char	
Date	VT_DATE	n/a	

Example: Using helper methods for data type conversion: Generally, data type conversion between ActiveX (Visual Basic and VBScript) and Java methods occurs automatically, as described in ActiveX to EJB bridge, converting data types. However, the following helper functions are provided for cases where automatic conversion is not possible:

- Byte helper function
- Currency helper function
- 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. 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) = CDb1(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. In addition, 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 KnowledgeBase 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), you need to redirect the output to a file. For example:

```
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

In general, 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.

- Visual Basic guidelines
- 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()`, `CInq()`, `CSng()` and `CDbl()` functions to tell the activeX to EJB bridge which data type you are using; for example `oMyObject.Foo(CDb1(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 have the following setup requirements:

- These clients are available on the Windows XP or Windows 2000 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. Run the application server client installation.
2. Select the applet client option.
3. Install an applet client.

4. Install the WebSphere Application Server plug-in for the browser. From the WebSphere Application Server Java Plug-in control panel, enter the following code:

```
-Djava.security.policy=<product_installation_dir>\properties\client.policy
-Dwas.install.root=<product_installation_dir>
-Djava.ext.dirs=<product_installation_dir>\classes;
<product_installation_dir>\java\jre\lib\ext;
<product_installation_dir>\java\jre\lib;
<product_installation_dir>\lib;<product_installation_dir>\properties
-Dcom.ibm.CORBA.ConfigURL=file:<product_installation_dir>\properties\sas.client.props
-classpath <product_installation_dir>\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 control panel from the **Start** menu. Click **Start > Control panel** > select the product Java plug-in.
- The applet container is the Web browser and the Java plug-in combination. You must first install the WebSphere Application Server Applet client 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 global 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` file.

1. Make a copy of the following file so that you can use it for an applet:
`<product_install_directory>/properties/sas.client.props`
2. Edit the copy of `sas.client.props` file 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 file you created in step 1, modify the following value:

```
-Dcom.ibm.CORBA.ConfigURL=file:<product_install_directory>\properties\sas.client.props
```

For more information on the `sas.client.props` file 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 `<product_installation_dir>/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.

- The classid and type attributes cannot be modified, and must be entered as described in the applet client example. The codebase attribute on the <OBJECT> tag must be excluded. Do not confuse the codebase attribute on the <OBJECT> tag with the codebase attribute on the <PARAM> tag. Although both attributes are called codebase, they are separate entities.
- The following code example illustrates the applet code. In this example, MyApplet.class is the applet code, applet.jar is the file that contains the applet code, and EJB.jar is the file that contains the enterprise bean code:

```
<OBJECT classid="clsid:8AE2D840-EC04-11D4-AC77-006094334AA9"
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=1.4">
<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=1.4" 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>
```

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

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 Web 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 Web 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 `installation_root/bin/setupCmdLine` command shell for WebSphere Application Server installations, or in the `installation_root/bin/setupClient` command shell 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 Data Base 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 Web 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 client.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/client.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.

Using the launchClient API

If you use the `launchClient` shell and `bat` 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.

Developing Pluggable application client code

As you prepare to install the Pluggable application client, remember that pluggable clients are only available on Windows systems.

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:
`install_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 the following 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 application jars and classes> -Djava.ext.dirs=%WAS_EXT_DIRS%
-Djava.naming.provider.url=iiop://<your WebSphere server machine name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
%SERVER_ROOT% %CLIENTSAS% <fully qualified class name to run>

$JAVA_HOME/bin/java $WAS_LOGGING
-classpath "$WAS_CLASSPATH: <list application jars and classes> -Djava.ext.dirs=$WAS_EXT_DIRS
-Djava.naming.provider.url=iiop://<your WebSphere server machine name>
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
$SERVER_ROOT $CLIENTSAS <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. 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

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 by using the setupClient shell, located in: Windows systems:

```
install_root\AppClient\bin\setupClient.bat
```

UNIX platforms:

```
install_root/AppClient/bin/setupClient.sh
```

2. Run the following Java compilation command to compile your client application. On Windows systems, enter:

```
"%JAVA_HOME%\bin\javac" -classpath "%WAS_CLASSPATH%;  
<list of your application jars and classes> " -extdirs %WAS_EXT_DIRS%  
<your application class>.java
```

On UNIX systems, enter:

```
$JAVA_HOME/bin/javac -classpath "$WAS_CLASSPATH:  
<list of your application jars and classes>" -extdirs $WAS_EXT_DIRS  
<your application class>.java
```

3. Run the following Java command to invoke your client application: On Windows systems, enter:

```
"%JAVA_HOME%\bin\java" %WAS_LOGGING% -Djava.security.auth.login.config=  
"%WAS_HOME%\properties\wsjaas_client.conf"  
-classpath "%WAS_CLASSPATH%;<list application jars and classes> -Djava.ext.dirs=%WAS_EXT_DIRS%  
-Djava.naming.provider.url=iiop://<your WebSphere server machine name>  
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory  
%SERVER_ROOT% %CLIENTSAS% <fully qualified class name to run>
```

On UNIX systems, enter:

```
$JAVA_HOME/bin/java $WAS_LOGGING  
-classpath "$WAS_CLASSPATH: <list application jars and classes> -Djava.ext.dirs=$WAS_EXT_DIRS  
-Djava.naming.provider.url=iiop://<your WebSphere server machine name>  
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory  
$SERVER_ROOT $CLIENTSAS <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

You must install WebSphere Application Server before performing this task.

1. Set the Thin application client environment by using the **setupCmdLine** shell, located in:

```
install_root\bin\setupCmdLine.bat (on Windows systems)  
install_root/bin/setupCmdLine.sh (on UNIX platforms)
```

2. Run the following Java compilation command to compile your client application: On Windows systems, enter:

```
"%JAVA_HOME%\bin\javac" -classpath "%WAS_CLASSPATH%;  
<list of your application jars and classes> " -extdirs %WAS_EXT_DIRS%  
<your application class>.java
```

On UNIX systems, enter:

```
$JAVA_HOME/bin/javac -classpath "$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:

On Windows systems, enter:

```
"%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%"  
-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%" %USER_INSTALL_PROP%  
-classpath "%WAS_CLASSPATH%;<list of your application jars and classes>"  
<fully qualified class name to run><your application parameters>
```

On UNIX systems, enter:

```
"$JAVA_HOME/bin/java" "WAS_LOGGING"  
-Djava.security.auth.login.config="$WAS_HOME/properties/wsjaas_client.conf"  
-Djava.ext.dirs="$JAVA_HOME/jre/lib/ext;%WAS_EXT_DIRS%"  
-Djava.naming.provider.url=<an IIOP URL or a corbaloc URL to your  
WebSphere server machine name>  
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory  
-Dserver.root="$WAS_HOME" $USER_INSTALL_PROP "$CLIENTSAS"  
-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:

On Windows systems, enter:

```
"%JAVA_HOME%\bin\java" "WAS_LOGGING"  
-Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"  
"-Dws.ext.dirs=<list of your application jars and classes>;  
%WAS_EXT_DIRS%;%WAS_USER_DIRS%">  
-Djava.naming.provider.url=<an IIOP URL or a corbaloc URL to your  
WebSphere server machine name>  
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory  
"-Dserver.root=%WAS_HOME%"  
"%CLIENTSAS%" %USER_INSTALL_PROP% -classpath "%WAS_CLASSPATH%"  
com.ibm.ws.bootstrap.WSLauncher  
<fully qualified class name to run><your application parameters>
```

On UNIX systems, enter:

```
"$JAVA_HOME/bin/java" "WAS_LOGGING"  
-Djava.security.auth.login.config="$WAS_HOME/properties/wsjaas_client.conf"  
"-Dws.ext.dirs=<list of your application jars and classes>  
$WAS_EXT_DIRS;$WAS_USER_DIRS"  
-Djava.naming.provider.url=<an IIOP URL or a corbaloc URL to your  
WebSphere server machine name>  
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory  
"-Dserver.root=$WAS_HOME"  
"$CLIENTSAS" $USER_INSTALL_PROP -classpath "$WAS_CLASSPATH"  
com.ibm.ws.bootstrap.WSLauncher  
<fully qualified class name to run><your application parameters>
```

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.

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 Web 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 the **J2EE** capability is enabled.
 3. Migrate application client JAR files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. 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. To deploy your application on Windows, refer to "Deploying J2EE application clients on workstation platforms."

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 > Defining J2EE application clients**.

Deploying J2EE application clients on workstation platforms

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.bat command (on Windows systems) or setupCmdLine.sh (on UNIX platforms) command 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

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.

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}`

clientRAR tool:

This section 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:

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:

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

Starting the Application Client Resource Configuration Tool and 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 `install_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 might 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:

Configuring new data source providers (JDBC providers) for application clients

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: 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 XML 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+PCswLSZ1MT4y0g="/>
</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.

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 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

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** > **JavaMail Providers** > **java_mail_provider_instance** > **JavaMail 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: 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}Mj42Mww6LCw2MDF1MT4y0g=="
mailFrom="mailSession:mailFrom" mailStoreHost="mailSession:mailStoreHost"
mailStoreUser="mailSession:mailStoreUser"
mailStorePassword="{xor}Mj42Mww6LCw2MDF1MT4y0g==" 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

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 **JavaMail Providers > MailProvider > JavaMail Sessions**. Complete one of the following actions:
 - Right click the **JavaMail Sessions** folder and select **New Factory**.
 - Click **Edit > New** on the menu bar.
5. Configure the JavaMail 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

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 Provider**.
 - 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:

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: 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

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. Do 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 Factory**.
 - 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.

WebSphere asynchronous messaging using the Java Message Service API for the Application Client Resource Configuration Tool

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 an overview of asynchronous messaging using JMS support provided by the 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.

Configuring Java messaging client resources

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.

During this task, 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*.

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. Right-click **Messaging Providers** and select **New**.
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:

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	<p>None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.</p> <p>Best effort nonpersistent Messages are never written to disk, and are thrown away if memory cache overruns.</p> <p>Express nonpersistent Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.</p> <p>Reliable nonpersistent Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.</p> <p>Reliable persistent Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.</p> <p>Assured persistent Highest degree of reliability where assured message delivery is supported.</p> <p>As Bus destination Use the delivery option configured for the bus destination.</p>

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

Related concepts

“Resource Adapters for the client” on page 185

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	<code>merlin:7276:BootstrapBasicMessaging,Gandalf:5557:BootstrapSecureMessaging</code>
----------------	--

where

BootstrapBasicMessaging corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).

Default	<ul style="list-style-type: none">• 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	<p>None There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded.</p> <p>Best effort nonpersistent Messages are never written to disk, and are thrown away if memory cache overruns.</p> <p>Express nonpersistent Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts.</p> <p>Reliable nonpersistent Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions.</p> <p>Reliable persistent Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions.</p> <p>Assured persistent Highest degree of reliability where assured message delivery is supported.</p> <p>As Bus destination Use the delivery option configured for the bus destination.</p>

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. <i>If you select this option, you must define a priority on the Specified priority property.</i>

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 <ul style="list-style-type: none">• 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. <i>If you select this option, you must define a priority on the Specified priority property.</i>

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. <i>If you select this option, you must define a timeout on the Specified expiry property.</i> 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 <ul style="list-style-type: none">• 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	<p>BINDINGS</p> <p>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.</p> <p>CLIENT</p> <p>WebSphere MQ client connection is used to connect to the queue manager. CLIENT is a typical TCP-based protocol.</p> <p>DIRECT</p> <p>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.</p> <p>QUEUED</p> <p>QUEUED is a standard TCP protocol.</p>
Recommended	<p>Queue connection factory transport type</p> <p>BINDINGS is faster by 30% or more, but it lacks security. When you have security concerns, BINDINGS is more desirable than CLIENT.</p> <p>Topic connection factory transport type</p> <p>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 (shown here on 2 lines for publication):</p> <pre>#wempschangeproperties WAS_nodeName_server1 -e default -o DynamicSubscriptionEngine -n maxClientQueueSize -v 0 -x executionGroupUUID</pre> <p>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.</p>

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=WEBSHERE`

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 WEBSHERE. 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.

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=WEBSHERE`

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 WEBSHERE. 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

Related tasks

“Configuring new JMS providers with the Application Client Resource Configuration Tool” on page 203

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
Range**

APPLICATION_DEFINED

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
Units
Default
Range**

Enum

Not applicable

APPLICATION_DEFINED

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
Units
Range**

Integer

Message priority level

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
Units
Default**

Enum

Not applicable

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
Default
Range

Enum
NORMAL
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
Default
Range

Enum
NORMAL
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
Default
Range

Enum
IEEEENORMAL
IEEEENORMAL

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
Default
Range

Enum
Cleared
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
Default
Range

Enum
APPLICATION_DEFINED

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
Units
Range

Integer
Milliseconds
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

Integer encoding:

Indicates whether integer encoding is normal or reversed when native encoding is not enabled.

Data type
Default
Range

Enum
NORMAL
NORMAL
REVERSED

Normal integer encoding is used.

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:

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:JMSPProvider xmi:id="JMSPProvider_3" name="genericJMSPProvider:name"
description="genericJMSPProvider:description"
externalInitialContextFactory="genericJMSPProvider:contextFactoryClass"
externalProviderURL="genericJMSPProvider:providerUrl">
<classpath>genericJMSPProvider: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}NTIsHB11MT4y0g=="
```

```

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 **JMS Connection Factories** folder.
3. Click the connection factory folder, and complete one of the following actions:
 - Right-click the folder and select **New Factory**.
 - 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 **JMS 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.

Example: Configuring MQ Queue and Topic connection factories and destination factories for application clients

The purpose of this article is to help you configure MQ Queue connection factory, MQ Topic connection factory, MQ Queue destination factory, and MQ Topic destination factory settings.

- Required fields:
 - MQ Queue Connection Factory Properties page: name, jndiName and transport type
 - MQ Topic Connection Factory Properties page: name, jndiName and broker Version
 - MQ Queue Factory Properties page: name, jndiName, persistence, priority, expiry, baseQueueName and targetClient
 - MQ Topic Factory Properties page: name, jndiName, persistence, priority, expiry, baseQueueName and targetClient
- Special cases:
 - The transport type must be CLIENT, or BINDINGS.

- The Broker Version must be MA0C, or MQSI.
 - The port must be a numerical value between -2417483648 and 2417483647.
 - The CCSID must be a numerical value between -2417483648 and 2417483647.
 - The persistence value must be APPLICATION_DEFINED, QUEUE_DEFINED, PERSISTENT or, NONPERSISTENT.
 - The priority must be APPLICATION_DEFINED, QUEUE_DEFINED, or SPECIFIED.
 - The expiry must be APPLICATION_DEFINED, UNLIMITED, or SPECIFIED.
 - The integer encoding must be Normal, or Reversed.
 - The decimal encoding must be Normal, or Reversed.
 - The floating encoding must be IEEENormal, IEEEReversed or S390.
 - The target client must be JMS or MQ.
 - On the MQ Queue Connection Factory Properties page, only set the queueManager, host, and port values. These are required fields if the transport type is CLIENT.
 - On the MQ Topic Connection Factory Properties page, only set the queueManager, host, and port (required) fields if the transport type is CLIENT.
 - On the MQ Topic Factory Properties, and the MQ Queue Factory Properties pages, only set the Integer encoding, decimal encoding, and floating point encoding (required) fields if you do not set the nativeEncoding value.
 - On the MQ Topic Factory Properties and the MQ Queue Factory Properties pages, the specified priority entry field must be an integer between 0 and 9 if priority is set to SPECIFIED .
 - On the MQ Topic Factory Properties and the MQ Queue Factory Properties pages, the specified expiry entry field must be a value greater than 0 if the expiry value is set to SPECIFIED.
- Example:

```
<resources.jms.JMSProvider xmi:id="JMSProvider_1" name="MQ JMS Provider"
description="mqJMSProvider:description"
externalInitialContextFactory="mqJMSProvider:contextFactoryClass"
externalProviderURL="mqJMSProvider:providerUrl">
<classpath>mqJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms.mqseries:MQQueueConnectionFactory"
xmi:id="MQQueueConnectionFactory_1" name="mqQCF:name" jndiName="mqQCF:jndiName"
description="mqQCF:description" userID="mqQCF:user" password="{xor}Mi40HB1lMT4y0g=="
queueManager="mqQCF:queueManager" host="mqQCF:host" port="1" channel="mqQCF:channel"
transportType="CLIENT" clientID="mqQCF:clientID" CCSID="2">
<propertySet xmi:id="J2EEResourcePropertySet_3">
<resourceProperties xmi:id="J2EEResourceProperty_3" name="mqQCF:customName"
value="mqQCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.mqseries:MQTopicConnectionFactory"
xmi:id="MQTopicConnectionFactory_1" name="mqTCF:name" jndiName="mqTCF:jndiName"
description="mqTCF:description" userID="mqTCF:user"
password="{xor}Mi4LHB1lNTE7NhE+Mjo=" host="mqTCF:host" port="1"
transportType="CLIENT" channel="mqTCF:channel" queueManager="mqTCF:queueManager"
brokerControlQueue="mqTCF:brokerControlQueue"
brokerQueueManager="mqTCF:brokerQueueManager" brokerPubQueue="mqTCF:brokerPubQueue"
brokerSubQueue="mqTCF:brokerSubQueue" brokerCCSubQ="mqTCF:brokerCCSubQ"
brokerVersion="MA0C" clientID="mqTCF:clientID" CCSID="2">
<propertySet xmi:id="J2EEResourcePropertySet_4">
<resourceProperties xmi:id="J2EEResourceProperty_4" name="mqTCF:customName"
value="mqTCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.mqseries:MQQueue" xmi:id="MQQueue_1" name="mqQ:name"
jndiName="mqQ:jndiName" description="mqQ:description" persistence="APPLICATION_DEFINED"
priority="SPECIFIED" specifiedPriority="1" expiry="SPECIFIED" specifiedExpiry="1"
baseQueueName="mqQ:baseQueueName" baseQueueManagerName="mqQ:baseQueueManagerName"
CCSID="1" integerEncoding="Normal" decimalEncoding="Normal"
floatingPointEncoding="IEEENormal" targetClient="JMS">
<propertySet xmi:id="J2EEResourcePropertySet_5">
<resourceProperties xmi:id="J2EEResourceProperty_5" name="mqQ:customName"
value="mqQ:customValue"/>
</propertySet>
</factories>
```

```

<factories xmi:type="resources.jms.mqseries:MQTopic" xmi:id="MQTopic_1"
name="mqT:name" jndiName="mqT:jndiName" description="mqT:description"
persistence="APPLICATION_DEFINED" priority="SPECIFIED" specifiedPriority="1"
expiry="SPECIFIED" specifiedExpiry="2" baseTopicName="mqT:baseTopicName" CCSID="3"
integerEncoding="Normal" decimalEncoding="Normal" floatingPointEncoding="IEEENormal"
targetClient="JMS" brokerDurSubQueue="mqT:brokerDurSubQueue"
brokerCCDurSubQueue="mqT:brokerCCDurSubQueue">
<propertySet xmi:id="J2EEResourcePropertySet_6">
<resourceProperties xmi:id="J2EEResourceProperty_6" name="mqT:customName"
value="mqT:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_7">
<resourceProperties xmi:id="J2EEResourceProperty_7" name="mqJMSProvider:customName"
value="mqJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>

```

Example: Configuring WAS Queue and Topic connection factories and destination factories for application clients

The purpose of this article is to help you configure Queue connection factory, Topic connection factory, Queue destination factory, and Topic destination factory settings.

- Required fields include:
 - Java Message Service (JMS) Provider Properties page: name
 - WebSphere Application Server Queue Connection Factory Properties page: name, jndiName and node
 - WebSphere Application Server Topic Connection Factory Properties page: name, jndiName, node and port
 - WebSphere Application Server Queue Factory Properties page: name, jndiName, node, persistence, priority and expiry
 - WebSphere Application Server Topic Factory Properties page: name, jndiName, topic name, persistence, priority and expiry
- Special cases:
 - The port value must be QUEUED or DIRECT.
 - The CCSID must be a numerical value between -2417483648 and 2417483647.
 - The persistence value must be APPLICATION_DEFINED, PERSISTENT, or NONPERSISTENT.
 - The priority value must be APPLICATION_DEFINED, or SPECIFIED.
 - The expiry value must be APPLICATION_DEFINED, UNLIMITED, or SPECIFIED.
 - On the WAS Topic Factory Properties, and the WAS Queue Factory Properties pages, the specified priority entry field must be an integer between 0 and 9, if the priority value is set to SPECIFIED .
 - On the WAS Topic Factory Properties, and the WAS Queue Factory Properties pages, the specified expiry entry field must be a value greater than 0 if expiry is set to SPECIFIED.
- Example:

```

<resources.jms:JMSProvider xmi:id="JMSProvider_2" name="WebSphere JMS Provider"
description="wasJMSProvider:description"
externalInitialContextFactory="wasJMSProvider:contextfactoryclass"
externalProviderURL="wasJMSProvider:providerURL">
<classpath>wasJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms.internalmessaging:WASQueueConnectionFactory"
xmi:id="WASQueueConnectionFactory_1" name="wasQCF:name" jndiName="wasQCF:jndiName"
description="wasQCF:description" userID="wasQCF:user" password="{xor}KD4sDhwZZS0s0i0="
node="wasQCF:Node">
<propertySet xmi:id="J2EEResourcePropertySet_8">
<resourceProperties xmi:id="J2EEResourceProperty_8" name="wasQCF:customName"
value="wasQCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.internalmessaging:WASTopicConnectionFactory"
xmi:id="WASTopicConnectionFactory_1" name="wasTCF:name" jndiName="wasTCF:jndiName"
description="wasTCF:description" userID="wasTCF:user" password="{xor}KD4sCxwZZTE+Mjo="
node="wasTCF:node" port="QUEUED" clientID="wasTCF:clientId">
<propertySet xmi:id="J2EEResourcePropertySet_9">

```



```

<resourceProperties xmi:id="J2EEResourceProperty_9" name="wasTCF:customName"
value="wasTCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.internalmessaging:WASQueue" xmi:id="WASQueue_1"
name="wasQ:name" jndiName="wasQ:jndiName" description="wasQ:description"
node="wasQ:node" persistence="APPLICATION_DEFINED" priority="SPECIFIED"
specifiedPriority="1" expiry="SPECIFIED" specifiedExpiry="1">
<propertySet xmi:id="J2EEResourcePropertySet_10">
<resourceProperties xmi:id="J2EEResourceProperty_10" name="wasQ:customName"
value="wasQ:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.internalmessaging:WASTopic" xmi:id="WASTopic_1"
name="wasT:name" jndiName="wasT:jndiName" description="wasT:description"
topic="wasT:topicName" persistence="APPLICATION_DEFINED" priority="SPECIFIED"
specifiedPriority="1" expiry="SPECIFIED" specifiedExpiry="1">
<propertySet xmi:id="J2EEResourcePropertySet_11">
<resourceProperties xmi:id="J2EEResourceProperty_11" name="wasT:customName"
value="wasT:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_12">
<resourceProperties xmi:id="J2EEResourceProperty_12" name="wasJMSProvider:customName"
value="wasJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>

```

Configuring new resource environment providers for application clients

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 Provider**.
 - 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

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 entry**, 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.

Creating locally defined objects for message destination references and message destinations

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 application client .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.

When a local object definition is created using the ACRCT, the JNDI name of the local object definition points to the reference for which the local object definition applies. The object might be a resource-ref, resource-env-ref or message-destination-ref. If the message-destination-ref has a message-destination-link, then point the local object definition to the message-destination. Local object definitions either point to the message-destination-ref (if the message-destination-ref has no link) or to the message-destination (if the message-destination-ref has a link). Any local object definitions pointing to message-destination-refs that have a link are ignored.

1. Start an assembly tool such as Application Server Toolkit (AST) or Rational Web Developer, and open an EAR file.
2. Create a locally defined object.
3. Point the object to the appropriate message destination.
4. Save the EAR file.

Managing application clients

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:

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
 - Data source 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:

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:

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:

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:

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:

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: 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: 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:

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 proceeding 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**.

Web services

Learn about Web services

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Develop and assemble applications that use Web services

- Implement Web services applications
- Plan your use of Web services
- Develop Web services applications
- Configure Web services deployment descriptors
- Assemble Web services applications

Deploy and administer applications

- Deploy Web services, using the administrative console
- Configure Web service client bindings
- Publish WSDL files
- Administer applications (same as any application)
- Administer applications (Education on Demand)

Develop Web services clients

- Develop Web services clients
- Develop client bindings from a WSDL file
- Assemble a Web services-enabled client WAR file into an EAR file
- Configure a client deployment descriptor (refer to the *Administering applications and their environment* PDF)
- Configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor (refer to the *Administering applications and their environment* PDF)
- Test Web services-enabled clients

Secure Web services

Invoke Web services using Web Services Invocation Framework (WSIF)

Refer to the *Securing applications and their environment* PDF.

- Enable Web services with WSIF
- Invoke Web services with WSIF (refer to the *Administering applications and their environment* PDF)
- Administer WSIF (refer to the *Administering applications and their environment* PDF)

Use the Universal Description, Discovery and Integration (UDDI) registry

- Access and administer an IBM WebSphere UDDI registry (refer to the *Administering applications and their environment* PDF)

Tune and troubleshoot Web services

- View WS-Security bindings, using the administrative console

Refer to the *Tuning performance* and *Troubleshooting and support* PDFs.

Conceptual overviews

Documentation Presentations

Refer to the article *Introduction: Web services* in the information center. Education on Demand offers:

- J2EE 1.4 Web services overview
- Web services in Version 6.0.x
- WS-Security in Version 6.0.x
- Universal Description, Discovery, and Integration (UDDI)

See also the IBM Redbook WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook

Note:

- Chapter 2 (implementation) does not apply to Version 6.
- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Education on Demand offers:

developerWorks offers:

- Deploying Web service applications
- Universal Description, Discovery, and Integration (UDDI) registry
- Tutorial 4 - Web services

The objective of this tutorial is to provide you with an understanding of the new standard JAX-RPC programming model to develop SOAP based Web service clients and endpoints. The endpoint is described by using WSDL. The zip file comes with all sample code required to run this tutorial.

Samples

The Samples Gallery offers:

- **WebSphere Bank**

Using the WebSphere Bank online bank, customers can open accounts, get account balances, and transfer funds between accounts. The WebSphere Bank application uses Web services, Java Message Service (JMS) API, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, Message-Driven Beans (MDB), JSP pages, and servlets.

- **Greenhouse by WebSphere**

Using the Greenhouse by WebSphere online supplier, customers can open accounts, select items and amounts to order, and check their order status. The Greenhouse by WebSphere application uses Web services, the Java message service (JMS) API, scheduler, asynchronous beans, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, message-driven beans (MDB), Java server pages (JSP)s, and the struts framework.

- **Adventure Builder**

The Adventure Builder customer Web site resides on the Web tier and is designed using a Web application architecture. This Web site communicates to the order processing backend module using Web services interactions. Adventure Builder, a basic Web site travel application built on the J2EE 1.4 platform, is a simple shopping application.

The customer can browse and select from a catalog of products, in this case vacation packages, assemble or build an entire vacation from different components, principally lodging and activities. The parts of a particular vacation package are determined by user responses given on a sequence of forms. You can maintain vacation package options in a virtual shopping cart, perform sign on and sign off procedures, create user accounts, and purchase a trip package, sending a purchase order to the order fulfillment system. The Adventure Builder application uses several J2EE 1.4 technologies.

- **Development Strategies - Address Book**

The Address Book sample illustrates accessing multiple Web Services in one application.

- **Migration - Stock Quote**

The Stock Quote sample illustrates migration of a stock quote client from Simple Object Access Protocol (SOAP) to Java API for XML-based RPC (JAX-RPC). WebSphere supports Web Services for J2EE (JSR 109) which builds on a client-programming model on JAX-RPC.

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.

Using Web services makes most sense if your application clients are non-J2EE applications, unless you have J2EE applications spread across the Web. It is recommended that you use J2EE technologies if all your clients are J2EE applications because performance can decrease when you use a Web service in a J2EE exclusive environment.

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 Web service client. Now, the accounting application can 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.

Use Web services applications with WebSphere Application Server by following the steps provided:

1. Plan to use Web services.
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. Refer to "Publishing the WSDL file" in the information center or in the *Administering applications and their environment* PDF.
After installing a Web services application, and optionally modifying the endpoint information, you might need 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.
11. Tune Web services.
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.

The flower shop owner can request price lists from each of the suppliers by obtaining a Web Services Description Language (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" in the information center 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

Web services are self-contained, modular applications that you can describe, publish, locate, and invoke over a network.

WebSphere Application Server supports Web services that are developed and implemented based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

A typical Web services scenario is a business application requesting a service from another existing application. The request is processed through a given Web address using SOAP messages over a HTTP, Java Message Service (JMS) transport or invoked directly as Enterprise JavaBeans (EJB). The service receives the request, processes it, and returns a response. Examples of a simple Web service include weather reports or getting stock quotes. The method call is synchronous, that is, it waits until the result is available. Transaction Web services, supporting quotes, business-to-business (B2B) or business-to-client (B2C) operations include airline reservations and purchase orders.

Web services can include the actual service or the client that accesses the service.

Web services are Web applications that help you be more flexible in your business processes by integrating with applications that otherwise do not communicate. The inner-library loan program at your local library is a good example of the Web services concept and its evolution. The Web service concept existed even before the term; the concept became widely accepted with the creation of the Internet. Before the Internet was created, you visited your library, searched the collections and checked out your books. If you did not find the book that you wanted, the librarian did a search for you by computer or phone and located the book at a nearby library. The librarian ordered the book for you and you picked it up after it was delivered to your local library. By incorporating Web services applications, you can streamline your library visit.

Now, you can search the local library collection and other local libraries at the same time. When other libraries provide your library with a Web service to search their collection (the service might have been provided through UDDI), your results yield their resources. Another Web service application might enable you to check the book out and get it sent to your home. Using Web services applications saves time and provides a convenience for you, as well as freeing the librarian to do other business tasks.

Web services reflect the service-oriented architecture (SOA) approach to programming. This approach is based on the idea of building applications by discovering and implementing network-available services, or by invoking the available applications to accomplish a task. Web services deliver interoperability, for example, Web services applications provide components created in different programming languages to work together as if they were created using the same language. Web services rely on existing transport technologies, such as HTTP, and standard data encoding techniques, such as Extensible Markup Language (XML), for invoking the implementation.

The key components of Web services include:

- Web Services Description Language (WSDL)

WSDL is the XML-based file that describes the Web service. The Web service request uses this file to bind to the service.

- SOAP

SOAP is the XML-based protocol that the Web service request uses to invoke the service.

For a more detailed scenario, see "Web services scenario: Overview" in the information center, which tells the story of a fictional online garden supply retailer named Plants by WebSphere, and how this retailer 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.

Version 6.0 uses Web Services for J2EE 1.1 as the standard for developing and implementing Web services. Web Services for J2EE 1.1 is one of the Web service APIs available in J2EE 1.4.

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*.

To review the entire Web Services for J2EE specification, see *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; most of these rules are from RMI with some additions for JAX-RPC. The following are the rules for defining a JAX-RPC interface:

- 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.

To learn more about JAX-RPC see [Web services: Resources for learning](#).

SOAP

Simple Object Access Protocol (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.

For more information about SOAP, see [Web services: Resources for learning](#).

SOAP with Attachments API for Java

SOAP with Attachments API for Java (SAAJ) is used for SOAP messaging that works behind the scenes in the Java API for XML-based RPC (JAX-RPC) implementation.

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. The SOAP message is manipulated using the SAAJ data model. The primary interface in the SAAJ model is `javax.xml.soap.SOAPElement`.

WebSphere Application Server uses SAAJ Version 1.2. The main benefit of SAAJ Version 1.2 is that the model extends the Document Object Model (DOM) model. The DOM model is used by applications that manipulate XML. Using this model applications are able to 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.

To review the entire SAAJ API, see [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. A link to this Web site is listed in Web services: Resources for learning.

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.

To learn more about the WS-Basic profile, including scenarios, UTF and BOM, see Web services: Resources for learning.

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 an 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 complements 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 `ws:swaRef` XML schema type. Use the `ws:swaRef` XML schema type to overcome the limitations of WSDL 1.1 attachments.

The `ws:swaRef` type is an extension of the `xsd:anyURI` type, where its value contains the content-ID of the attachment.

Web services: Resources for learning

This topic provides relevant supplemental information about the following Web services-related topics:

- Web services overview
- Developing Web services:
 - Includes developing Web services based on the Java 2 Platform, Enterprise Edition (J2EE) and Java API for XML-based remote procedure call (JAX-RPC) specifications.
- Universal Description Discovery and Integration (UDDI)
 - Includes an overview about UDDI and information about the UDDI Java API.
- The Web Services Invocation Framework (WSIF)
 - Includes a look into the Apache Software Foundation and its maintenance of WSIF.
- Web Services-Interoperability (WS-I) Basic Profile
 - Includes an overview about the WS-I Basic Profile.
- SOAP
 - Includes an overview about SOAP, information about the SOAP syntax and processing rules.
- Security

Includes a roadmap to security, the WS-Security specification, best practices, a profile of the OASIS Security Assertion Markup Language (SAML) and more.

- Samples

Includes 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 5.1.1 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 Java 2 Platform, Enterprise Edition (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 some of the basic JAX-RPC programming concepts. It describes the JAX-RPC client and server programming models and provides some simple examples for illustration. 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.

UDDI

- Universal Description, Discovery and Integration
This article is a detailed overview of Universal Description, Discovery, and Integration (UDDI).
- 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 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. 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 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. Web Services security also provides a general-purpose mechanism for associating security tokens with messages. Additionally, Web Services Security 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 Web Services 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 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 encryption 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 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, 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 Web Services 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).

Deploying Web services

This task explains how to deploy a Web service into WebSphere Application Server.

To deploy Web services that are based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification, you need an enterprise application, also known as an enterprise archive (EAR) file that is configured and enabled for Web services.

If you have a Web service that was deployed on a previous version of WebSphere Application Server, you might want to run the **wsdeploy** command so that you can benefit from performance features that have been added to this release.

This task is one of the steps in developing and implementing Web services.

You can use either the administrative console or the **wsadmin** scripting tool to deploy an EAR file. If you are installing an application containing Web services by using the **wsadmin** command, specify the **-deployws** option. If you are installing an application containing Web services by using the administrative console, select **Deploy WebServices** in the Install New Application wizard. For more information about installing applications using the administrative console see Installing a new application.

If the Web services application is previously deployed with the **wsdeploy** command, it is not necessary to specify Web services deployment during installation.

The following actions deploy the EAR file with the **wsadmin** command:

1. Start `install_root\bin\wsadmin` from a command prompt. If you are using Linux or Unix platforms, start `install_root/bin/wsadmin.sh`.
2. Enter the **\$AdminApp install EARfile "-usedefaultbindings -deployws"** command at the **wsadmin** prompt.

You have a Web service installed into the Application Server.

Protect your Web services applications by adding security to the applications.

wsdeploy command

This topic explains how to use the **wsdeploy** command-line tool with Web services that are based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification. The **wsdeploy** command adds WebSphere product-specific deployment classes to a Web services-compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file. These classes include:

- Stubs
- Serializers and deserializers
- Implementations of service interfaces

This deployment step must be performed at least once, and can be performed more often. Deployment can be performed separately using the **wsdeploy** command, assembly tools, or when the application is installed. When using the **wsadmin** command for installation, specify the **-deployws** option.

The **wsdeploy** command operates as noted in the following list:

- Each module in the enterprise application or JAR file is examined
- If the module contains Web services implementations, indicated by the presence of the `webservices.xml` deployment descriptor, the associated Web Services Description Language (WSDL) files are located and the **WSDL2Java** command is run with the `deploy-server` option.
- If the module contains Web services clients, indicated by the presence of the client deployment descriptor, the associated WSDL files are located and the **WSDL2Java** command is run with the `deploy-client` option.
- The files generated by the **WSDL2Java** command are compiled and repackaged.

See **WSDL2Java** command for more information about the files that are generated for deployment.

When the generated files are compiled, they can reference application-specific classes outside the EAR or JAR file, if the EAR or JAR file is not self-contained. In this case, use either the `-jardir` or `-cp` option to specify additional JAR or zip files to be added to CLASSPATH variable when the generated files are compiled.

wsdeploy command syntax

The command syntax is noted in the following example:

```
wsdeploy Input_filename Output_filename [options]
```

Required options:

- ***Input_filename***

Specifies the path to the EAR or JAR file to deploy.

- ***Output_filename***

Specifies the path of the deployed EAR or JAR file. If *output_filename* already exists, it is silently overwritten. The *output_filename* can be the same as the *input_filename*.

Other options:

- **`-jardir` *directory***

Specifies a directory that contains JAR or zip files. All JAR and zip files in this directory are added to the CLASSPATH used to compile the generated files. This option can be specified zero or more times.

- **`-cp` *entries***

Specifies entries to add to the CLASSPATH when the generated classes are compiled. Multiple entries are separated the same as they are in the CLASSPATH environment variable, with a semicolon on Windows platforms and a colon for Linux and Unix platforms.

- **`-codegen`**

Specifies to generate but not compile deployment code. This option implicitly specifies the `-keep` option.

- **`-debug`**

Includes debugging information when compiling, that is, use `javac -g` to compile.

- **`-help`**

Displays a help message and exit.

- **`-ignoreerrors`**

Do not stop deployment if validation or compilation errors are encountered.

- **`-keep`**

Do not delete working directories containing generated classes. A message is displayed indicating the name of the working directory that is retained.

- **`-novalidate`**

Do not validate the Web services deployment descriptors in the input file.

- **`-trace`**

Displays processing information, including the names of the generated files.

Example The following example illustrates how the options are used with the **wsdeploy** command:

```
wsdeploy x.ear x_deployed.ear -trace -keep
Processing web service module x_client.jar.
Keeping directory: f:\temp\Base53383.tmp for module: x_client.jar.
Parsing XML file:f:\temp\Base53383.tmp\WarDeploy.wsdl
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java.
Done processing module x_client.jar.
```

Messages

- Flag `-fis` not valid.
Option `f` was not recognized as a valid option.
- Flag `-c` is ambiguous.
Options can be abbreviated, but the abbreviation must be unique. In this case, the **wsdeploy** command cannot determine which option was intended.
- Flag `-c` is missing parameter `-p`.
A required parameter for an option is omitted.
- Missing `p` parameter.
A required option is omitted.

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" in the information center 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* > **Web modules** > *module_instance* > **Web services client bindings**.
For EJB modules, click **Applications** > **Enterprise Applications** > *application_instance* > **EJB 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.

To view this page, click **Applications >Enterprise Applications > application_instance > Web modules > module_instance>Web services client bindings**.

For EJB modules, click **Applications >Enterprise Applications > application_instance > EJB modules > module_instance>Web services client bindings**.

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 > Web modules > module_instance>Web services client bindings > Edit> preferred_port_instance**.

For EJB modules, click **Applications >Enterprise Application > application_instance > EJB 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 > Web modules > module_instance>Web services client bindings > Edit**.

For EJB modules, click **Applications >Enterprise Applications > application_instance > EJB modules > module_instance>Web services client bindings > Edit**.

Port:

Identifies the name of a port.

Request timeout:

Specifies the time, in milliseconds, that the 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.

Configuring the scope of a Web service port

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 bean module, including implementation scope, client bindings and deployment descriptor information. There are three levels of scope that can be set: application, session and request.

Deploy the Web service into WebSphere Application Server.

Web Services for Java 2 platform Enterprise Edition (J2EE) specifies that Web services implementations must be stateless. Therefore, to maintain specification compliance, the scope can remain at the application level because the state relevant to the individual sessions level or the requests level is not supposed to be maintained in the implementation. If you want to deviate from the specification and want to access a different JavaBean instance, because you are looking for information that is located in another JavaBean implementation, the scope settings need to change.

The setting that you configure for the scope determines how frequently a new instance of a service implementation class is created for the Web service ports in a module. Use this task to configure the scope of a Web service port.

You can also configure the scope with the wsadmin tool.

To change the scope setting through the administrative console:

1. Open the administrative console.
2. Click **Applications >Enterprise Applications > application_instance > Web Modules > module_instance>Web Services Implementation Scope**. If you are using an EJB module click **EJB Modules** instead of **Web Modules**.
3. Set the scope to application, session or request. The application scope causes the same instance of the implementation to be used for all requests on the application. The session scope causes the same instance to be used for all requests in each session. The request scope causes a new instance to be

used for every request. For example, with the scope set to application, every message that comes to the server accesses the same JavaBean instance because that is the way the scope settings are configured.

4. Click **Apply**.
5. Click **OK**.

The scope for a Web service port is configured.

Now you can finish any other configurations, start or stop the application, and verify the expected behavior of the Web service.

Web services implementation scope

The scope determines when a new implementation instance is created for Web service ports. For example, setting the scope to `application` causes the same instance of the implementation to be used for each request. Setting the scope to `session` causes the same instance of the implementation to be used for each requests of a session. Setting the scope to `request` causes a new instance to be created for each request.

Use this page to view and manage the scope of the ports of a Web service application.

To view this administrative console page, click **Applications >Enterprise Applications > application_instance > Web modules > module_instance>Web services implementation scope**.

Port:

Specifies a port name for a Web service. A module can contain one or more Web services, each of which can contain one or more ports.

Web service:

Specifies the name of the Web service. A module can contain one or more Web services.

URI:

Specifies the Uniform Resource Identifier (URI) of the binding file that defines the scope. The URI is relative to the Web module.

Scope:

Specifies the scope of a port. The valid values for scope are `request`, `session` and `application`.

Web Services Invocation Framework (WSIF): Enabling Web services

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 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 (EJB)) 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.

If you want to know more about the issues that WSIF addresses, see [Goals of WSIF](#).

If you want to know how WSIF addresses these issues, see [An overview of WSIF](#).

To use WSIF, see the following topics in the information center:

- Using WSIF to invoke Web services.
- WSIF system management and administration.
- WSIF API.

For more information about working with WSIF, visit the Web sites listed in [Web services: Resources for Learning](#).

Goals of WSIF

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: As is explained in Web services are not just SOAP services, 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: 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. 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.

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.

An overview of WSIF

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.

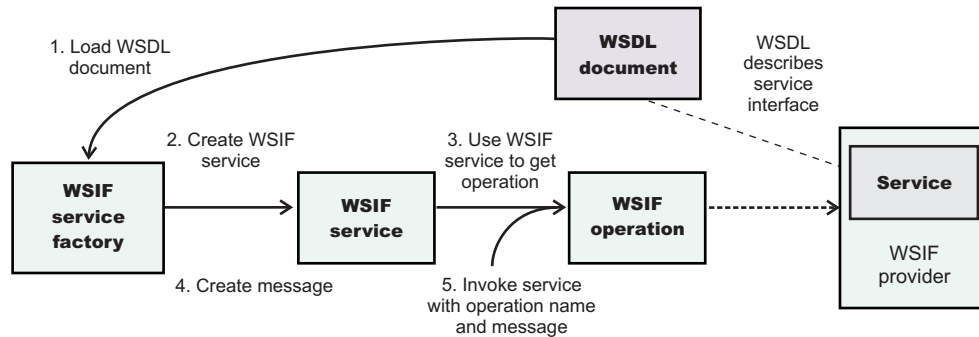
WSIF is designed to work both in an unmanaged environment (stand-alone) and inside a managed container. You can use the Java Naming and Directory Interface (JNDI) to find the WSIF service, or you can use the location described in the WSDL.

For more conceptual information about WSIF and WSDL, see the following topics:

- WSIF and WSDL
- WSIF architecture
- Using WSIF with 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: 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" in the information center.

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" in the information center.

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 "Using the WSIF providers" in the information center.

Using WSIF with 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 on how to configure a client to dynamically select between multiple bindings, see "Developing a WSIF service" in the information center.

WSIF and WSDL: WSDL is the acronym for Web Services Description Language.

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.

For more information on 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: For the Web Services Invocation Framework (WSIF), dynamic invocation means providing the following levels of support when invoking Web services:

1. Support for WSDL extensions and bindings that were not known at build time.
2. Support for Web services that were not known at build time.

WSIF supports (1) through the use of providers.

The providers support (2) by using the WSDL description to access the target service.

Getting started with 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" in the information center 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.
- "Managing the UDDI Registry" in the information center explains how to use the UDDI pages in the WebSphere 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" in the information center 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).

"UDDI Registry troubleshooting" in the information center might be useful if you encounter any problems or unexpected behaviour while using the UDDI Registry, and "UDDI Registry messages" explains any UDDI messages which 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" in the information center introduces some terms with which you might need to be familiar with to use a UDDI Registry.

"UDDI Registry user interface" in the information center tells you 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.

See "Displaying the user interface" in the information center for the URL for the UDDI User Interface.

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 IBM UDDI version 3 Client for Java.

"IBM JAXR Provider for the UDDI Registry" in the information center 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" in the information center might be useful if you encounter any problems or unexpected behaviour while using the UDDI Registry, and "UDDI Registry messages" in the information center explains any UDDI messages which you might see.

Planning to use Web services

This topic discusses how to plan your use of Web services that are developed and implemented based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

Read the "Web services scenario: Overview" in the information center which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how this retailer incorporated the Web services concept.

Web services are Web applications that help you be more flexible in your business processes by integrating with applications that otherwise do not communicate.

Web services reflect the service-oriented architecture approach to programming. This approach is based on the idea of building applications by discovering and implementing network-available services, or by invoking the available applications to accomplish a task. Web services deliver interoperability, for example, Web services applications provide a way for components created in different programming languages to work together as if they were created using the same language. Web services rely on existing transport technologies, such as HTTP, and standard data encoding techniques, such as Extensible Markup Language (XML), for invoking the implementation.

To plan to use Web services:

1. Identify your goals and design Web services to fit your e-business solution. Consider what you want to accomplish by using Web services. Decide how Web services fit into your current topology, applications and programming model. Determine how the Web services process requests on the server and how the clients manage and use the Web service.
2. Design your Web services for reliability, availability, manageability and security. For example, you want your Web services to process a transaction in a reasonable time at all hours of the day and provide users with good security characteristics, such as authentication for buyers. Planning to use Web services to work with WebSphere Application Server helps to meet these requirements.
3. To support Web services, extend WebSphere Application Server to support Web services standards. For interoperable Web services running on platforms supplied by multiple vendors, standards are essential.
4. Decide what development and implementation tools to use. You can use a variety of manual development and implementation tasks. Whether you have an existing Web service to implement or you want to develop your own from a Java bean or from Enterprise JavaBeans (EJB), you can choose different tasks respective to your resources. You can also use Rational Application Developer (RAD) to complete development and implementation tasks.
See Developing Web services for information about developing Web services based on the Java language through the WebSphere Application Server. To read more about RAD see the information center for the product.
5. Install WebSphere Application Server.
6. Review Web services Samples.

You have a design plan for implementing Web services applications into your business architecture.

Develop a Web service.

This topic explains how to develop a Web service using the Web Services for J2EE specification.

Service-oriented architecture

A *service-oriented architecture (SOA)* is a collection of services that communicate with each other, for example, passing data from one service to another or coordinating an activity between one or more services.

Companies want to integrate existing systems to implement Information Technology (IT) support for business processes that cover the entire business value chain. A variety of designs are used, ranging from rigid point-to-point electronic data interchange (EDI) to Web auctions. By using the Internet, companies make their IT systems available to internal departments or external customers, but the interactions are not flexible and are without standardized architecture.

Because of this increasing demand for technologies that support connecting and sharing resources and data, a need exists for a flexible, standardized architecture. SOA is a flexible architecture that unifies business processes by structuring large applications into building blocks, or small modular functional units or services, for different groups of people to use inside and outside the company. The building blocks can be one of three roles: service provider, service broker, or service requestor. See Web services approach to a service-oriented architecture to learn more about these roles.

Requirements for an SOA

To efficiently use an SOA, follow these requirements:

- **Interoperability between different systems and programming languages.**

The most important basis for a simple integration between applications on different platforms is to provide a communication protocol. This protocol is available for most systems and programming languages.

- **Clear and unambiguous description language.**

To use a service offered by a provider, it is not only necessary to be able to access the provider system, but the syntax of the service interface must also be clearly defined in a platform-independent fashion.

- **Retrieval of the service.**

To support a convenient integration at design time or even system run time, a search mechanism is required to retrieve suitable services. Classify these services as *computer-accessible*, *hierarchical* or *taxonomies* based on what the services in each category do and how they can be invoked.

Web services approach to a service-oriented architecture

The Web services approach implements a service-oriented architecture (SOA). A major focus of Web services is to make functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages. These services can be new applications or just wrapped around existing legacy systems to make them network-enabled. A service can rely on another service to achieve its goals.

Each SOA building block can assume one or more of three roles:

- **Service provider**

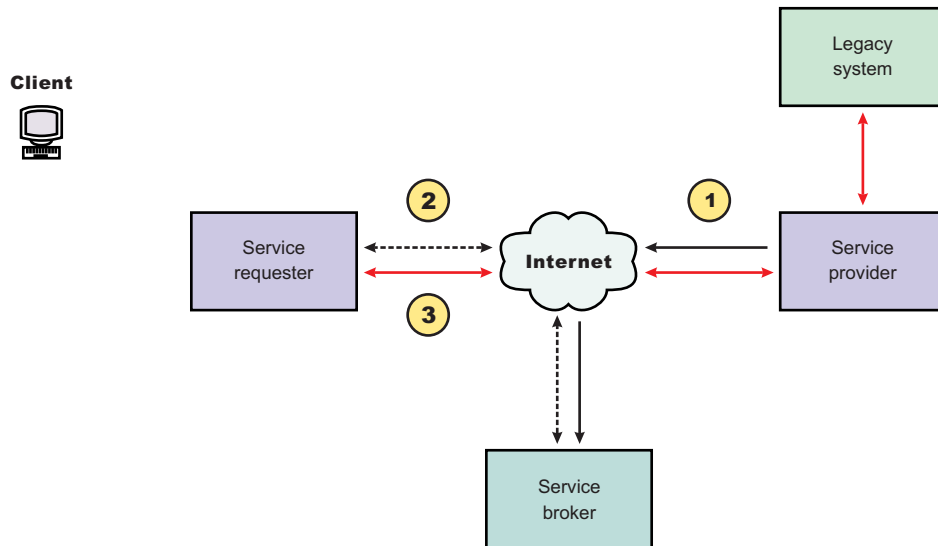
The service provider creates a Web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or how to exploit free services for other value. The provider also has to decide which category to list the service in for a given broker service and what sort of trading partner agreements are required to use the service.

- **Service broker**

The service broker, also known as *service registry*, is responsible for making the Web service interface and implementation access information available to any potential service requestor. The implementer of the broker decides the scope of the broker. Public brokers are available through the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, some decisions need to be made about the amount of the offered information. Some brokers specialize in many listings. Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. Some brokers catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, the number of listings or the accuracy of the listings. The Universal Description, Discovery and Integration (UDDI) specification defines a way to publish and discover information about Web services.

- **Service requester**

The service requestor or Web service client locates entries in the broker registry using various find operations and then binds to the service provider to invoke one of its Web services.



Characteristics of the SOA

The presented SOA illustrates a loose coupling between the participants, which provides greater flexibility in the following ways:

- A client is coupled to a service. Therefore, the integration of the server takes place outside the scope of the client application programs.
- Old and new functional blocks or applications and systems, are encapsulated into components that work as services.
- Functional components and their interfaces are separate, so that new interfaces can be plugged in more easily.
- Within complex applications, the control of business processes can be isolated. A business rule engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.
- Services can be incorporated dynamically during run time.
- Bindings are specified using configuration files and can be easily adapted to new needs.

Properties of a service-oriented architecture

The service-oriented architecture offers the following properties:

- **Web services are self-contained.**

On the client side, no additional software is required. A programming language with Extensible Markup Language (XML) and HTTP client support is enough to get you started. On the server side, a Web server and a SOAP server are required. It is possible to enable an existing application for Web services without writing a single line of code.

- **Web services are self-describing.**

Neither the client nor the server knows or cares about anything besides the format and content of the request and response messages (loosely coupled application integration). The definition of the message format travels with the message; no external metadata repositories or code generation tool are required.

- **Web services can be published, located, and invoked across the Internet.**

This technology uses established lightweight Internet standards such as HTTP and it leverages the existing infrastructure. Some other standards that are required include, SOAP, Web Services Description Language (WSDL), and UDDI.

- **Web services are language-independent and interoperable.**

Client and server can be implemented in different environments. Existing code does not have to change in order to be Web services enabled.

- **Web services are inherently open and standard-based.**

XML and HTTP are the major technical foundation for Web services. A large part of the Web service technology has been built using open-source projects.

- **Web services are dynamic.**

Dynamic e-business can become reality using Web services because with UDDI and WSDL you can automate the Web service description and discovery.

- **Web services are composable.**

Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This chaining shortens development time and enables best-of-breed implementations.

- **Web services are loosely coupled.**

Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that supports a more flexible reconfiguration for an integration of the services.

- **Web services provide programmatic access.**

The approach provides no graphical user interface; it operates at the code level. Service consumers need to know the interfaces to Web services, but do not need to know the implementation details of services.

- **Web services provide the ability to wrap existing applications.**

Already existing stand-alone applications can easily integrate into the service-oriented architecture by implementing a Web service as an interface.

Web services business models supported

The properties and benefits of using a service-oriented architecture (SOA) such as Web services is well suited for binding small modules that perform independent tasks within a highly heterogeneous e-business model. Web services can be easily wrapped around existing applications in your business model and plugged into different business processes.

For connecting to a large monolithic system that does not support the implementation of different flexible business processes, other approaches might be better suited, for example, to satisfy specialized features, such as performance or security.

The following business models are easily implemented by using an architecture including Web services:

- **Business information**

Sharing of information with consumers or other businesses. Web services can be used to expand the reach through such services as news streams, local weather reports, integrated travel planning, and intelligent agents.

- **Business integration**

Providing transactional, fee-based services for customers. A global network of suppliers can be easily created. Web services can be implemented in auctions, e-marketplaces, and reservation systems.

- **Business process externalization**

Web services can be used to model value chains by dynamically integrating processes to a new solution within an organizational unit or even with those of other e-businesses. This modeling can be achieved by dynamically linking internal applications to new partners and suppliers, to offer their services to complement internal services.

Setting up and deploying a new UDDI Registry

Start the installation of WebSphere Application Server Version 6 and create the profile for your application server (for example, server1) to be used to host UDDI.

You have a choice of deploying either a default UDDI node, or a user customized UDDI node.

A default UDDI node would be a suitable option for initial evaluation of UDDI and for development and test purposes. 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 user customized UDDI node, you create the UDDI database and datasource to your own specifications, and then use the `uddiDeploy.jacl` script to deploy the UDDI application.

The main difference between *default* and *user customized*, in the context of these set up tasks, refers to a number of vital UDDI properties. For a default set up these vital properties are automatically set to default values and are not changeable by the user. For a user customizable set up, the user is given an opportunity to set these vital properties, but once set they can not be changed for this configuration.

Important: Important Note for DB2 users - Starting the WebSphere Application Server in which the UDDI Registry is deployed

On Unix and Linux platforms, before you start the WebSphere server that is hosting your UDDI DB2 application, you must run the `db2profile` script which is found in the home directory for you db2 userid, for example `/home/db2inst1/sqllib/db2profile`

Similarly when in an ND configuration and starting the server via the Deployment Manager, you must ensure the `db2profile` is run before starting the relevant nodeagent.

The information above is true during initial set up and deployment of UDDI and also every time you start the application server or node agent. For this reason it is strongly recommended that you update the user profile for the user that will start the nodeagent and server to also execute the `db2profile`.

If running the profile manually type:

```
. /home/db2inst1/sqllib/db2profile
```

In the above example, notice that the '.' is followed by a single space character.

Deploying into a Network Deployment cell (but not a cluster)

It is important to note that the `uddiDeploy.jacl` script must be run with the Deployment Manager as the target.

If you are deploying into a network deployment cell you cannot create a default Cloudscape node using `uddiDeploy.jacl`. You may, however, manually create the default Cloudscape database (with a default profile) by following the instructions in [Creating a Cloudscape database and adding the parameter 'DEFAULT'](#) as the last argument.

You will not be able to use the default option on `uddiRemove` for a UDDI node that is deployed into an application server which is part of a Network Deployment cell. See ["Removing a UDDI Registry node"](#) for more information.

If you have a non default UDDI setup in a base application server, you can issue an `addNode -includeapps` command which will add the necessary definitions into the deployment manager.

Deploying into a cluster within a Network Deployment cell

It is important to note that `uddiDeploy.jacl` and `uddiRemove.jacl` can not be used in a cluster environment.

It is assumed a single database will be used for all members of the cluster.

You can follow the same guidelines for a non cluster set up in general for the resources such as the JDBC provider and datasource as described in this Information Center, but they may need updating on individual cluster members to correctly access the shared database for example.

The options that are available are:

- Deploy UDDI into a standalone server , as described in this Information Center (using `uddiDeploy`), and then create a cluster using that server as a template for the other members.
- If using an cluster that already exists, you can use the admin console or `wsadmin` for example (and not `uddiDeploy.jacl`), to deploy the `uddi.ear` across the cluster members, but follow the additional advice in the main instructions.

Changing from a base application server to a Network Deployment cell.

It is possible to move from a base application server to a network deployment cell using the standard `addNode` command. During the move, any applications may be lost unless you use the `-includeapps` option. This applies to all applications and not just UDDI applications. See the `addNode` command for details. This applies to a default or a user customized UDDI node.

Moving from a default UDDI node to a user customized UDDI node.

After testing the UDDI deployment in a default UDDI node, you can move to a user customized node by recreating the database using the instructions in [Setting up a customized UDDI node](#), but you must be aware that any data saved in the default node (both policies, properties and user data) will be lost in the move.

Moving between Cloudscape, DB2 and Oracle databases.

If you decide to move between one type of database to another, the datasource of the old database will still have a JNDI name of `datasources/uddids`. You must either rename this JNDI name to something different, or delete the datasource before you define the new database, create the new datasource and initialize the database.

You now have the choice of a default UDDI node, or a user customized UDDI node.

Setting up a customized UDDI node

You can set up a customized UDDI node by completing the following steps:

1. Create a database schema to hold the UDDI registry by executing one of the following:
 - Creating a DB2 database
 - Creating an Oracle database
 - Creating a Cloudscape database

Note: You must not run the final step to set the default node indicator in the database.

2. Create a J2C Authentication Data Entry (not required for Cloudscape):
 - a. Expand **Global Security** and **JAAS Configuration** (on the right), then click on **J2C Authentication Data**
 - b. Select **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, which is used to read and write to the UDDI registry database.

Password

The password associated with the userid specified above.

Description

a suitable description to describe the chosen userid.

Click 'Apply' and then Save the changes to the master configuration

3. Create a JDBC Provider (if a suitable one does not already exist):
 - For DB2, select the options **DB2 Legacy CLI-based Type 2 JDBC Driver** and **Connection Pool datasource**.
 - For Oracle, select the options **Oracle JDBC Driver** and **Connection Pool datasource**.
 - For Cloudscape, select the **Cloudscape JDBC Driver** and **Connection Pool datasource**.

For details on how to create a JDBC provider, see *Creating and configuring a JDBC provider using the administrative console*.

4. Create a datasource for the UDDI Registry by following these steps:
 - a. Expand **Resource** and **JDBC Providers**
 - b. Select the desired 'scope' of the JDBC provider you selected or created earlier. For example, select:
Server: yourservername
This displays 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 Version 4 option)
 - e. Select 'New' to create a new datasource
 - f. Fill in the details for the datasource as follows:

Name a suitable name, such as UDDI Datasource

JNDI name

set to **datasources/uddids** - this value is obligatory.

Note: You must not have any other datasources using this JNDI name. If you have another datasource using this JNDI name, then you must either remove it or change its JNDI name. For example, if you have previously created a default UDDI node using Cloudscape, you should use the `uddiRemove.jacl` script with the default option to remove the datasource and the UDDI application instance, before continuing.

Description

a suitable description

Category

set to `uddi`

Data store helper class name

filled in for you as:

- for **DB2** - `com.ibm.websphere.rsadapter.DB2DataStoreHelper`
- for **Oracle 9i** - `com.ibm.websphere.rsadapter.OracleDataStoreHelper`
- for **Oracle 10g** - `com.ibm.websphere.rsadapter.Oracle10gDataStoreHelper`
- for **Cloudscape** - `com.ibm.websphere.rsadapter.CloudscapeDataStoreHelper`

Relational Database Management System data source properties

- for **DB2 - Database name** - for example:

UDDI30

- for **Oracle - URL** - for example:
jdbc:oracle:oci8:@<Oracle database name>
- for **Cloudscape - Database name** - for example:
\${USER_INSTALL_ROOT}/databases/com.ibm.uddi/UDDI30

Component-managed authentication alias

- for **DB2**, select the alias that you created in step 3 from the pulldown. It will have the node name appended in front of it, for example MyNode/UDDIAlias.
- for **Oracle**, select the alias that you created in step 3 from the pulldown. It will have the node name appended in front of it, for example MyNode/UDDIAlias.
- for **Cloudscape** leave this set to (none).

Container-managed authentication alias

Set to DefaultPrincipalMapping

Mapping-configuration alias

Set to (none)

- g. Use this Data Source in container-managed persistence (CMP), and ensure it is unchecked.
5. Test the connection to your UDDI database by clicking on the "Test Connection" button for the datasource. 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.
6. Deploy the UDDI Registry into the server by running the uddiDeploy.jacl command from a command prompt as shown.

This file is located in

WAS_HOME/bin

The syntax of the command is:

```
wsadmin [-conntype none] -f uddiDeploy.jacl <node> <server>
```

where:

-conntype none

is optional, and is only needed if the application server is not running.

<node>

the name of the WebSphere node on which the target server runs. Note that the node name is case sensitive.

<server>

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.

You are recommended to deploy the UDDI application using the uddiDeploy.jacl, but note that you can also use the administrative console to deploy the application in the normal way. If you use the administrative console you must ensure that the Classloader Mode for the application is set to PARENT_LAST, and that the WAR class loader Policy is set to Application. The uddiDeploy.jacl command in a command prompt will do this for you.

For example, to deploy UDDI on node 'MyNode' and server 'server1' on a Windows system you might enter the following (assuming that server1 is already started):

```
wsadmin -f uddiDeploy.jacl MyNode server1
```

You should now start the UDDI application, or start the application server if it is not already running.

As you have chosen a user customized UDDI node, you will need to set up the properties for the UDDI node using UDDI administration, and initialize the node before it is ready to accept UDDI requests (see "Configuring the UDDI Registry Application" in the information center for details).

Setting up a default UDDI node

There are two ways to setup a default UDDI node:

- Either by executing the `uddiDeploy.jacl` script and specifying the **default** option. This creates a running default UDDI node within a Cloudscape database, or
- Executing an additional step after you have created your database.

Run one of the following

1. **Optional: For a default Cloudscape UDDI node:**

For a default Cloudscape configuration Network Deployment read the section 'Installing into a Network Deployment Cell' first.

Deploy the UDDI Registry by running the `uddiDeploy.jacl` script from a command prompt as shown.

This file is located in

`WAS_HOME/bin`

The syntax of the command is (shown here on 2 lines for publication):

```
wsadmin [-conntype none] -wsadmin_classpath WAS_HOME/cloudscape/lib -f
uddiDeploy.jacl <node> <server> default
```

where:

-conntype none

is optional, and is only needed if the application server is not running.

WAS_HOME

is the directory name of the WebSphere Application Server install location

<node>

is the name of the WebSphere node on which the target server runs. Note that the node name is case sensitive.

<server>

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.

default

using the 'default' option causes the command to create a UDDI node, with default policies within a Cloudscape database and datasource. This is a special case only for Cloudscape and creates everything required to run a UDDI node, in a standalone application server.

For example, to create a UDDI node called 'MyNode' on server 'server1' on a Windows system, you might enter the following (this assumes `server1` is started). The command is shown on 2 lines for publication.

```
wsadmin -wsadmin_classpath C:\Progra~1\IBM\WebSphere\AppServer\cloudscape\lib
-f uddiDeploy.jacl MyNode server1 default
```

If the server is not started the command is (shown here on 2 lines for publication):

```
wsadmin -conntype none -wsadmin_classpath C:\Progra~1\IBM\WebSphere\AppServer\cloudscape\lib
-f uddiDeploy.jacl MyNode server1 default
```

(Note that these should be entered as one command on a single line)

You should now start the UDDI application, or start the application server if it is not already running.

2. **Optional: For a default DB2, default Cloudscape or default Oracle UDDI node:**

DB2, Cloudscape or Oracle may be used for a single application server installation or a Network Deployment installation.

- a. Initially you need to define some resources and run some scripts to set up the database. Go to [Creating the DB2 database](#), [Creating the Cloudscape database](#) or [Creating the Oracle database](#) to create and load the database, ensuring you have run the final step to set the default node indicator in the database, and return to this point. You must create a JDBC Provider if a suitable one does not already exist or select one, and a Datasource to reference the UDDI database.

- For DB2, select the 'DB2 Legacy CLI-based Type 2 JDBC Driver' option and Connection Pool datasource option.
 - When using Oracle ensure you select 'Oracle JDBC Driver' and the Connection Pool datasource option.
 - For Cloudscape, select the supplied **Cloudscape JDBC Driver**
 - Refer to Creating and configuring a JDBC provider using the administrative console for details on how to create a JDBC Provider.
- b. Create a J2C Authentication Data Entry for DB2 or Oracle access (not required for Cloudscape) using the WebSphere Application Server administrative console by following these steps:
- 1) Expand 'Global Security' and 'JAAS Configuration' and click on 'J2C Authentication Data Entries'
 - 2) Select 'New' to create a new J2C authentication data entry
 - 3) Fill in the details as follows:

Alias a suitable (short) name, such as UDDIAlias

Userid

your DB2 userid, such as db2admin, or your Oracle userid, such as SYSTEM.

Password

The password associated with your userid.

Description

a suitable description

Click 'Apply' and then Save the changes to the master configuration

- c. Create a datasource for the UDDI Registry by following these steps:

- 1) Expand 'Resource' and 'JDBC Providers'
- 2) Select the desired 'scope' of the JDBC provider created earlier. For example, select:
Server: yourservername
to show the JDBC providers at the server level.
- 3) Select your DB2 or Oracle JDBC provider as selected or created earlier
- 4) Under 'Additional Properties', select 'Data Sources' (**not** the Data Sources Version 4 option)
- 5) Select 'New' to create a new datasource
- 6) Fill in the details for the datasource as follows:

Name a suitable name, such as UDDI Datasource

JNDI name

set to **datasources/uddids** - this value is obligatory.

Note: Note that you must not have any other datasources using this JNDI name. If you have another datasource using this JNDI name, then you must either remove it or change its JNDI name. For example, if you have previously created a default UDDI node using Cloudscape, you should use the uddiRemove.jacl script with the default option to remove the datasource and the UDDI application instance before continuing.

Description

a suitable description

Category

set to uddi

Data store helper class name

filled in for you as:

- for **DB2** - com.ibm.websphere.rsadapter.DB2DataStoreHelper

- for **Oracle 9i** - com.ibm.websphere.rsadapter.OracleDataStoreHelper
- for **Oracle 10g** - com.ibm.websphere.rsadapter.Oracle10gDataStoreHelper
- for **Cloudscape** - com.ibm.websphere.rsadapter.CloudscapeDataStoreHelper

Relational Database Management System data source properties

- for **DB2 - Database name** - for example:
UDDI30
- for **Oracle - URL** - for example:
jdbc:oracle:oci8:@<Oracle database name>
- for **Cloudscape - Database name** - for example:
\${USER_INSTALL_ROOT}/databases/com.ibm.uddi/UDDI30

Component-managed authentication alias

- for **DB2**, select the alias that you created in step 3 from the pulldown. It will have the node name appended in front of it, for example MyNode/UDDIAlias.
- for **Oracle**, select the alias that you created in step 3 from the pulldown. It will have the node name appended in front of it, for example MyNode/UDDIAlias.
- for **Cloudscape** leave this set to (none).

Container-managed authentication alias

select the alias that you created earlier from the pulldown. It will have the node name appended in front of it, for example MyNode/UDDIAlias.

Mapping-configuration alias

set to DefaultPrincipleMapping from the pulldown (this might already be filled in).

Leave all other fields unchanged.

Click 'Apply' and Save the changes to the master configuration.

- d. Test the connection to your UDDI database by clicking on the "Test Connection" button for the datasource. 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.
- e. Deploy the UDDI Registry into the server by running the uddiDeploy.jacl script from a command prompt as shown.

This file is located in

WAS_HOME/bin

The syntax of the command is:

```
wsadmin [-conntype none] -f uddiDeploy.jacl <node> <server>
```

where:

-conntype none

is optional, and is only needed if the application server is not running.

<node>

the name of the WebSphere node on which the target server runs. Note that the node name is case sensitive.

<server>

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.

You are recommended to deploy the UDDI application using the uddiDeploy.jacl, but note that you can also use the administrative console to deploy the application in the normal way. If you use the administrative console you must ensure that the Classloader Mode for the application is set to PARENT_LAST, and that the WAR class loader Policy is set to Application. The uddiDeploy.jacl command in a command prompt will do this for you.

For example, to deploy UDDI on node 'MyNode' and server 'server1' on a Windows system you might enter the following (assuming that server1 is already started):

```
wsadmin -f uddiDeploy.jacl MyNode server1
```

You should now start the UDDI application, or start the application server if it is not already running.

As you have chosen to use a default UDDI node, it will be initialized when the UDDI application is started for the first time.

Creating a DB2 database

Perform this task if you want to use DB2 as the database store for your UDDI Registry data. You only need do this once for each UDDI Registry, as part of Setting up and deploying a UDDI Registry.

Before you begin

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.

To create the DB2 database follow the steps shown below:

1. Run the following commands to setup the DB2 environment variables from the DB2 Command Line Processor (that is, at a prompt which requires db2 to be entered before each command):
 - a. Not required for DB2 Version 8 and onwards: db2set DB2_INDEX_2BYTEVARLEN=YES
 - b. db2set DB2CODEPAGE=1208
2. Create the DB2 database by entering the following command from the DB2 Command Line Processor:
 - a. db2 create database <DataBaseName> using codeset UTF-8 territory en where <DataBaseName> is the name of the database being created.
3. Configure the DB2 database by entering the following commands from the DB2 Command Line Processor:
 - a. db2 connect to <DataBaseName> user <DB2UserID> using <DB2Password>
 - b. db2 update db cfg for <DataBaseName> using applheapsz 2048
 - c. db2 update db cfg for <DataBaseName> using logfilsiz 8192
 - d. db2 connect reset
 - e. db2 terminate

4. Create additional database structures by entering the following commands from the DB2 Command Line Processor:
 - a. `db2 connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
 - b. `db2 create bufferpool <BufferPoolName> size 250 pagesize 32K`
 - c. `db2 connect reset`
 - d. `db2 terminate`
 - e. `db2 force application all`
 - f. `db2 terminate`
 - g. `db2stop`
 - h. `db2start`
5. Create further database structures by entering the following commands from the DB2 Command Line Processor:
 - a. `db2 connect to <DataBaseName> user <DB2UserID> using <DB2Password>`
 - b. `db2 create regular tablespace uddits pagesize 32K managed by system using ('<TableSpaceName>') extentsize 64 prefetchsize 32 bufferpool <BufferPoolName>`
 - c. `db2 create system temporary tablespace <TempTableSpacename> pagesize 32K managed by system using ('<TempTableSpacename>') extentsize 32 overhead 14.06 prefetchsize 32 transferrate 0.33 bufferpool <BufferPoolName>`
6. Enter the following commands exactly as shown (noting in particular one step uses `-vf` rather than `-tvf`) into a DB2 Command Window to define the database structures needed to store the UDDI data (before running these commands change dirstory to `WAS_HOME/UDDIReg/databaseScripts`):
 - 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`
7. This last step should only be run if you want your database to be used as a default UDDI node.
 - a. Enter the following command from the DB2 Command Line Window: `db2 -tvf uddi30crt_70_insert_default_database_indicator.sql`

Continue with setting up and deploying your UDDI Registry node.

Creating an Oracle database

Perform this task if you want to use Oracle as the database store 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.

Note: Only Version 9i¹ and Oracle 10g²

1.

Restrictions:

discoveryURL (Business) maximum 4000 bytes, UDDI specification 4096 characters; accessPoint (bindingTemplate) maximum 4000 bytes, UDDI Specification 4096 characters; instacneParms (tModellInstanceInfo) maximum 4000 bytes, UDDI specification 8192 characters; overviewURL (tModellInstanceInfo) maximum 4000 bytes, UDDI specification 4096 characters; Digital Signature maximum 4000 bytes.

Before you begin

Note that this task will create two new schemas, `ibmudi30` and `ibmuds30`. You will be unable to complete this task if you already have existing schemas of those 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, and suggested values 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 two following actions depending on your level of Oracle:
 - For Version 10g and later:
`sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_25_tables_oracle.sql`
 - For Oracle 9i:
`sqlplus <OracleUserID>/<OraclePassword> @ uddi30crt_25_tables_oracle_pre10g.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 Cloudscape database

Perform this task if you want to use Cloudscape as the database store for your UDDI Registry, and you do not want to use the default option on `uddiDeploy.jacl` (see 'Setting up a default UDDI node'). The most likely reasons for not using the default option on `uddiDeploy.jacl` are that you want to set up a customized UDDI node, or that you are deploying UDDI into a Network Deployment cell. You need only perform this task once for each UDDI Registry, as part of the Setting up and deploying a UDDI Registry.

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 install will be `WAS_HOME/UDDIReg/databasescripts`

arg2 is the path to the location where you want to install the Cloudscape database, such as `WAS_HOME/databases/com.ibm.uddi`

2.

Restrictions:

`discoveryURL` (business) maximum 4000 bytes, UDDI specification 4096 characters.

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 some other 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.

Run the following java -jar command to create a UDDI Cloudscape database using UDDICloudscapeCreate.jar, which is created in WAS_HOME/Lib directory.

1. On Solaris and HP-UX platforms:

```
java -Djava.ext.dirs=WAS_HOME/cloudscape/lib:WAS_HOME/java/lib/ext -jar
    UDDICloudscapeCreate.jar arg1 arg2 arg3 arg4
```

2. On all other platforms:

```
java -cp WAS_HOME/cloudscape/lib/db2j.jar -jar UDDICloudscapeCreate.jar arg1 arg2 arg3 arg4
```

Continue with setting up and deploying your UDDI Registry node.

Using a remote database for the UDDI Registry

It is possible for the UDDI Registry database to be hosted on a separate system (remote system) from the system on which the UDDI Registry application is deployed.

This is achieved using standard database capabilities of the database product used for the UDDI Registry database. You should refer to documentation for the database product if you are not familiar with these capabilities. Some considerations specific to each database product are:

Remote DB2

Create a DB2 UDDI database on the remote system, and use the DB2 Client to create an alias to reference it. Use the alias name as the Database name in the UDDI datasource.

Remote Oracle

Create the Oracle tables used for an Oracle UDDI database on the remote system, and use the URL property of the UDDI datasource to reference the remote Oracle instance.

Networked Cloudscape

Create a Cloudscape UDDI database on the remote system, and use the Cloudscape Network Server using Universal data source properties (Database name, Server name and Port number) of the UDDI datasource to reference the remote Cloudscape database.

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.

Open a browser window and enter the URL to access the UDDI Registry User Interface. Under the 'Quick Find' heading on the Find tab, select the 'Business' radio button and enter '%' in the box. Click the 'Find' button, and you should see the business displayed in the detail frame which will be this UDDI node's business entity. You can click on the listed business in order to see its detail.

If you see a business listed in the detail frame, your UDDI Registry has been deployed successfully.

As a further IVP, 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.

Developing Web services applications

This topic explains how to develop a Web service using 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 .

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) 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 JSR-101 and JSR-109 programming models 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.

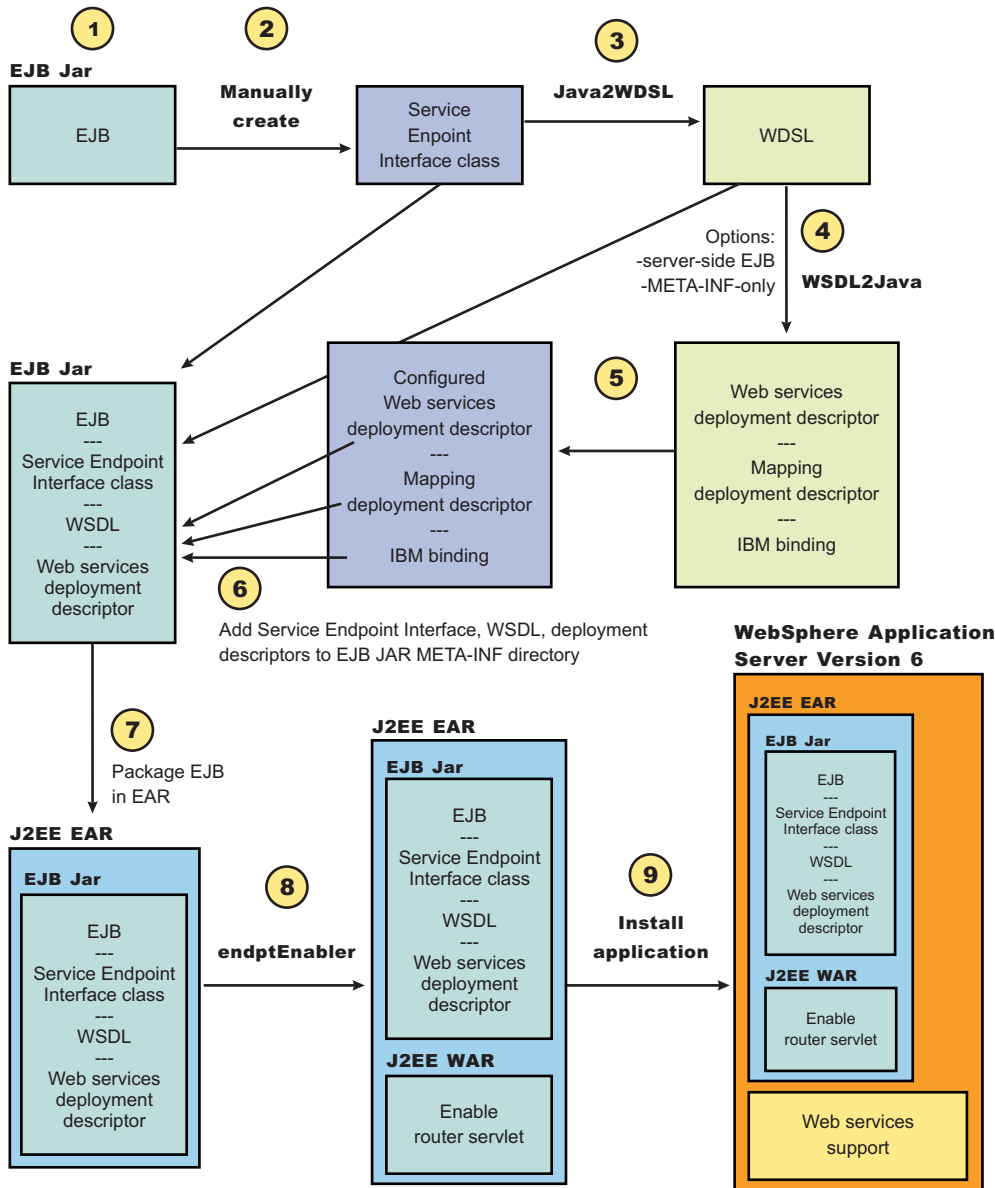
6. Assemble an enterprise archive (EAR) file from a JAR file or assemble an EAR file from a WAR file.

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.



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 services specifying 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. You can review the JAX-RPC specification through [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	<code>wsdl:portType</code>

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="request"/>

<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</c>
  </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</a>
</soap:body>

```

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. It is orthogonal to the security support 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. See "Secure Sockets Layer" in the *Securing applications and their environment* PDF for more information.

Web services applications can also use Federal Information Processing Standard (FIPS) approved ciphers for more secure TLS connections. For information on FIPS, refer to the *Securing applications and their environment* PDF.

WebSphere Application Server uses the Java Secure Sockets Extension (JSSE) package to support SSL and TLS.

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 include:

- The `REQUEST_TRANSPORT_PROPERTIES` property and `RESPONSE_TRANSPORT_PROPERTIES` property 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.java.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.java.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.

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 will contain the HTTP header identifiers and the associated values.
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 to retrieve transport headers, you can retrieve information from HTTP headers from incoming responses.

You need a Web services client that you can enable to retrieve HTTP transport headers.

Retrieving 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 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 will contain the HTTP header identifiers values to retrieve.
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 one 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 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.
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 RESPONSE_TRANSPORT_PROPERTIES property to null, the header does not retrieve from the response headers. 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 indicated in the HashMap are copied into a new instance of a HashMap for each method invocation.

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.
5. Retrieve the new HashMap instance that contains the HTTP header identifiers and the associated values from the Stub using the `com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES` property.

You have a Web service that is able to receive HTTP transport 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. This HashMap is used to create a new HashMap instance that contains these keys. The values are filled in the new 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> The connection header is ignored for HTTP 1.0. When using HTTP 1.1, the following values are set: <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> The expect header is ignored for HTTP 1.0. When using HTTP 1.1, the following values are set: <ul style="list-style-type: none"> 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.echString("Foo");
HashMap checkReceivedHeaders = (HashMap) ((Stub)
binding)._getProperty(Constants.RESPONSE_TRANSPORT_PROPERTIES);
```

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 `-serviceName` 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://localhost: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 `complexType`s. 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.

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.

Command-line syntax

The command-line syntax is:

```
WSDL2Java [arguments] WSDL-URI
```

- **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.

For more information about the XML types that are defined by the JAX-RPC specification, you can review section 4 of the JAX-RPC specification. A link to the specification is available in [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 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.xml` 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 three ways that you can configure the HTTP outbound transport level security for a Web service acting as a client to another Web service client. You can also configure the HTTP outbound transport level security with an assembly tool or by using the Java properties.

Configuring the HTTP outbound transport-level security for a Web service is based on the Secured Sockets Layer (SSL) configuration repertoires of the WebSphere Application Server. Review "Configuring Secure Sockets Layer" in the information center for details.

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 Java properties, the properties are configured as system properties. However, the configuration that is specified in the binding takes precedence over the Java properties.

Configure the HTTP outbound transport level security with the following steps provided in this task section.

Open the administrative console.

1. Click **Applications > Enterprise Applications > *application_instance* > Web Modules or EJB Modules > *module_instance* > Web Services Client Security Bindings.**
2. Click **HTTP SSL Configuration** to access the HTTP SSL configuration panel. Select **HTTP SSL enabled**. Select the SSL configuration from the list in the HTTP Basic Authentication panel.

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_name*.**
2. Under Related Items, click **Web module > *URI_file_name* > Web Services: Client Security Bindings.**
3. Under HTTP SSL Configuration, click **Edit.**

HTTP SSL enabled:

Specifies secure socket communications for the HTTP transport for this port. When enabled, WebSphere Application Server uses the HTTP SSL Configuration setting.

HTTP SSL configuration:

Specifies which alias of the SSL configuration to use with the HTTP transport for this port.

This option is used if you select **HTTP SSL Enabled**. SSL aliases are defined in the Secure Sockets Layer configuration repertoire, which you can configure by clicking **Security > SSL**.

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.

You must configure the assembly tool before you can use it.

This task is one of three ways that you can configure the HTTP outbound transport level security for a Web Service acting as a client to another Web service. You can also configure the HTTP outbound transport level security with the administrative console or by using the Java properties.

The configuration of HTTP outbound transport-level security for a Web service is based on the Secured Sockets Layer (SSL) configuration repertoires of the WebSphere Application Server. Review "Configuring Secure Sockets Layer" in the information center for details.

If you choose to 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 Java properties, the properties are configured as system properties. However, the configuration specified in the binding takes precedence over the Java properties.

Configure the HTTP outbound transport level security with the following steps provided in this task section.

1. Start an assembly tool. 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. Configure the HTTP outbound transport level security 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 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 "Using Java Secure Socket Extension and Java Cryptography Extension with servlets and enterprise bean files" 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.

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.

1. Open the administrative console.
 - a. Click **Applications** > **Enterprise Applications** > *application_instance* > **Web Modules** or **EJB Modules** > *module_instance* > **Web Services Client Bindings**.
 - b. Click **HTTP Basic Authentication** to access the HTTP basic authentication panel. Enter the values in the HTTP Basic Authentication panel.
2. Click **Applications** > **Enterprise Applications** > *application_instance*. Under Additional Properties, click **Publish WSDL files** which brings you to the **Publish WSDL zip files** panel.

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_name*.
2. Under Related Items, click **Web module** > *URI_file_name* > **Web Services: Client Security Bindings**.
3. Under HTTP Basic Authentication, click **Edit**.

Basic authentication ID:

Specifies the user name for the HTTP basic authentication for this port.

Basic authentication password:

Specifies the password for the HTTP basic authentication for this port.

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.

You must configure the assembly tool before you can use it.

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. 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. 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 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 > Process Definition > Control > Java Virtual Machine > Custom Properties** to define the property in the Control, or click **Application Server > server > Process Definition > Servant > Java Virtual Machine > Custom Properties** to define the property in the Servant.
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.

You must configure the assembly tool before you can use it.

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 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. Start an assembly tool. 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. 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 wsadmin

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" in the information center.

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 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 connections 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.

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" in the information center 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 <code>java.naming.factory.initial</code> property.
jndiProviderURL	Specifies the JNDI provider URL, which is mapped to the <code>java.naming.provider.url</code> 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.

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>&<property>=<value>&...
```

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.

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. You can also use assembly tools and the Rational Application Developer. 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 unmanaged client run environment by following the listed actions:

1. Develop thin application client code on a server machine.
The thin application client provides the necessary run-time to support the communication needs between the client and the server.
2. Run the **setupCmdLine.bat** command if you are using a Windows platform.
Or, you can set **WAS_USER_SCRIPT** to `instance_root\bin\setupCmdLine.bat`, which has the same effect as running the **setupCmdLine** command.

Linux and UNIX-based platforms **source** the script to the parent shell to inherit the exported variables by running this command:

```
. /setupCmdLine.bat (there is a space between the period and the slash)
```

3. Configure the path. You can add the WebSphere and Java bin directories to your path by typing:

On Windows platforms:

```
set PATH=%WAS_PATH%;%PATH%
```

On UNIX:

```
export PATH=$WAS_PATH:$PATH
```

You have set up a development and unmanaged client run-time environment so that you can develop Web services.

Develop Web services. This topic is a good starting point in learning about how to develop a J2EE Web service.

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.xmi` 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 **javac** commands for the Windows platform, and the Linux and Unix platforms in the Developing thin application client code topic, to compile the interface. Use the name of the service endpoint interface class in the **javac** command for the class to compile.

You have developed a service endpoint interface that you can use to develop Web services.

This example uses an AddressBook Java interface. 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 seimpl 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, *bindingImpl.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 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 containing 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 **javac** commands for Windows platform, and the Linux and Unix platforms that are listed in the Developing thin application client code topic to compile the interface. In the **javac** command, use the name of the service endpoint interface class for the class to compile.

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 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 wsdIURL** 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 `portTypeHome.java` EJB home interface template. If necessary, modify the template.
3. Edit the `bindingImpl.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 or UNIX 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 wsdIURL** 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 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 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.

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 wsd/URL** 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.

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. (Optional) Configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor. Complete this step if you are deploying a managed client that runs in the J2EE client container and you want to override the

default client settings. See `ibm-webservicesclient-bnd.xmi` assembly properties for more information about the `ibm-webservicesclient-bnd.xmi` deployment descriptor.

7. 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.

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.

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.

Configure the `ibm-webservicesclient-bnd.xml` 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.

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.xml.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-ext.xml.
```


Complete the client implementation.

Assemble a Web services-enabled client JAR and EAR file.

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 is 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 IBM 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

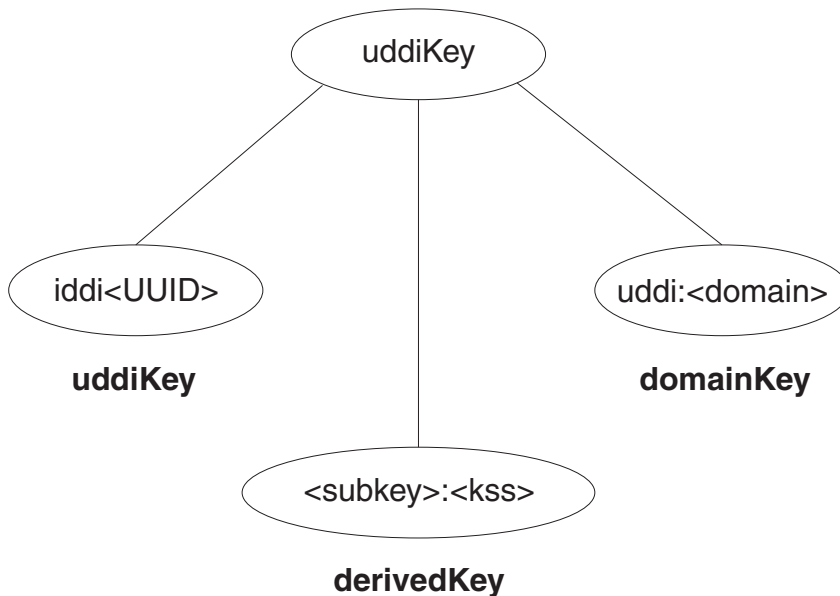
Entity keys are identifiers used to address entities within a UDDI registry. Each entity; `businessEntity`, `businessService`, `bindingTemplate`, and `tModel` (also `Subscriptions`), has a unique identifier generated or assigned when first published in the UDDI registry. Within a particular registry, 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 IBM 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.

A UDDI Version 3 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).

The public UDDI Universal Business Registry (UBR) is a root registry which has a root key generator of uddi:keygenerator so to avoid collisions with keys of entities published in the UBR, it is recommended that private Root registries do not use this as their root key generator.

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:aPublisherSubdivisonIdentifier: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:aPublisherSubdivisonIdentifier: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:aPublisherSubdivisonIdentifier: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:aPublisherSubdivisonIdentifier:a:keygenerator
```

valid keys are:

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisonIdentifier:a:1

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisonIdentifier:a:2

Use of digital signatures with the UDDI Registry

In UDDI v3, Publishers can digitally sign UDDI elements while they are publishing. The UDDI v3 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.

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 Certification 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 require 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 v3 Registry provides helper classes to assist you when you genearte applications to create, sign and use the UDDI v3 publish UDDI elements. These classes also help generating client applications that are designed to use the UDDI v3 Inquire service to retrieve and validate UDDI v3 elements.

The IBM 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 Javadoc welcome 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 Central called *UDDIv3ClientSignedBusinessSample.java*.

See "Securing Web services using XML digital signature" in the information center for a full explanation about Digital Signatures.

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 IBM WebSphere 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 Registry API, visit http://uddi.org/pubs/uddi_v3.htm.

Changes from the UDDI Version 3 specification are documented within the IBM WebSphere UDDI Registry information.

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

Inquiry API for the UDDI V3 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
- Find_qualifiers for API functions in the UDDI Registry
- The invocation pattern
- Inquiry API functions

For more information refer to the UDDI Version 3 Specifications.

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 one to locate and obtain detail on 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. For more detail on the functions see Section 5.1.8 in the UDDI Version 3 Specification . The `V3Client` offers all functions/arguments described in the UDDI Version 3 specification. It is worth noting that the UDDI gui does not support all functions/arguments described in the UDDI Version 3 specification.

The queries available are:

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 run-time 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 on 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.

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 identifierBag 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 qualifer 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 V3 Registry:

The requests in this section represent commands that are used to publish, delete and update information contained in a UDDI registry. The messages defined in this section all behave synchronously.

The Publishing API calls defined that UDDI operators support are:

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 V3 Registry:

The UDDI Custody and Ownership API is to transfer custody or ownership of one or more entities contained in a UDDI registry.

The UDDI registry supports only intra-node ownership transfer and **not** inter-node custody transfer.

The UDDI 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 API specification at http://www.uddi.org/pubs/uddi_v3.htm.

Security API for the UDDI V3 Registry:

The security API is a part of the Version 2 Publishing API, but in Version 3 it has its own API set. The Version 3 security API includes the following 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 API specification at http://www.uddi.org/pubs/uddi_v3.htm.

IBM 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. The API documentation be found at the Javadoc welcome page.

This client also contains a XML Digital Signature utility class called SignatureUtilities provided to construct and validate Digital Signatures on UDDI elements. See the XML Digital Signature documentation 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 WAS_HOME/UDDIReg/client

IBM UDDI V3 Client samples

Samples using uddiv3client are available through the Web Services UDDI Samples link in the Samples Central page of the IBM Developer Domain Web Site. There are 5 samples.

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.

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" in the information center.

UDDI Registry SOAP Service End Points

UDDI V3 supports multiple versions and, depending on WAS security settings and UDDI SOAP service user-data constraint transport-guarantee settings, supports different end-points for different services.

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://hostname:9080/uddisoap/inquiryapi

Publish service

Default (soap.war) context-root='/uddisoap' and url-pattern = 'publishAPI' or 'publishapi'.

Default URL: https://hostname:9443/uddisoap/publishapi or
http://hostname:9080/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://hostname:9080/uddiv3soap/services/UDDI_Inquiry_Port

Publish service

Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Publish_Port'

Default URL: https://hostname:9443/uddiv3soap/services/UDDI_Publish_Port or
http://hostname:9080/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://hostname:9443/uddiv3soap/services/UDDI_Security_Port or

http://hostname:9080/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 WebSphere Application Server with security enabled and the UDDI SOAP service user data constraint transport-guarantee default settings are unchanged, then services with default end-point URLs with HTTPS and port 9443 require their data to be transported confidentially. Requests that are not using HTTPS will be rejected.

If you configure the WebSphere Application Server with security disabled or with security enabled and user data constraint transport-guarantee set to NONE, the services with default end-point URLs with HTTPS and port 9443 you can also use the HTTP and port 9080.

Programming the 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. Although the samples are written in Java, 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 standard documentation:

http://www.uddi.org/pubs/uddi_v3.htm

For more information on using the SOAP API, refer to "The UDDI Registry application programming interface".

UDDI4J programming interface (Deprecated)

The following considerations are specific to the support for UDDI4J specification provided by WebSphere Application Server:

- **UDDI4J class libraries provided.**

WebSphere Application Server provides two UDDI4J class libraries:

uddi4jv2.jar

This class library contains classes which support Version 2 of the UDDI specification.

uddi4j.jar

This class library is provided for compatibility with WebSphere Application Server and supports Version 1 of the UDDI specification. The classes in this library are deprecated.

Note that the uddi4jv2 APIs are deprecated in IBM WebSphere UDDI Registry Version 3. The IBM UDDI Version 3 Client for Java is the preferred API for accessing UDDI using Java.

UDDI EJB Interface (Deprecated)

The UDDI EJB interface is deprecated in WebSphere Application Server version 6 or later and supports UDDI version 2 API requests only.

This section describes how to use the EJB application programming interface (API) of the IBM WebSphere UDDI Registry component to publish, find and delete UDDI entries.

The client classes that are required for the EJB interface are contained in \$WAS_HOME/UDDIReg/ejb/ejbclient.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 Inquiry API functions, and all the public methods on the PublishBean correspond to UDDI Publish API functions. Not all UDDI API functions are implemented, for example get_authToken, discard_authToken, get_businessDetailExt.

The two EJBs use container-managed transactions. The transaction attribute for the methods of the InquiryBean is NotSupported, and for the methods of the PublishBean it is Required. You must not change the transaction attributes as this could result in undesirable behavior.

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-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

Using the EJB Client

In this section it is assumed that you have installed both WebSphere Application Server version 6.0 and the UDDI Registry (and is running). You cannot use the EJB Client from a machine that does not have WebSphere installed.

1. Set up your environment for communicating with WebSphere:
 - UNIX: . \$WAS_HOME/bin/setupCmdLine.sh (note that there is a space between the '.' and \$WAS_HOME...)
 - Windows: %WAS_HOME%/bin/setupCmdLine.bat
2. Ensure that your CLASSPATH includes the uddiejbclient.jar (from \$WAS_HOME/UDDIReg/ejb) and the code for your client.
3. Compile your EJB client programs:
 - UNIX: \$JAVA_HOME/bin/javac -extdirs \$WAS_EXT_DIRS:\$JAVA_HOME/jre/lib -classpath \$WAS_CLASSPATH:\$CLASSPATH yourcode.java
 - Windows: %JAVA_HOME%\bin\javac -extdirs %WAS_EXT_DIRS%;%JAVA_HOME%\jre\lib -classpath %WAS_CLASSPATH%;%CLASSPATH% yourcode.java
4. Execute the compiled programs:
 - UNIX: \$JAVA_HOME/bin/java -Djava.ext.dirs=\$WAS_EXT_DIRS:\$JAVA_HOME/jre/lib -Dwas.install.root=\$WAS_HOME -Dserver.root=\$WAS_HOME \$CLIENTSAS \$CLIENTSOAP -cp \$WAS_CLASSPATH:\$WAS_HOME/UDDIReg/ejb/uddiejbclient.jar:\$CLASSPATH <class name> <args>
 - Windows: %JAVA_HOME%\bin\java -Djava.ext.dirs=%WAS_EXT_DIRS%;%JAVA_HOME%\jre\lib -Dwas.install.root=%WAS_HOME% -Dserver.root=%WAS_HOME% %CLIENTSAS% %CLIENTSOAP% -cp %WAS_CLASSPATH%;%WAS_HOME%\UDDIReg\ejb\uddiejbclient.jar;%CLASSPATH% <class name> <args>

Ensure that your PATH statement starts with WAS_HOME\java\bin

Datatypes package in the UDDI Registry:

The following table lists the classes in the com.ibm.uddi.datatypes package, the elements in the UDDI v3.0 XML schema, and the correspondence between the two.

com.ibm.uddi.datatypes Class	Corresponding UDDIv2.0 XML Schema Element	Notes on DatatypeClass
AccessPoint	accessPoint	
Address	address	
	String addressLine	
AdressLineList		Encapsulates a vector of addressLine Strings
AddressList		Encapsulates a vector of Address objects
AssertionStatusItem	assertionStatusItem	
AssertionStatusItemList		Encapsulates a vector of AssertionStatusItem objects
AssertionStatusReport	assertionStatusReport (response message)	
	String authInfo	
AuthToken		Object containing authInfo String and operator name
	String bindingKey	
BindingDetail	bindingDetail (response message)	
BindingTemplate	bindingTemplate	
BindingTemplateList	bindingTemplates	Encapsulates a vector of Bindingtemplate objects
BusinessDetail	businessDetail (response message)	
BusinessDetailExt	businessDetailExt (Response message)	**
BusinessEntity	businessEntity	
BusinessEntityExt	businessEntityExt	**
BusinessEntityExtList		Encapsulates a vector of BusinessEntityExt objects **
BusinessEntityList		Encapsulates a vector of BusinessEntity objects
BusinessInfo	businessInfo	
BusinessInfoList	businessInfo	Encapsulates a vector of businessInfo objects
	String businessKey	
BusinessList	businessList (response message)	
BusinessService	businessService	
BusinessServiceList	businessServices	Encapsulates a Vector of BusinessService objects
CategoryBag	categoryBag	
	String completionStatus	
Contact	contact	
ContactList	contacts	Encapsulates a vector of Contact objects
Description	description	
DescriptionList		Encapsulates a vector of Description objects
DiscoveryUrl	discoveryURL	
DiscoveryUrlList	discoveryURLs	Encapsulates a vector of DiscoveryURL objects
DispositionReport	dispositionReport	

DispositionreportException		Exception thrown by EJB interface functions when an error occurs
Email	email	
EmailList		Encapsulates a vector of Email objects
EndPoint		Used as baseclass for AccessPoint and HostingRedirector providing mutual exclusivity
ErrInfo	errInfo	
	findQualifier	
FindQualifier	findQualifiers	
	String fromKey	
HostingRedirector	hostingRedirector	
IdentifierBag	identifierbag	
InquiryOptions		Encapsulates a FindQualifiers object and a maxrows field. Used in find_* API calls to specify search options
InstanceDetails	instanceDetails	
	String instanceParms	
	String keyValue	
KeyedReference	keyedReference	
keysOwned	keysOwned	
LanguageString		Abstract class, extended by some of the datatypes, which represents a string that can optionally be tagged with xml:lang.
Name	name	
NameList		Encapsulates a vector of Name objects
OverviewDoc	overviewDoc	
	String overviewURL	
	String personName	
Phone	phone	
PhoneList		Encapsulates a vector of Phone objects
PublisherAssertion	publisherAssertion	
PublisherAssertionList		Encapsulates a vector of Publisher Assertion objects
PublisherAssertions	publisherAssertions (response message)	
RegisteredInfo	registeredInfo (response message)	
	relatedBusinessInfo	Not used
	relatedBusinessInfos	Not used
RelatesBusinessesList	relatedBusinessesList	
RelatedBusinessInfo	relatedBusinessInfo	
RelatedBusinessInfos	relatedBusinessInfos	
Result	result	
ResultList		Encapsulates a Vector of Result objects

ServiceDetail	serviceDetail (response message)	
ServiceInfo	serviceInfo	
ServiceInfoList	serviceInfos	Encapsulates a vector of serviceInfo objects
	String serviceKey	
ServiceList	serviceList (response message)	
	sharedRelationships	Not used
SharedRelationships	sharedRelationships	
Tmodel	tModel	
TModelBag	tModelBag	
TModelDetail	tModelDetail (response message)	
TModelInfo	tModelInfo	
TModelInfoList	tModelInfos	Encapsulates a vector of TModelInfo objects
TModelInstanceInfo	tModelInstanceInfo	
TModelInstanceInfoList	tModelInstanceDetails	Encapsulates a vector of TModelInstanceInfo objects
	String tModelKey	
TModelList	tModelList (response message)	
TModels		Encapsulates a vector of TModel objects
	String toKey	
	String uploadRegister	
UploadRegisterList		Encapsulates a vector of uploadRegister strings

** Used in UDDI API functions relating to BusinessDetailExtension. These UDDI API functions are not implemented in Version 1 of the IBM WebSphere UDDI Registry.

In general, a datatype called *DatatypeList* contains a vector of *Datatype* objects. Often these correspond to XML schema elements with plural names. (For example the datatype *Contact* corresponds to XML element *contact*, and *ContactList* corresponds to *contacts*.) Where there is no "plural" XML schema element for a particular *Datatype*, often there is still a *DatatypeList* where it is useful to have one, for example *AddressList*.

The exceptions to this naming convention occur when there is an existing XML schema element ending in "List". The exceptions are: *TModelList*, *ServiceList*, *BusinessList*. In these cases, the corresponding datatypes are given the same names as the XML schema elements, and the datatypes that would have had these names are called: *TModels*, *BusinessServiceList*, *BusinessEntityList*.

EJB interface methods in the UDDI Registry:

Inquiry

```
findBinding
findBusiness
findRelatedBusinesses
findService
findTModel
```

getBindingDetail
getBusinessDetail
getServiceDetail
getModelDetail

Publish

addPublisherAssertions
deleteBinding
deleteBusiness
deletePublisherAssertions
deleteService
deleteModel
getAssertionStatusReport
getRegisteredInfo
getPublisherAssertions
saveBinding
saveBusiness
saveService
saveModel
setPublisherAssertions

Each method is overloaded and can take various combinations of arguments. The Javadoc information contains detailed information about each method.

Note that the `get_authToken` and `discard_authToken` methods are not implemented, as WebSphere security is used instead.

Assembling Web services applications

This topic explains how to assemble a Web services application that is based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

This task provides information about what you need to assemble a Web service and in what order you should assemble the parts, for example an enterprise archive (EAR) file. Assembling a Web service is done after you develop the application and configure the deployment descriptors.

Assemble Web services applications by following the actions in the steps for this task section.

1. Start an assembly tool. 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. Assemble a Web services-enabled enterprise JAR file into an EAR file.
3. (Optional) Enable the EAR file. When the EAR file contains enterprise JavaBeans (EJB) modules, it must have the Web services endpoint Web archive (WAR) file added with the **endptEnabler** command-line tool or an assembly tool before deployment.
4. Assemble a Web services-enabled WAR file into an EAR file.

You have a Web services-enabled EAR file that you can deploy into WebSphere Application Server.

Now you need to deploy the Web services-enabled EAR file into WebSphere Application Server.

Assembling a JAR file that is enabled for Web services from an enterprise bean

This topic explains how to assemble a Web service-enabled enterprise bean Java archive (JAR) file with an assembly tool.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

You need the following artifacts that are generated from the **WSDL2Java** command to complete this task:

- An assembled enterprise bean JAR file that is not enabled for Web services
- A compiled Java class for the service endpoint interface
- A Web Services Description Language (WSDL) file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptor, and Java API for XML-based remote procedure call (JAX-RPC) mapping file.

Assemble a Web services-enabled enterprise bean JAR file from Java code by following the actions in the steps for this task section.

1. Start the assembly tool. 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. Click **File > Import** to import the enterprise bean JAR file into the assembly tool.
3. Open the J2EE perspective by clicking **Windows > Open Perspective > Other > J2EE**.
4. Switch to the Navigator pane by clicking the **Navigator** tab.
5. Locate the project containing the JAR file that you just imported in the **Navigator** pane.
6. Expand the `ejbModule` entry until the `META-INF` directory display is displayed. Expand the `META-INF` directory.
7. Right-click the `META-INF` directory and click **New > Folder**. Create a subfolder named `wsdl` in the `META-INF` directory.
8. Copy the WSDL file to the `META-INF\wsdl` directory by right-clicking the `wsdl` directory and click **Import > File system**. Browse the WSDL file for this Web service and click **Finish**.
9. Copy the JAX-RPC mapping file, `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` files into the `META-INF` directory.
10. Import the service endpoint interface class so that the service endpoint interface package begins in the `ejbModule` directory. You can import either the source file or a compiled class file. If you import the source file it automatically compiles.

You have the artifacts required to Web service-enable an Enterprise JavaBeans (EJB) module for Web services. The artifacts are added to the JAR file. Now you need to configure the deployment descriptors so that you can deploy the Web service into the WebSphere Application Server run time environment.

The `AddressBook.jar` JAR file contains the following files after assembly. The files added in this task are in bold. These files include the WSDL file, the deployment descriptors, and the JAX-RPC mapping file.

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookBean.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
```

META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml

Assemble the EAR file so that you can deploy the EAR file into WebSphere Application Server.

Assembling a Web services-enabled enterprise bean JAR file from a WSDL file

This task explains how to assemble a Web services-enabled enterprise bean Java archive (JAR) file from a Web Services Description Language (WSDL) file with an assembly tool.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

You need the following artifacts to complete this task:

- An assembled enterprise bean JAR file that contains the Enterprise JavaBeans (EJB) implementation and all classes that generate from the **WSDL2Java** command tool when the role argument is `develop-server` and the container argument is `EJB`.
- A WSDL file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file.

Assemble a Web services-enabled enterprise bean JAR file from a WSDL file by following the actions in the steps for this task section.

1. Start the assembly tool. 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. Click **File > Import** to import the enterprise bean JAR file into the assembly tool.
3. Open the J2EE perspective by clicking **Windows > Open Perspective > Other > J2EE**.
4. Switch to the Project Navigator pane by clicking the **Project Navigator** tab.
5. Locate the project for the JAR file you just imported in the **Project Navigator** pane.
6. Expand the `ejbModule` entry so that the `META-INF` directory is displayed. Expand the `META-INF` directory.
7. Right-click the `META-INF` directory and select **New > Folder**. Create a subfolder named `wsdl` in the `META-INF` directory.
8. Copy the WSDL file to the `META-INF\wsdl` directory by right-clicking the `wsdl` directory and click **File > Import > File system**. Browse the WSDL file for this Web service and click **Finish**.
9. Copy the JAX-RPC mapping file as specified by the deployment descriptor `<jaxrpc-mapping-file>` element of the `webservices.xml` file.
10. Copy the `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors into the `META-INF` subdirectory in the same manner.

You have the artifacts required to Web service-enable an Enterprise JavaBeans (EJB) module for Web services. The artifacts are added to the JAR file. Now you need to configure the deployment descriptors so that you can deploy the Web service into the WebSphere Application Server run time environment.

After assembling the `AddressBook.jar` JAR file contains the following files after assembly. The files added in this task are in bold. These files include the WSDL file, the deployment descriptors, and the JAX-RPC mapping file.

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookSoapBindingImpl.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsd1/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

Configure the `webservices.xml` deployment descriptor. You need to configure the deployment descriptors for the Web service so that WebSphere Application Server can process the incoming Web services requests.

Assembling a WAR file that is enabled for Web services from Java code

This topic explains how to assemble a WAR file that is enabled for Web services from Java code with an assembly tool.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

You need the following artifacts that are generated by the **WSDL2Java** command-line tool to complete this task:

- An assembled WAR file that contains the `web.xml` file, but is not enabled for Web services.
- The Java class for the service endpoint interface
- A Web Services Description Language (WSDL) file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file classes that are generated by the **WSDL2Java** command.

Assemble a Web services-enabled WAR file from Java code by following the actions in the steps for this task section.

1. Start an assembly tool. 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. Click **File > Import** to import the WAR file into the assembly tool.
3. Open the J2EE perspective by clicking **Windows > Open Perspective > Other > J2EE**.
4. Switch to the Navigator pane by clicking the **Navigator** tab.
5. Locate the project for the WAR file you just imported in the **Navigator** pane.
6. Expand the WebContent directory so that the WEB-INF directory is displayed. Expand the WEB-INF directory.
7. Confirm that the WEB-INF/web.xml deployment descriptor for the Web module contains a `<servlet-class>` element that indicates the Java bean class that is implementing the service:
 - a. Double-click **Web Deployment Descriptor**.
 - b. In the Web Deployment Descriptor editor, click the **Servlets** tab at the bottom of the editor window.
 - c. Enter the full path name of the Java bean class that implements the Web service in the **Servlet class** field.

- d. Close the editor window to save your changes.
8. Right-click the WEB-INF directory and click **New > Folder**. Create a subfolder named wsdl in the WEB-INF directory.
9. Copy the WSDL file to the WEB-INF\wsdl directory by right-clicking the wsdl directory and click **Import > File system**. Browse the WSDL file for this Web service and click **Finish**.
10. Copy the JAX-RPC mapping file, as specified by the <jaxrpc-mapping-file> element of the webservices.xml file.
11. Copy webservices.xml,ibm-webservices-bnd.xmi and ibm-webservices-ext.xmi files into the WEB-INF subdirectory in the same manner.
12. Import the service endpoint interface class so that the service endpoint interface package begins in the JavaSource directory. When you import the source file, it is automatically compiled.

The artifacts required to enable the Web module for Web services are added to the WAR file.

Now you can assemble the WAR file that is enabled for Web services into an EAR file.

Assembling a Web services-enabled WAR file from a WSDL file

This topic explains how to use to assemble a Web archive (WAR) file from a Web Services Description Language (WSDL) file that is enabled for Web services.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

You need the following artifacts to complete this task:

- An assembled WAR file that contains the Enterprise JavaBeans (EJB) implementation, all the classes that generate from the **WSDL2Java** command tool and the web.xml deployment descriptor file.
- A WSDL file
- The complete webservices.xml, ibm-webservices-bnd.xmi, and ibm-webservices-ext.xmi deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file.

Assemble a Web services-enabled WAR file from a WSDL file by following the actions in the steps for this task section.

1. Start an assembly tool. 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. Click **File > Import** to import the WAR file into the assembly tool.
3. Open the J2EE perspective by clicking **Windows >Open Perspective > Other > J2EE**.
4. Switch to the Navigator pane by clicking the **Navigator** tab.
5. Locate the project for the WAR file that you just imported in the **Navigator** pane.
6. Expand the WebContent directory so that the WEB-INF directory is displayed. Expand the WEB-INF directory.
7. Confirm that the WEB-INF/web.xml deployment descriptor for the Web module contains a <servlet> element including the <servlet-name> element:
 - a. Double-click **Web Deployment Descriptor**. If you are in the Navigator view, you need to double-click on the web.xml deployment descriptor. If you are in the Project Explorer view, you can double-click on Deployment Descriptor. Both of these ways open the Web Deployment Descriptor Editor.
 - b. In the Web Deployment Descriptor editor, click the **Servlets** tab.

- c. Enter the full path name of the Java bean class implementing Web services in the **Servlet class** field.
- d. Close the editor window to save your change.
8. Right-click the WEB-INF directory and select **New > Folder**. Create a subfolder named `wsdl` in the WEB-INF directory.
9. Copy the WSDL file to the WEB-INF\wsdl directory by right-clicking the wsdl directory and click **Import > File system**. Browse the WSDL file for this Web service and click **Finish**.
10. Copy the JAX-RPC mapping file as specified by the deployment descriptor `<jaxrpc-mapping-file>` element of the `webservices.xml` file.
11. Copy the `webservices.xml`, `ibm-webservices-ext.xmi`, and the `ibm-webservices-bnd.xmi` deployment descriptors in the WEB-INF subdirectory.

The artifacts required to enable the Web module for Web services is added to the WAR file.

Now you can assemble the WAR file that is enabled for Web services into an EAR file.

Assembling an enterprise bean JAR file into an EAR file

This task explains how to assemble the enterprise bean Java archive (JAR) file that you created in the previous task into an enterprise archive (EAR) file with an assembly tool. Assembling the JAR file, and now the EAR file, are required tasks to enable Java code for Web services.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

Before assembling a Web services-enabled EAR file you must assemble an enterprise bean JAR file that you want to enable for Web services. To learn more about the artifacts that are needed for the assembly of the enterprise bean JAR file see [Assemble an enterprise bean JAR file from Java code that is enabled for Web services](#).

To assemble a Web services-enabled EAR file:

1. Start the assembly tool. 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. Assemble the Web services-enabled JAR file into an EAR file. The EAR file can contain an enterprise bean or application client JAR files, WAR files, Web applications, and metadata describing the applications or `application.xml` files.

A Web services-enabled EAR file.

In the following example, there is an `application.xml` deployment descriptor packaged with a Web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.jar
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
<application id="Application_ID">
  <display-name>AddressBookJ2WEE</display-name>
```

```

<description>AddressBook EJB Example from Java</description>
<module id="EjbModule_1">
  <ejb>AddressBook.jar</ejb>
</module>
</application>

```

Enable the EAR file. Then, deploy the EAR file into WebSphere Application Server.

Assembling a Web services-enabled WAR into an EAR file

This task explains how to assemble a Web services-enabled Web archive (WAR) file into an enterprise archive (EAR) file with an assembly tool.

You can assemble Web Services for J2EE modules with assembly tools provided with WebSphere Application Server.

You must configure the assembly tool before you can use it.

Assemble a Web services-enabled WAR file into an EAR file using the steps provided in this task section.

1. Start the assembly tool. 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. Assemble the Web services-enabled WAR file into an EAR file. Now assemble the EAR file that contains the JAR or WAR files. The EAR file can contain an enterprise bean or application client JAR files; Web applications or WAR files; and metadata describing the applications or application.xml files.

A Web services-enabled EAR file.

In the following example, there is an application.xml deployment descriptor packaged with a Web services-enabled JAR file called AddressBook.jar that is packaged into an EAR file called AddressBook.ear. The EAR file contains:

```

META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.war

```

An example of the application.xml deployment descriptor is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
<application id="Application_ID">
  <display-name>AddressBook</display-name>
  <description>AddressBook Example from Java bean</description>
  <module id="WebModule_1">
    <web>
      <web-uri>AddressBook.war</web-uri>
      <context-root>/AddressBook</context-root>
    </web>
  </module>
</application>

```

Deploy Web services.

Enabling an EAR file for Web services

This task explains how to enable the enterprise archive (EAR) file that you have assembled with artifacts that are necessary to build a Web service. The EAR file is an application that is enabled for Web services.

You need to assemble the Java archive (JAR) file before you can assemble these files into the EAR file, because the enterprise JAR file is an artifact that is required to build the EAR file.

You can add router modules to your application that is enabled for Web services with either the **endptEnabler** command-line tool with assembly tools provided with WebSphere Application Server. The tool that you choose to use for this task depends on your preference to work with a command-line tool or a graphical user interface.

These tools add one or more router modules to the EAR file for each Web service-enabled enterprise bean JAR module contained in the EAR file. A router module provides an endpoint for the Web service in a particular enterprise bean JAR module.

Each router module supports a specific transport such as HTTP or Java Message Service (JMS). If no enterprise bean JAR modules are located in the EAR file, it is not necessary to use these tools.

- Enable an EAR file that is enabled for Web services with an assembly tool
- Enable an EAR file with the **endptEnabler** command-line tool. In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application.

.Deploy the EAR file into WebSphere Application Server. An assembled EAR file that is enabled for Web services is required for deployment.

Enabling an EAR file for Web services with the endptEnabler command:

This task explains how to use the **endptEnabler** command-line tool to enable an enterprise archive (EAR) file for Web services.

Before doing this task, you need to assemble a Web services-enabled enterprise JAR into an EAR file.

The **endptEnabler** command-line tool adds one or more router modules to the EAR file for each Web service-enabled enterprise bean JAR module within the EAR file. A router module provides an endpoint for the Web services in a particular enterprise bean JAR module.

Each router module supports a specific transport such as HTTP or Java Message Service (JMS). If no enterprise bean JAR modules exist in the EAR file, it is not necessary to use these tools.

Enable an EAR file with the **endptEnabler** command by following the actions listed in the steps for this task section.

1. Invoke the **endptEnabler** command from the *install_root\bin* directory. If you are using Linux or Unix platforms, invoke the command from the *install_root/bin* directory.
2. Enter the name of the EAR file, when prompted.
3. Enter various input values as requested by the **endptEnabler** command. You are prompted for various input values for each enterprise bean JAR module that is enabled for Web services in the EAR file. Typically, you accept the defaults for each prompt. See **endptEnabler** prompts and commands for more information about **endptEnabler** command prompts.

An HTTP or JMS router module is added to the EAR file for each enterprise bean JAR module that is enabled for Web services and contained in the EAR file. For HTTP, a context-root is configured for the application so that the Web service can be invoked through a Web address. The Web address used to invoke the Web service is:

```
http://host[:port]/context-root/services/port-component-name
```

Deploy the EAR file into WebSphere Application Server. An assembled EAR file that is enabled for Web services is required for deployment.

endptEnabler command: The **endptEnabler** command enables a set of Web services within an enterprise archive (EAR) file. The **endptEnabler** command must run on EAR files containing Enterprise JavaBeans (EJB) modules that are enabled for Web services.

Each router module provides a Web service endpoint for a particular transport. For example, you can add a HTTP router module can be added so that the Web service can receive requests over the HTTP transport. You can add a Java Message Service (JMS) router module so that the Web service can receive requests from a JMS queue or topic.

In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application. The **endptEnabler** command makes a backup copy of your original EAR file in the event that you need to remove or add services at a later time. If your EAR file contains an enterprise bean Java archive (JAR) file that is enabled for Web services, you must run the **endptEnabler** command before the EAR file is deployed. Otherwise, you do not need to run the command.

endptEnabler usage syntax

Invoke the **endptEnabler** command from the WebSphere Application Server bin directory. The command syntax is presented in the following example:

```
endptEnabler
  [-verbose|-v]
  [-quiet|-q]
  [-help|-h|-?]
  [-properties|-p properties-filename]
  [-transport|-t default-transport]
  [-enableHttpRouterSecurity]
  [ear-filename]
```

All parameters are optional and described in the following list:

- **-verbose, -v**
Details and displays progress messages as the tool processes the EAR file. This command-line option is mapped to the verbose global property.
- **-quiet, -q**
No displays of per-module progress messages as the tool processes the EAR file. This command-line option is mapped to the quiet global property.
- **-help, -h, -?**
Displays a brief help message, explaining the various options.
- **-properties, -p <properties-filename>**
Reads properties from the *properties-filename* properties and controls the behavior of the tool.
- **-transport, -t <default-transport>**
Specifies the default list of transports for which router modules are created for each enterprise bean JAR file contained in the EAR file. This command-line option is mapped to the defaultTransports global property. Examples include:
-transport http (the default)
-transport jms
-t http,jms
- **-enableHttpRouterSecurity**
Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the Enterprise JavaBeans (EJB) are secured in the enterprise bean JAR file. This command-line option is mapped to the http.enableRouterSecurity global property.
- **<ear-filename>**
Specifies the name of the EAR file to be processed.

If the *ear-filename* parameter is not entered on the command line, the interactive mode is used. In the interactive mode, you are prompted for the EAR file name, the router module names and other important values as the processing occurs. The following dialog is an example of the `endptEnabler` interactive mode.

In this dialog, the user input is in fixed width font, and the `endptEnabler` output is in bold.

```
endptEnabler<enter>
WSWS2004I: IBM WebSphere Application Server Release 5
WSWS2005I: Web Services Enterprise Archive Endpoint Enabler Tool.
WSWS2007I: (C) COPYRIGHT International Business Machines Corp. 1997, 2003
WSWS2006I: Please enter the name of your EAR file: AddressBook.ear<enter>

WSWS2003I: Backing up EAR file to: AddressBook.ear~

WSWS2016I: Loading EAR file: AddressBook.ear
WSWS2017I: Found EJB Module: AddressBookEJB.jar

WSWS2029I: Enter http router name for EJB Module AddressBookEJB
[AddressBookEJB_HTTPRouter.war]:<enter>
WSWS2030I: Enter http context root for EJB Module AddressBookEJB
[/AddressBookEJB]:<enter>
WSWS2024I: Adding http router for EJB Module AddressBookEJB.jar.
WSWS2036I: Saving EAR file AddressBook.ear...
WSWS2037I: Finished saving the EAR file.
WSWS2018I: Finished processing EAR file AddressBook.ear.
```

If the *ear-filename* parameter is entered on the command-line, the non-interactive mode is used. In the non-interactive mode, the router module names and other important values are determined from the user-specified properties or default values.

endptEnabler properties

With the `endptEnabler` command you can control its run-time behavior by specifying a set of properties with the `-properties` command-line option. These properties are organized in one of two ways: global and per-module. Global properties affect the overall behavior of the tool as it processes multiple enterprise bean JAR modules within the EAR file. Per-module properties affect the processing of a particular enterprise bean JAR module.

Global properties

The following table describes the global properties supported by the `endptEnabler` command:

Property name	Description	Default value
verbose	Displays detailed progress messages.	False
quiet	Displays only brief progress messages.	False
http.enableRouter Security	Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJBs are secured in the enterprise bean JAR file.	False
http.router ModuleNameSuffix	Specifies the suffix used to construct default HTTP router module names. The <code>.war</code> extension is added by the <code>endptEnabler</code> command.	<code>_HTTPRouter</code>
jms.routerModule NameSuffix	Specifies the suffix used to construct default JMS router module names. The <code>.jar</code> extension is added by the <code>endptEnabler</code> command.	<code>_JMSRouter</code>

jms.default DestinationType	Specifies the default destination type to use for all JMS router modules added to the EAR file. This type should be either queue or topic.	queue
defaultTransports	Specifies the default list of transports for which router modules are created. The list can contain the values http and jms. Multiple values are separated by a comma. Examples are: http, jms and http,jms.	http

Per-module properties

The following table describes the per-module properties supported by the **endptEnabler** command. The *ejbJarName* variable refers to the name of an enterprise bean JAR module within the EAR file, without the .jar extension.

Property name	Description	Default value
<ejbJarName> .transports	Lists the transports for which router modules are created for a particular enterprise bean JAR file. The list can contain the values http and jms. Multiple values are separated by a comma. Examples are: http, jms and http,jms.	http
<ejbJarName> .http.skip	Specifies the flag which bypasses the addition of an HTTP router module, even if it otherwise is added (based on other properties). Valid values are true and false.	false
<ejbJarName> .http.routerModuleName	Specifies the name of the HTTP router module for a particular enterprise bean JAR file.	<i>ejbJarName_HTTPRouter</i>
<ejbJarName> .http.contextRoot	Specifies the context root associated with the HTTP router module for a particular enterprise bean JAR file.	<i>/ejbJarName</i>
<ejbJarName> .jms.skip	Specifies the flag which bypasses the addition of an JMS router module even if it otherwise is added (based on other properties). Valid values are true and false.	false
<ejbJarName> .jms.routerModuleName	Specifies the name of the JMS router module for a particular enterprise bean JAR file.	<i>ejbJarName_JMSRouter</i>
<ejbJarName> .jms.activationSpecJndiName	Specifies the Java Naming and Directory Interface (JNDI) name of the activation specification that should be configured for the Message Driven Bean (MDB) within the JMS router module.	null
<ejbJarName> .jms.listenerInputPortName	Specifies the name of the listener port to configure for the MDB within the JMS router module. The listener port is configured only if an activationSpecJndiName property is not specified.	null

<ejbJarName>.ejb JarName>.jms. destinationType	Specifies the JMS destination type associated with the MDB within the JMS router. Valid values are queue and topic.	queue
---	---	-------

Properties example

Suppose an EAR file contains an enterprise bean JAR file named, StockQuoteEJB.jar that contains Web services. The following set of properties can be used to control the **endptEnabler** command run-time behavior as it processes the EAR file:

```
StockQuoteEJB.transports=http,jms
StockQuoteEJB.http.routerModuleName=StockQuoteEJB_HTTP
StockQuoteEJB.http.contextRoot=/StockQuote
StockQuoteEJB.jms.routerModuleName=StockQuoteEJB_JMS
StockQuoteEJB.jms.destinationType=queue
```

endptEnabler examples

The following commands are examples of how the **endptEnabler** command can be used:

```
endptEnabler MyApp.ear
endptEnabler -t jms,http MyApp.ear
endptEnabler -v -properties MyApp.props MyApp.ear
endptEnabler -q -t jms MyApp.ear
endptEnabler -v -t http,jms
```

Enabling endpoints for a Web service-enabled EAR file with an assembly tool:

This task explains how to enable an enterprise archive (EAR) file that has been assembled with all the required artifacts for Web services. The artifacts are assembled into a Java archive (JAR) file in the previous task.

Before doing this task, you need to Assemble an enterprise bean Java archive (JAR) file into an EAR file.

You can assemble the EAR file with assembly tools that are provided with WebSphere Application Server.

Before you can use the assembly tool, you must configure the assembly tool.

This task is one of the steps required to develop and implement a Web service.

You can add one or more router modules to your Web services-enabled application. The EAR file is the Web services-enabled application.

A *router module* provides an endpoint for the Web services in a particular enterprise bean JAR module.

Each router module supports a specific transport such as HTTP or Java Message Service (JMS). If no Web services-enabled enterprise bean JAR modules exist in the EAR file, it is not necessary to use these tools.

To enable end points for a Web service-enabled EAR file, follow the steps provided in this task section.

1. Start an assembly tool. 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. Right-click the EJB project to enable.
3. Click **Web Services > Endpoint Enabler**.
4. Specify the transport and router module names in the corresponding fields.
5. Click **OK**.

You have added an HTTP or JMS router module to the EAR file for each enterprise bean JAR module contained in the EAR file. For HTTP, a context-root is configured for the application so that the Web services can be invoked through a Web address. The Web address used to invoke the Web service is:

`http://host[:port]/context-root/services/port-component-name`

Now that the EAR file has been enabled, you can deploy the EAR file into WebSphere Application Server.

Data access resources

Learn about data access resources

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Establish data access using enterprise beans

- Develop enterprise beans
- Assemble beans into a data access application
- Map the beans to database tables
- Configure access to a relational database

Configure access to a non-relational database

- Create a resource adapter
- Create a connection factory

Establish data access without using enterprise beans

- Use optional IBM data access beans
- Verify your database vendor requirements (for relational databases)
- Configure access to a relational database
- Configure access to a non-relational database: See previous
- Set up JDBC - Cloudscape database
- Set up JDBC - DB2 database
- Set up JDBC - Oracle database
- Set up JDBC and J2C security in the administrative console

Establish data access using SDO

- The Service Data Objects framework

Logically name the new resource

- Establish resource references

Deploy your data access application

- Install application files (same as any application type)
- Deploy applications (Education on Demand)
- Administer applications (Education on Demand)

Test and monitor database connections

- Test connections
- Remedy stale connections
- Use the JDBC connection pool counter for relational databases
- Use the J2C connection pool counter for non-relational databases

Troubleshoot

- Resolve connectivity issues specific to database type

Conceptual overviews

Documentation

Refer to the article *Introduction: Data access resources* in the information center.

Presentations

Education on Demand offers:

- Adding security for JDBC and J2C resources
- Resource management
- JDBC
- JavaServer Faces Overview
- JavaServer Faces Architecture
- Service Data Objects Overview
- JDBC mediator
- EJB mediator

See also the IBM Redbook DB2 UDB V8 and WebSphere V5 Performance Tuning and Operations Guide

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

developerWorks offers:

- **Tutorial 5 - Java Connector Architecture (JCA 1.5)**
This tutorial illustrates the Java Connector Architecture (JCA), which defines the standards that allow J2EE application servers to interact with a back-end. This is called an Enterprise Information System (EIS). With JCA 1.5, it is possible for a back-end system to initiate communications towards the J2EE application server, and to propagate a transactional context too. The EIS will control the transaction boundaries in this scenario, the J2EE container acts as a transactional resource controlled by the EIS. The zip file comes with all sample code required to run this tutorial.

Samples

The Samples Gallery offers:

- **WebSphere Bank**
Using the WebSphere Bank online bank, customers can open accounts, get account balances, and transfer funds between accounts. The WebSphere Bank application uses Web services, Java Message Service (JMS) API, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, Message Driven Beans (MDB), JCA 1.5 API, JSP pages, and servlets.
- **Album catalog**
Album Catalog illustrates access to relational data in a web application via Service Data Objects (SDO) and JDBC Mediator. It illustrates the use of SDOs and JDBC Mediator to display and manipulate data from a relational database in a disconnected fashion.

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. See "Hardware and software requirements" in the information center for more information.

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 (RRA) 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 version 6.0 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 default WebSphere relational resource adapter settings.

This is the WebSphere-provided relational resource adapter for handling data access to any relational data base. This adapter is preinstalled by WebSphere Application Server. 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 > 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 392
- For information about resource adapters and data access, see “Data access portability features”
- For RRA settings, see “WebSphere relational resource adapter settings” on page 393
- For information about CMP connection factories, see “Connection factory” on page 398
- For information about enterprise beans, see “Introduction: EJB applications” in the information center

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 = 8788931090149908285L;

    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);

        // Note: 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. However, certain applications may need to differentiate
// between a custom helper and an existing helper of the same type,
// so WebSphere has provided the value 'DataStoreHelper.CUSTOM_HELPER'
// for this purpose. If this functionality is needed by your application
// insert the following line into your code:
// getMetaData().setHelperType(DataStoreHelper.CUSTOM_HELPER);

}

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 are the connection factories 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 DataSource 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 can not 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 DataSource for CMP* checkbox.

To view this administrative console page, click **Resources >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 > 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 access data from databases.

For applications that need access to relational databases, the JDBC provider and data source together are functionally equivalent to the J2EE Connector Architecture (JCA) connection factory. The WebSphere Application Server prerequisite Web site has a current list of supported providers. See "Hardware and software requirements" in the information center for more information.

Data sources

Installed applications uses a *data source* to access the data from the database.

A data source is associated with a JDBC provider that supplies the specific JDBC driver implementation class. The data source represents the J2EE Connector Architecture (JCA) connection factory for the relational resource adapter. Application components use the data source to access connection instances to a specific database; a connection pool is associated with each data source.

You can create multiple data sources with different settings, and associate them with the same JDBC provider. (One reason to do this is to provide access to different databases.) JDBC providers that are supported by WebSphere Application Server are required 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.)

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 (Version 4.0) - 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 applications and beyond. 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-datatypes 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 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, 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, using the Version 6.0 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:

When accessing any database, establishing connections is an expensive operation. *Connection pooling* enables administrators to establish a pool of database connections that applications can share on an application server. When connection pooling capabilities are used, performance improvements up to 20 times the normal results are realized.

Each time a resource attempts to access a backend store (such as a database), the resource must connect to that data store. A connection requires resources to create, maintain, and then release the connection when it is no longer required.

The total data store overhead for an application is particularly high for Web-based applications because Web users connect and disconnect more frequently. In addition, user interactions are typically shorter. Often, more effort is spent connecting and disconnecting than is spent during the interactions. Also, because Internet requests can arrive from virtually anywhere, you can find usage volumes large and difficult to predict.

To help lessen these overhead problems, the 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 resources for future requests.

WebSphere Application Server supports JDBC 3.0 APIs to provide support 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.

Each entity bean transaction requires an additional connection to the database specifically to handle the transaction. Take this into account when calculating the number of data source connections.

On UNIX platforms, a separate DB2 process is created for each connection and these processes quickly affect performance on systems with low memory and cause errors.

If clones are used, one data pool exists for each clone. This is important when configuring the database maximum connections.

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. 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 a *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.

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 non pooled 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)

Condition	Description
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).

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.

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.

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 `ConnectionFactory`. 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*, *PreparedStatement*s, and *ResultSet*s 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.)

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 than 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 unshareable 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. Furthermore, 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.

Connections and transactions:

All connection usage occurs within the scope of either a global transaction or a local transaction containment (LTC) boundary.

Connection behavior depends on your current operating scope. This article discusses some of the common characteristics you see when using connections in one of the transaction scopes.

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 409 topic for more details.

JDBC AutoCommit behavior

All JDBC connections, when first obtained through a getConnection() call, contain the setting AutoCommit = TRUE by default.

- 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 resources: 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. Furthermore, 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.

Passing client information to a database: Some databases enable you to set client information on the database connections using a backend-specific proprietary connection API. For some databases (such as DB2) you can also set the client information as a data source property. WebSphere Application Server before Version 6.0 only enables setting the client information as a data source property. This capability is somewhat limited, however, because the client information cannot be dynamically changed on the data source or the connections obtained from that data source. Also, setting the client information on the data source causes all connections created from that data source to have the same information. For example, if you set the `ApplicationName` as part of the data source *clientInformation*, all connections from that data source have the same application name. Because many different applications can access the same data source, this might not be desired.

With WebSphere Application Server Version 6.0, you can set the client information on some connections and not others, and you can set different client information on different database connections from the same data source. You can pass client information in one of two ways:

- Explicitly, by calling a proprietary API, `setClientInformation(Properties)` on the `com.ibm.websphere.rsadapter.WSCConnection`.
- Implicitly, using the `WAS.clientinfo` trace string. You can enable this dynamically from the administrative console just like a normal trace. For more information about passing client information implicitly, see “Setting client information implicitly” on page 417.

The API is defined on the `WSConnection` class which is part of the `com.ibm.websphere.rsadapter` package. You must cast your database connection in your applications to `com.ibm.websphere.rsadapter.WSConnection` before calling the API, as this is a WebSphere Application Server proprietary API. The API takes a properties object as an input parameter that provides the flexibility of adding new client information if and when it is introduced by the backend database, without any changes to this API itself.

```
public void setClientInformation (Properties props)throws SQLException;
```

For an example of using this API, see “Example: setClientInformation(Properties) API.”

Example: setClientInformation(Properties) API:

Usage Scenario

This API enables you to set client information on the WebSphere Application Server connection. Some of the client information is passed on to the backend database if that database supports such functionality.

Example

```
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) {System.out.println("got an exception during lookup: " + e);}

WSConnection conn = (WSConnection) ds.getConnection();
Properties props = new properties();
props.setProperty(WSConnection.CLIENT_ID, "user123");
props.setProperty(WSConnection.CLIENT_LOCATION, "127.0.0.1");
props.setProperty(WSConnection.CLIENT_ACCOUNTING_INFO, "accounting");
props.setProperty(WSConnection.CLIENT_APPLICATION_NAME, "appname");
props.setProperty(WSConnection.CLIENT_OTHER_INFO, "cool stuff");
conn.setClientInformation(props);
conn.close()
```

Parameters

props contains the client information to be passed. Possible values are:

- `WSConnection.CLIENT_ACCOUNTING_INFO`
- `WSConnection.CLIENT_LOCATION`
- `WSConnection.CLIENT_ID`
- `WSConnection.CLIENT_APPLICATION_NAME`
- `WSConnection.CLIENT_OTHER_INFO`
- `WSConnection.OTHER_CLIENT_TYPE`

Refer to the `WSConnection` documentation for more details on which client information is passed to the backend database. To reset the client information, call the method with a null parameter.

Exceptions

This API creates an SQL exception if the database issues an exception when setting the data.

Setting client information implicitly: You can choose to *explicitly* pass client information of application requests to database connections by calling an IBM proprietary API, `setClientInformation(Properties)`, on the `com.ibm.websphere.rsadapter.WSConnection` object within your application code. In some cases,

however, you might want WebSphere Application Server to handle the passing of client information to database connections. This method of setting the client information is referred to as *implicit*. You might choose the implicit method because:

- You want to keep your application free of proprietary APIs, *or*
- Your application uses container-managed persistence (CMP), in which case you cannot use the proprietary API to set client information on database connections.

The WebSphere Application Server trace facility provides the capability for setting client information implicitly. You can designate one of two special trace groups to enable or disable client information passing: `WAS.clientinfo` trace or `WAS.clientinfopluslogging` trace .

Note:

Connection sharing:

In the case of connection sharing, setting the client information implicitly is done only on the first acquired connection handle. If connection sharing is enabled and two or more `getConnection()` methods are called (resulting in two handles on the same connection), then only the first `getConnection()` call causes the client information to be implicitly passed to the back-end database. This is not true for the explicit case; every `setClientInformation()` method is driven down to the database regardless of connection sharing.

Implicit/explicit co-existence:

When both explicit and implicit are used, some combination of the explicitly set data and implicitly set data is combined, but for the most part the explicit takes precedence. For example, if the application sets the client accounting information to "myAccountingInfo", the final accountingInfo string that is passed to the backend database looks something like:

```
000325_WSRdbManagedConnectionImpl@1234_myAccountingInfo: where 000325 is the thread id, WSRdbManagedConnectionImpl@1234 is the websphere connection instance.
```

Client information reset:

In the implicit case, client information is reset when the connection is put back in the pool and only if the `WAS.clientinfo` and `WAS.clientinfopluslogging` are disabled (that is, `WAS.clientinfo=all=disabled:WAS.clientinfopluslogging=all=disabled`).

In the explicit case however, the reset is done only when the application issues `setClientInformation(null)` on the `WSConnection`.

WAS.clientinfo trace

By default, the implicit mechanism is disabled. You can turn on this mechanism dynamically (without stopping and starting your application server) or statically by setting the WebSphere Application Server trace group `WAS.clientinfo=all=enabled`.

The information implicitly collected and set on the database connection consists of the *user name*, *user location* and *application name*.

Note: User name and user location can only be implicitly collected and set on the database connection if you enable WebSphere global security.

username

Name of the user that initiated the application request. This option is collected and passed to the backend database (when supported) only if WebSphere global security is enabled. Information here is collected by calling `WSSecurityHelper.getFirstCaller()`.

user location

In the form of `cell:node:server`. This option is collected and passed to the backend database (when appropriate) only when WebSphere global security is enabled. Information here is collected by calling `WSSecurityHelper.getFirstServer()`.

application name

Name of the application running. This value is the output of *getApplication()* from the *J2EEName* object. This value is collected regardless of the Global Security setting.

WAS.clientinfopluslogging trace

When debugging database problems, such as deadlocks, there is a set of information that is needed to help with the debugging effort. This information is typically obtained by enabling RRA trace, and EJB container trace. However, there are some cases where timing is an issue when reproducing a given problem. Having too much tracing information can alter the behavior of the application (such as change the timing), and the problem might no longer occur.

Because of this, a new trace group is provided where only a minimum set of information is collected. This trace group is *WAS.clientinfopluslogging*. This function sets the client information implicitly on the connection (just like the *WAS.clientinfo* trace), as well as logs and traces important application activities. Those activities are:

- SQL Strings that were run (such as, select userId from tabl1 where id=? for update).
- Start, commit, and rollback of transactions.
- EJB calls (such as, Create, Remove, findByPrimaryKey, and so on).

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 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" in the information center for more information.

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*.

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
- CMP persistence functions
- Container-managed relationships
- Resource references
- 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
- Oracle
- DB2 database software
- Supported hardware, software, and APIs

Programming Specifications

- Enterprise JavaBeans Technology (Source for download of the Enterprise Javabeans 2.1 specification)
- Java™ 2 Platform, Enterprise Edition (J2EE™)
- Java™ Management Extensions (JMX)
- JDBC™ 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)

CMP persistence functions

Though this article addresses the EJB 2.0 specification, you still might find parts of it pertinent to your environment.

- Enterprise JavaBeans™ 2.0 Container-Managed Persistence Example

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™ 2.0 Container-Managed Persistence Example

Resource references

Though this article addresses the EJB 2.0 specification, you still might find parts of it pertinent to your environment.

- Accessing Databases from Web Applications

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 (Still pertinent to WebSphere Application Server Version 6.0)
- Database authentication in WebSphere Application Server V5 (Still pertinent to WebSphere Application Server Version 6.0)
- Understanding WebSphere Application Server EJB access intents

Rational Application Developer

- Rational Application Developer for WebSphere Software

WebSphere Version 5.x Information Center

- IBM WebSphere™ Version 5.x Information Center

IBM Cloudscape

- IBM Cloudscape
- developerWorks article: Cloudscape Network Server with WebSphere Application Server

Oracle

- Oracle

DB2 database software

- DB2

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.

You can use Structured Query Language in Java (SQLJ) support with both BMP and CMP beans when you are using the DB2 Universal JDBC driver provider with DB2 as your backend database. DB2 for z/OS users who wish to use SQLJ support with CMP beans must use the DB2 Universal JDBC driver provider.

The Service Data Objects (SDO) 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.

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.*

Using a resource reference to access your data source or connection factory is required when running in WebSphere Application Server. (The J2EE application *deployer* actually creates a JDBC provider and data source for relational database access, or creates a J2EE Connector Architecture (JCA) connection factory for non-relational database access. See “Deploying data access applications” on page 515 for more information.)

3. Get a connection to a data source (the “Getting connections” section of Connection life cycle). *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 DataSources defined by the JDBC 2.0 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:

- **WSDDataSource** 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 **WSDDataSource** 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 *WSRRRAFactory* class. For more information see the **JDBCConnection** API documentation topic (as listed in the API documentation index).
- **WSRRRAFactory** 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 **WSRRRAFactory** 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” on page 417 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;

    // Persistence table name

    private String tableName = "cmtest";

    // DataSource JNDI name

    private String dsName = "java:comp/env/jdbc/SimpleDS";

    // DataSource

    private DataSource ds = null;

    // bean instance variables.

    private int id;
    private String name;

    /**
     * In setEntityContext method, you need to get the AccessIntentService
     * object in order for the subsequent methods to get the AccessIntent
     * object.
     * Other ejb methods will call the private getConnection() to get the
     * connection which has all specific connection properties
     */

    public void setEntityContext(javax.ejb.EntityContext ctx) {
        myEntityCtx = ctx;

        try {
            aiService =
                (AccessIntentService) getInitialContext().lookup(
                    "java:comp/websphere/AppProfile/AccessIntentService");
            ds = (DataSource) getInitialContext().lookup(dsName);
        }
        catch (javax.naming.NamingException ne) {
            throw new javax.ejb.EJBException(
                "Naming exception: " + ne.getMessage());
        }
    }

    /**
     * ejbCreate
     */

    public fvt.example.SimpleKey ejbCreate(int newID)
        throws javax.ejb.CreateException, javax.ejb.EJBException {
        Connection conn = null;

```

```

PreparedStatement ps = null;

// Insert SQL String

String sql = "INSERT INTO " + tableName + " (id, name) VALUES (?, ?)";

id = newID;
name = "";

try {
    // call the common method to get the specific connection

    conn = getConnection();
}
catch (java.sql.SQLException sqle) {
    throw new EJBException("SQLException caught: " + sqle.getMessage());
}
catch (javax.resource.ResourceException re) {
    throw new EJBException(
        "ResourceException caught: " + re.getMessage());
}

try {
    ps = conn.prepareStatement(sql);
    ps.setInt(1, id);
    ps.setString(2, name);

    if (ps.executeUpdate() != 1) {
        throw new CreateException("Failed to add a row to the DB");
    }
}
catch (DuplicateKeyException dke) {
    throw new javax.ejb.DuplicateKeyException(
        id + "has already existed");
}
catch (SQLException sqle) {
    throw new javax.ejb.CreateException(sqle.getMessage());
}
catch (CreateException ce) {
    throw ce;
}
finally {
    if (ps != null) {
        try {
            ps.close();
        }
        catch (Exception e) {
        }
    }
}
return new SimpleKey(id);
}

/**
 * ejbLoad
 */

public void ejbLoad() throws javax.ejb.EJBException {

    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;

    String loadSQL = null;

    try {
        // call the common method to get the specific connection

```

```

    conn = getConnection();
}
catch (java.sql.SQLException sqle) {
    throw new EJBException("SQLException caught: " + sqle.getMessage());
}
catch (javax.resource.ResourceException re) {
    throw new EJBException(
        "ResourceException caught: " + re.getMessage());
}

// You need to determine which select statement to be used based on the
// AccessIntent type:
// If READ, then uses a normal SELECT statement. Otherwise uses a
// SELECT...FORUPDATE statement
// If your backend is SQLServer, then you can use different syntax for
// the FOR UPDATE clause.

if (intent.getAccessType() == AccessIntent.ACCESS_TYPE_READ) {
    loadSQL = "SELECT * FROM " + tableName + " WHERE id = ?";
}
else {
    loadSQL = "SELECT * FROM " + tableName + " WHERE id = ? FOR UPDATE";
}

SimpleKey key = (SimpleKey) getEntityContext().getPrimaryKey();

try {
    ps = conn.prepareStatement(loadSQL);
    ps.setInt(1, key.id);
    rs = ps.executeQuery();
    if (rs.next()) {
        id = rs.getInt(1);
        name = rs.getString(2);
    }
    else {
        throw new EJBException("Cannot load id = " + key.id);
    }
}
catch (SQLException sqle) {
    throw new EJBException(sqle.getMessage());
}
finally {
    try {
        if (rs != null)
            rs.close();
    }
    catch (Exception e) {
    }
    try {
        if (ps != null)
            ps.close();
    }
    catch (Exception e) {
    }
    try {
        if (conn != null)
            conn.close();
    }
    catch (Exception e) {
    }
}

/**
 * This method will use the AccessIntentService to get the access intent;
 * then gets the isolation level from the DataStoreHelper

```

```

    * and sets it in the connection spec; then uses this connection
    * spec to get a connection which has the specific connection
    * properties.
    **/

private Connection getConnection()
throws java.sql.SQLException, javax.resource.ResourceException, EJBException {

    // get current access intent object using EJB context
    intent = aiService.getAccessIntent(myEntityCtx);

    // Assume this bean only supports the pessimistic concurrency
    if (intent.getConcurrencyControl()
        != AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC) {
        throw new EJBException("Bean supports only pessimistic concurrency");
    }

    // determine correct isolation level for currently configured database
    // using DataStoreHelper
    int isoLevel =
        WSCallHelper.getDataStoreHelper(ds).getIsolationLevel(intent);
    connSpec = WSRRAFactory.createJDBCConnectionSpec();
    connSpec.setTransactionIsolation(isoLevel);

    // Get connection using connection spec
    Connection conn = ((WSDDataSource) ds).getConnection(connSpec);
    return conn;
}

```

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 116.

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 94 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));
}
```

Looking up data sources with resource references for relational access

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 a data source is looked up directly, the connection gets all default properties for the missing resource reference. For example, the *sharing-scope* is a shareable connection resulting in the possibility that the physical connection is the same each time the connection is requested from the data source. This situation can cause a multitude of problems if you expect unshareable connections.
- 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, you specify it in the resource references Bindings page.

Isolation level and resource reference:

This article discusses the criteria and effects of setting isolation level for various data access components.

In a J2EE 1.2 module, you can specify the isolation level at an enterprise bean method level, bean level, or module level. This capability has been removed from the J2EE 1.3 module. Because WebSphere

Application Server complies with the J2EE 1.3 specification, you cannot specify isolation level on the EJB method level or bean level for a J2EE 1.3 (or later version) module. Additionally, if a JDBC application, a bean-managed persistence (BMP) bean, or a servlet runs in a global transaction, and you are using shareable connections, you cannot set the isolation level on a connection.

When a container-managed persistence (CMP) 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 that this method or the bean has chosen. All other connection users can also use the access intent and application profile support to manage their concurrency control.

For all JDBC connections (excluding those used by CMP beans), you can specify an isolation level default on the resource reference. For shareable connections that run in global transactions, this default is the only way to set the *isolationLevel* object 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.

- For example, a data source is bound to two resource references: *jdbc/RRResRef* and *jdbc/RResRef*. *RRResRef* has the *RepeatableRead* isolation level defined. *RResRef* 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/RResRef* JNDI name to look up the data source.

Creating or changing a resource reference:

A resource reference supports application provider 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 ability insulates the application provider from the run time configuration, and simplifies the process of changing the run time configuration.

This article assumes that you have created an enterprise application whose resource references you want to update.

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 article describes how to update the resource references of an enterprise application using an assembly tool such the Application Server Toolkit (AST) or Rational Web Developer.

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) you want to change.
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**.

Binding to a data source:

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 DataSource 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: The combination of concurrency and access type determines the isolation level for the persistence manager. The actual isolation level depends upon the particular database, as shown in the following table.

AccessIntent profile	Isolation level						For Update
	DB2	Oracle*	SyBase	Informix	Cloudscape	SQL Server	
wsPessimistic-Update-WeakestLockAtLoad (Default policy)	RR	RC	RR	RR	RR	RR	No (*Oracle, Yes)
wsPessimistic-Update	RR	RC	RR	RR	RR	RR	Yes
wsPessimistic-Read	RR	RC	RR	RR	RR	RR	No
wsOptimistic-Update	RC	RC	RC	RC	RC	RC	No
wsOptimistic-Read	RC	RC	RC	RC	RC	RC	No

AccessIntent profile	Isolation level						For Update
	DB2	Oracle*	SyBase	Informix	Cloudscape	SQL Server	
wsPessimistic-UpdateNo-Collisions	RC	RC	RC	RC	RC	RC	No
wsPessimistic-Update-Exclusive	S	S	S	S	S	S	Yes

- RC = JDBC Read committed
- RR = JDBC Repeatable read
- S = JDBC Serializable
- * Note: Oracle does not support JDBC Repeatable Read (RR). Therefore, wsPessimisticUpdate-weakestLockAtLoad and wsPessimisticUpdate behave the same way on Oracle as do wsPessimisticRead and wsOptimisticRead. Because of an Oracle restriction, the OracleXADataSource JDBC class **cannot** run with an S transaction isolation level. So you cannot use this class to run an application containing enterprise beans whose access intent policies are configured to cause the bean to load with S isolation.
- Setting access intents per EJB method support is deprecated for Version 6.0. It is recommended that you set access intents only for the entire bean.

Structured Query Language (SQL) keywords and restrictions

The following table shows which SQL keywords are used during update intent locking and 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	FOR UPDATE OF	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
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

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 effect 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

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.

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(param1,param2,..paramn):method2(param1,param2,..paramn):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.

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 */ }
    }
}

```

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 verify 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;

        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.getEISProductVersion());
        }
    }
}
```

```

    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

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.

If you move from one of the earlier implementations of the J2EE Connector Architecture to the current implementation, be aware of the following:

- This version 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. Versions 5.0 and later support **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.

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 6.0 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, 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.

Tips for developing entity beans

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.

However, there are still situations where developers must 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 access 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.

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 FirstName 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 Data Objects

The Service Data Objects (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 backend data store connections or transactions.
- SDO is XML-integrated in that it provides services to easily convert retrieved data to and from XML format.

For a good introduction to SDO, refer to Introduction to Service Data Objects.

SDO is made up of two essential components, the DataGraph and the Data Mediator Service (DMS).

A DataGraph is a structured result returned in response to a service request. The DMS transforms the native back end query results into the DataGraph, which is independent of the originating back end 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 data objects 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.

DataObjects represent 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 450 for more information). If there are multiple values for a single attribute, they are held in a list. 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.

The Data Mediator Service (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 backend. 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 backend, it extracts the changes made from the original state of the DataGraph and flushes those changes to the backend. 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, the JDBC DMS and the Enterprise JavaBeans (EJB) DMS.

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 88, and “Service Data Objects : Resources for learning” on page 467

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 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 backend.

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:

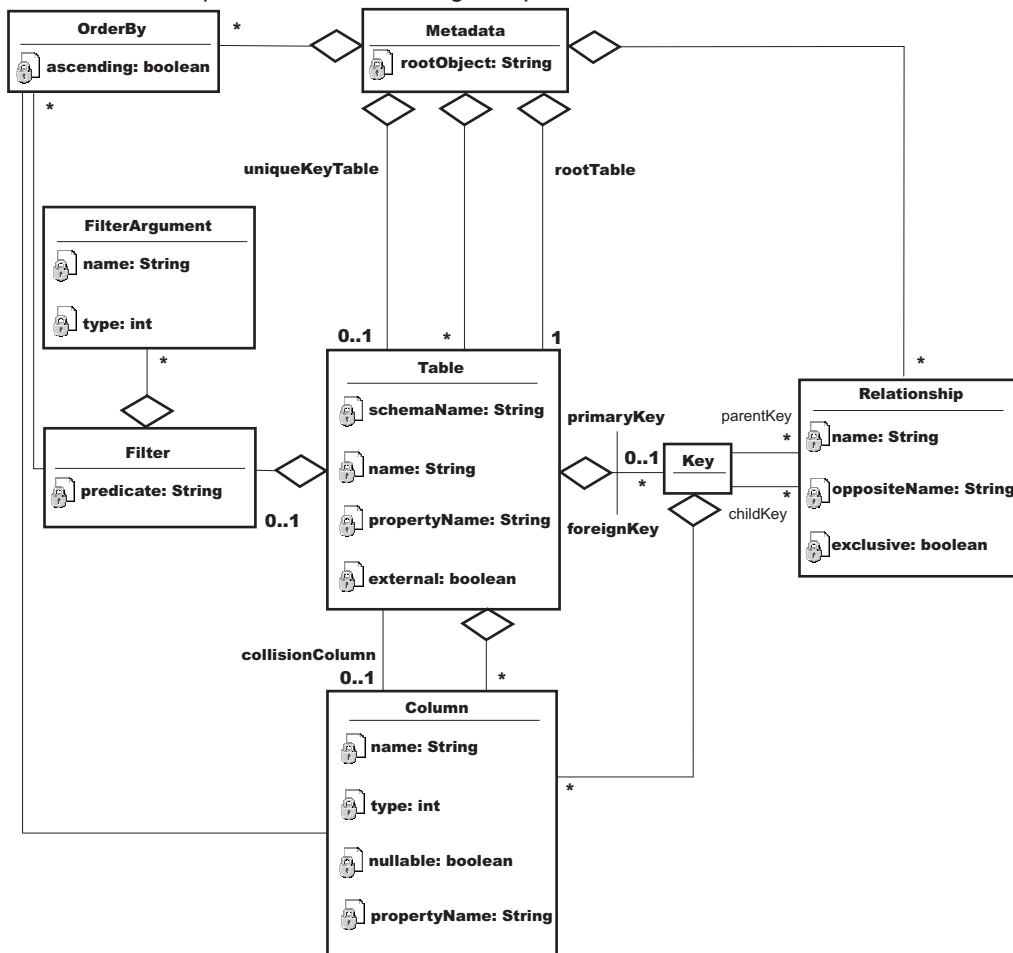


Table This represents a table within the target database and is composed of the following items:

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 supplied 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
QueryInfo queryInfo = factory.createQueryInfo();
queryInfo.setFilter("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.setQueryInfo(queryInfo);

..... // setting up mediator and such

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 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:

Known 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.

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) at instance creation. The ConnectionWrapper object contains the connection used by the JDBC DMS and indicates whether the mediator manages the current transaction. When the JDBC DMS is managing the transaction, it performs commit and rollback operations as required. However, it 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 created for use by the mediator, then the mediator assumes that there is an existing transaction and that it is being managed externally. No commit or rollback operations are performed by the connection wrapper in this case.

Optimistic concurrency control

Optimistic concurrency control (OCC) assumes that data collisions are rare when multiple clients update the same backend data concurrently. A *data collision* is when the data that was used to populate the client's DataGraph is changed in the database before submitting the changes made to the data. When a data collision does occur, the JDBC DMS causes an exception to occur. The client application must determine how to recover from the collision. For example, the client can re-read the data and restart the transaction. Once one exception occurs, there is no way of knowing whether there are more exceptions deeper in the DataGraph than the DataObject that cause the displayed exception. An OCC column is not created by default on a table, you must create one manually. You do this by adding an *OCC Integer* column to the table and then specifying 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.

Setting the OCC Column

In order to take advantage of OCC, you must have a column in the table created for this purpose. Here is how you do this:

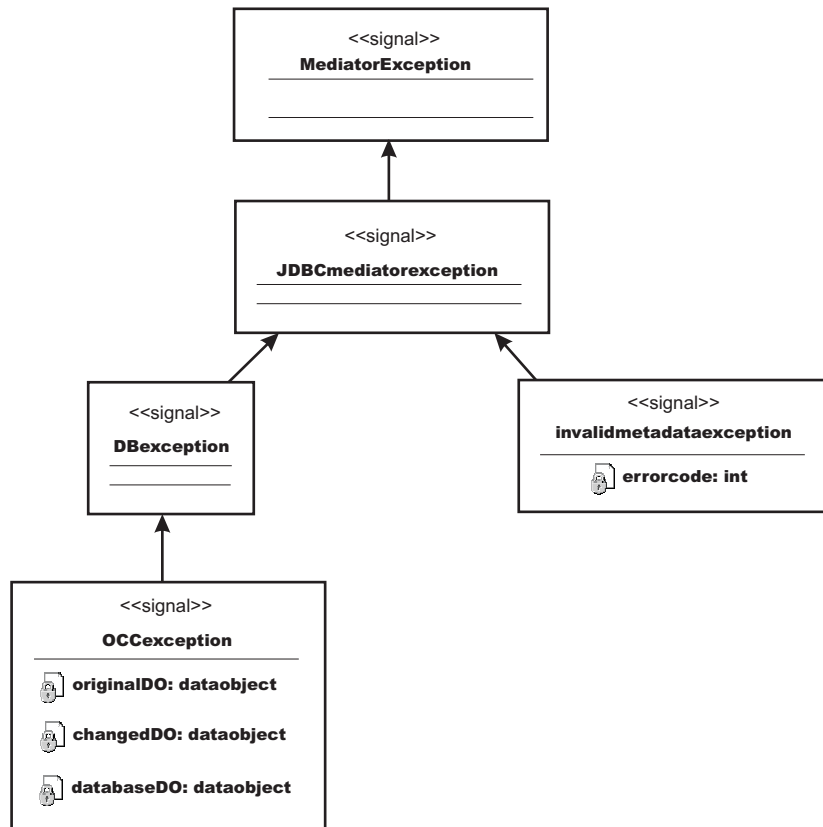
```

Column collisionColumn = table.addIntegerColumn("OCC_COUNT");
collisionColumn.setNullable(false);
table.setCollisionColumn(collisionColumn);

```

This creates the OCC column and makes sure that it does not allow null values, then designates it as the table collision column. If there is no OCC column defined for a table, then the DMS does not monitor and notify for update collisions.

You can see a code example that forces a collision to demonstrate detection and shows the exception that occurs as a result at “Example: OCC data collisions and JDBC mediator” on page 456.



JDBC mediator exceptions:

The Mediatorexception 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"));
}
```

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 467.

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: JavaService Faces and JDBC Mediator" in the information center.

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 and populating copy helper objects, and updating them, 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.

What happens at run time?

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 466.

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 Version 6.0 , you should 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:


```
remove (java.lang.Object)
findByPrimaryKey(<primaryKeyClass>)
```
- 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 461 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
Many:Many ejb relationship	No	No	No	No
EJB persistence inheritance	No	No	No	No
EJB RDB mapping multi table	No	Yes	No	Yes
EJB cmp field with converter	Yes	Yes	No	Yes

Transactional

- All mediator calls (including create) must be done within a transaction scope – either a user transaction or a container transaction.

Access Intent

- The mediator is designed to be used in disconnected caching scenarios where the `DataGraph` is the disconnected cache. It is assumed that you defined an optimistic update access intent for your application.
- When `applyChanges` writes directly to the database, the structured query language (SQL) update statements executed contain a predicate that compares optimistic predicate fields values to the old values of the fields. If the comparison returns false and the update fails, an exception occurs.
- 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.
- If optimistic predicate fields are defined in the EJB-RDB mapping, then you must retrieve at least one of these fields. The field should be updated either by the caller or a database trigger – the mediator does not automatically increment or set the field.

- 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.
- When retrieving or updating data through the EJB container, the access intent in effect on the EJB is used.

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 462.

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: The mediator query is an array of Enterprise JavaBeans (EJB) query statements. There is a query statement for each node type in the DataGraph. Each query specifies search predicates, the container managed persistence (CMP) fields or expressions to project into the DataObject,

and which EJB relationships to use to construct DataObject references. The EJB query statements follow normal rules and conventions for EJB query with the addition of the following rules:

- The FROM clause contains only one element. This is a path expression where the source is defined by a query statement earlier in the array of query statements.
- The first query statement FROM clause references either an EJB Abstract Schema Name (ASN) or an EJB collection parameter.
- An EJB type can appear in only one FROM clause except for subselects. The FROM clauses of the queries must form a hierarchy where no EJB type appears more than once, and there is one and only one path to an EJB type.
- 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 causes 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 causes 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:

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 461). 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 above 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 multi valued 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 above 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"
emp="//@root/@Emp.1 //@root/@Emp.0" />
      <Dept deptno="2" name="WBI_Sales" projects="//@root/@Project.1"
emp="//@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 EmpId , e.name as EmpName ,
       e.dept.deptno as DeptNo , e.dept.name as DeptName
from Emp as e

```

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 in((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 in((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.
}
```

DataGraph schema:

DataGraph schema created by the EJB mediator

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 schema defined by the caller

The caller can create the DataGraph either using Eclipse Modeling Framework (EMF) APIs to programmatically create the schema dynamically, or by generating static Java classes from an EMF model. In either case the Eclass of the root DataObject must be passed to the mediator. If the DataGraph pattern includes a “dummy” root, then this root must be part of the user supplied schema and the Eclass of the root is passed on the create call.

A user created schema can contain additional attributes and references that are not set by the mediator when constructing DataObjects. However, if any attribute or reference required by the mediator query does not exist or has an inconsistent type, an exception occurs.

The Eclass names used in this schema are by default the EJB ASN names. You can override this by passing to the mediator a *java.util.Map* argument.

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

This is the same as above. There must still be a dummy root SDO that contains all other SDO, except the Eclass name of the root can be any valid name (it does not have to be *DataGraphRoot*), and the reference names on the root can be any name.

ROOT_CONTAINS_SOME

There must be a dummy root SDO. An EJB relationship referenced in the query can be defined as an SDO contained reference. If not, then the dummy root MUST be defined to contain the SDO.

NO_DUMMY_ROOT

There is no dummy root SDO. The DataGraph root refers to the SDO returned by the first mediator query statement. Because a DataGraph can have only one root instance, if the query returns zero or more than one record, an exception occurs.

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.addColumn("CUSTFIRSTNAME");
custTable.addColumn("CUSTLASTNAME");
custTable.addColumn("CUSTSTREETADDRESS");
custTable.addColumn("CUSTCITY");
custTable.addColumn("CUSTSTATE");
custTable.addColumn("CUSTZIPCODE");
custTable.addColumn("CUSTAREACODE");
custTable.addColumn("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.addColumn("ORDERNUMBER");
orderNumber.setNullable(false);
```

```
orderTable.addColumn("ORDERDATE");
orderTable.addColumn("SHIPDATE");
Column custFKColumn = orderTable.addColumn("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,

```
ConnectionFactory 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 468, 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.websphere.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 <code>java.util.Map</code> 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

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.SQLException`, 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.RemoteException {
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.RemoteException
    {
        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.RemoteException 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.RemoteException
remoteException)
{
java.rmi.RemoteException re = remoteException;
Throwable t = null;
while (true) {
t = re.getCause();
try { re = (java.rmi.RemoteException)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.RemoteException
{
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
getContext().setRollbackOnly();
throw new java.rmi.RemoteException("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 *RemoteException* 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 TCP/IP 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) +
                " " + rs.getString(2));
        }
//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.rmi.RemoteException;
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.RemoteException {
        getDS();
    }
    /**
    * ejbCreate method
    * @exception javax.ejb.CreateException
    * @exception java.rmi.RemoteException
    */
    public void ejbCreate() throws javax.ejb.CreateException, java.rmi.RemoteException {}
    /**
    * ejbPassivate method
    * @exception java.rmi.RemoteException
    */
    public void ejbPassivate() throws java.rmi.RemoteException {}
    /**
    * ejbRemove method
    * @exception java.rmi.RemoteException
    */
    public void ejbRemove() throws java.rmi.RemoteException {}

    /*****
    /** 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 retrievable.
    *****/
    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) + " " + rs.getString(2));
        }
    }
}
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.RemoteException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws java.rmi.RemoteException {
    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.rmi.RemoteException;
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.RemoteException {
    getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.RemoteException {}
/**
 * ejbPassivate method
 * @exception java.rmi.RemoteException
 */
public void ejbPassivate() throws java.rmi.RemoteException {}
/**
 * ejbRemove method
 * @exception java.rmi.RemoteException
 */
public void ejbRemove() throws java.rmi.RemoteException {}

/*******
/** 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) + ", " +
rs.getString(1) + " " + rs.getString(2));
            }
        }
        //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.RemoteException

```

```

*/
public void setSessionContext(javax.ejb.SessionContext ctx) throws java.rmi.RemoteException {
    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. 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.rmi.RemoteException;
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
 * @exception java.rmi.RemoteException
 * ejbActivate calls getDS(), which performs the
 * JNDI lookup for the datasource.
 */
public void ejbActivate() throws java.rmi.RemoteException {
    getDS();
}

/**
 * ejbCreate method for a BMP entity bean
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception javax.ejb.CreateException

```

```

* @exception java.rmi.RemoteException
*/
public WebSphereSamples.ConnPool.EmployeeBMPKey ejbCreate(
    String empNo, String firstName, String lastName, String middleInit, int edLevel)
    throws javax.ejb.CreateException, java.rmi.RemoteException {

    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 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 (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 java.rmi.RemoteException
 * @exception javax.ejb.FinderException
 */
public WebSphereSamples.ConnPool.EmployeeBMPKey
    ejbFindByPrimaryKey(WebSphereSamples.ConnPool.EmployeeBMPKey primaryKey)
        throws java.rmi.RemoteException, javax.ejb.FinderException {
    loadByEmpNo(primaryKey.empNo);
    return primaryKey;
}
/**
 * ejbLoad method
 * @exception java.rmi.RemoteException
 */
public void ejbLoad() throws java.rmi.RemoteException {
    try {
        EmployeeBMPKey pk = (EmployeeBMPKey) entityContext.getPrimaryKey();
        loadByEmpNo(pk.empNo);
    } catch (FinderException fe) {
        throw new RemoteException("Cannot load Employee state from database.");
    }
}
/**
 * ejbPassivate method
 * @exception java.rmi.RemoteException
 */
public void ejbPassivate() throws java.rmi.RemoteException {}
/**
 * ejbPostCreate method for a BMP entity bean
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception java.rmi.RemoteException
 */
public void ejbPostCreate(String empNo, String firstName, String lastName,
    String middleInit, int edLevel) throws java.rmi.RemoteException {}
/**
 * ejbRemove method
 * @exception java.rmi.RemoteException
 * @exception javax.ejb.RemoveException
 */
public void ejbRemove() throws java.rmi.RemoteException, 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 RemoteException("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 RemoteException(se.getMessage());
}
catch (SQLException sq) {
    System.out.println("SQL Exception during get connection or process SQL: " +
        sq.getMessage());
    throw new RemoteException(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());
        }
    }
}
}
try {
    con = ds.getConnection();
    ps = con.prepareStatement(sql);
    ps.setString(1, empNo);
    if (ps.executeUpdate() != 1){
        throw new RemoteException("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 RemoteException(se.getMessage());
}
catch (SQLException sq) {
    System.out.println("SQL Exception during get connection or process SQL: " +
        sq.getMessage());
    throw new RemoteException(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());
        }
    }
}
}
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 RemoteException(se.getMessage());
}
catch (SQLException sq) {

    System.out.println("SQL Exception during get connection or process SQL: " +
        sq.getMessage());

    throw new RemoteException(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, firstme, 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 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 (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
 * @exception java.rmi.RemoteException
 */

```



```

public void setEntityContext(javax.ejb.EntityContext ctx) throws java.rmi.RemoteException {
    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
 * @exception java.rmi.RemoteException
 */
public void unsetEntityContext() throws java.rmi.RemoteException {
    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 exceptions resulting from foreign key conflicts, or deadlock when entity beans are configured for optimistic concurrency control:

Exceptions resulting from foreign key conflicts, which signify 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.

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.

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 with either 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
-dbvendordb2\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 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.

Note: DB2 V8.1 fix pack 6 provides a new db2optimize option to improve application performance. If your SQLJ application uses the explicit connection context instead of the default context, you can take advantage of this db2optimize option in the SQLJ customizer tool to enable *context caching*. The following example code demonstrates proper syntax for the option:

```
sqlj -db2optimize SQLJTransactionTest.sqlj
db2sqljcustomize -url jdbc:db2://localhost:50000/jtest -user dbuser1 -password dbpwd1
SQLJTransactionTest_SJProfile0.ser
```

If you want to use this option 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.

Assembling data access applications

A data access application uses a data source to access data from a database.

This article assumes that you have created an enterprise application that has an EJB module and that you want this application to access one or more resources.

To access a resource, you associate a data source with a JDBC provider that supplies the specific JDBC driver implementation class. The data source represents the J2EE Connector Architecture (JCA) connection factory for the relational resource adapter.

You can create multiple data sources associated with the same JDBC provider. Each JDBC provider supports the ConnectionPoolDataSource or XADataSource interfaces. These interfaces enable the application to run in a single-phase or two-phase transaction protocol.

This article describes how to configure resource references for an application so the application can access resources using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that the **J2EE** capability is enabled.
3. Define the resource reference attributes.
4. Bind this resource reference to a resource like a data source.
5. Configure isolation level, access intent assembly settings.
 - 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.

Files for the updated application are shown in the Project Explorer view.

After testing your application, you can deploy your application to an application server.

Migrating a Version 4.0 data access application to Version 6.0

To use the connection management infrastructure in WebSphere Application Server Version 6.0, 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.

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 Creating a resource reference.
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 Creating a resource reference.
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. Create a JDBC provider and a data source object following the instructions in Creating a JDBC provider and data source.
2. Install the application, following the instructions in Installing a new application and bind the resource references to the data source that you created.

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

Before installing a data access application into the WebSphere Application Server environment, you must first ensure that the appropriate database objects are available. This action includes creating and configuring any databases or tables required, setting necessary configuration parameters to handle expected load, and configuring any necessary JDBC providers and data source objects for servlets, enterprise beans, and client applications to use.

1. If your database configuration does not already exist:
 - a. Create a database to hold the data.

- b. Create tables required by your application.

If your application uses entity enterprise 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 does not use entity beans

You must use your database server interfaces to create the tables.

- c. See Minimum required properties for vendor-specific data sources for certain vendors' databases requirements.
2. Migrate Version 4.0 data access applications
3. If your enterprise application contains a Web application or an EJB application that uses connection pooling to access a relational database, see "Creating and configuring a JDBC provider and data source" on page 524.
4. If your application requires access to a non-relational database, you need to configure a resource adapter and a connection factory rather than a JDBC provider and a data source.
5. If your enterprise application contains an application client that accesses a relational database, see *Configuring data access for application clients*.
6. Consider the security of lookups with component managed authentication. See *Security of lookups with component managed authentication* for more information.

Relationship of assembly and administrative console data access settings

This article provides miscellaneous tips for using supported databases. See also the related links.

Always consult the product documentation for a list of the database brands and versions that are supported by your particular WebSphere Application Server version, edition, and FixPak.

Notes about various databases

- When using local DB2 databases for data access by session clients on AIX Version 4.3.3 or later versions, in some cases you cannot establish multiple connections for session clients. This is because AIX, by default, does not permit 32-bit applications to attach to more than 11 shared memory segments per process. Of these 11 shared segments, a maximum of 10 can be used for local DB2 connections. To use EXTSHM with DB2 and avoid stale connections when there are large numbers of session clients, do the following:
 - In DB2 client environment (that is the WebSphere Application Server run time environment in this case):

```
export EXTSHM=ON
```

- In DB2 UDB Server environment:

```
export EXTSHM=ON
db2set DB2ENVLIST=EXTSHM
```

- When using Sybase 11.x, you might encounter the following error when HttpSession persistence is enabled:

```
DBPortability W Could not create database table: "sessions"
com.sybase.jdbc2.jdbc.SybSQLException: The 'CREATE TABLE' command is not
allowed within a multi-statement transaction in the 'database_name' database
```

where *database_name* is the name of the database for holding sessions.

If you encounter the error, issue the following commands at the Sybase command line:

```
use database_name
go
sp_dboption db,"ddl in tran ",true
go
```

- Sybase 12.0 does not support local transaction modes with a JTA enabled data source. To use a connection from a JTA enabled data source in a local transaction, install Sybase patch EBF9422.

Additional administrative tasks for specific databases

For your convenience, this article provides instructions for enabling some popular database drivers, and performing other administrative tasks often required to provide data access to applications running on WebSphere Application Server. These tasks are performed outside of the WebSphere Application Server administrative tools, often using the database product tools. Always refer to the documentation accompanying your database driver as the authoritative and complete source of driver information.

See the Supported hardware, software, and APIs for the latest information about supported databases, drivers, and operating systems.

Enabling JDBC 2.0

Ensure that your operating system environment is set up to support JDBC 2.0. This action is required to use data sources created through WebSphere Application Server.

The following steps make it possible to find the appropriate JDBC 2.0 driver for use with WebSphere Application Server administration:

Enabling JDBC 2.0 with DB2 on Windows NT systems

To enable JDBC 2.0 use on Windows NT systems:

- For DB2 Version 7.2
 1. Stop the DB2 JDBC Applet Server service.
 2. Run the following batch file:

```
SQLLIB\java12\usejdbc2.bat
```
 3. Stop WebSphere Application Server (if it is running) and start it again.
- For DB2 Version 8.1
 - JDBC 2.0 is supported by default, there are no additional steps for you to perform.

Perform the steps once for each system.

Determining the level of the JDBC API in use for DB2 on Windows NT systems

To determine the JDBC level in use on your system:

- For DB2 Version 7.2
 - If JDBC 2.0 is in use, this file exists:

```
SQLLIB\java12\inuse
```
 - If JDBC 1.0 is in use, this file exists:

```
SQLLIB\java11\inuse
```

or no *java11* directory exists.
- For DB2 Version 8.1
 - Go to directory *SQLLIB\samples\java*, compile and run the class *db2JDBCVersion.java*.

Enabling JDBC 2.0 with DB2 on UNIX systems

- For DB2 Version 7.2
 - Before starting WebSphere Application Server, call *\$INSTHOME/sql/lib/java12/usejdbc2* to use JDBC 2.0. For convenience, you might want to put this in your root user's startup script. For example, on AIX, add the following to the root user's *.profile*:

```
if [ -f /usr/lpp/db2_07_01/java12/usejdbc2 ] ; then
    . /usr/lpp/db2_07_01/java12/usejdbc2
fi
```
- For DB2 Version 8.1
 - JDBC 2.0 is supported by default, there are no additional steps for you to perform.

Determining the level of the JDBC API in use for DB2 on UNIX systems

- For DB2 Version 7.2
 - To determine if you are using JDBC 2.0, you can echo *\$CLASSPATH*. If it contains *\$INSTHOME/sql/lib/java12/db2java.zip* then JDBC 2.0 is in use.

If it contains

```
$INSTHOME/sql1lib/java/db2java.zip
```

then JDBC 1.0 is in use.

- For DB2 Version 8.1
 - Go to directory *sql1lib/samples/java*, compile and run the class *db2JDBCVersion.java*.

Sourcing the db2profile script on UNIX systems

Before starting WebSphere Application Server to host applications requiring data access, source the *db2profile*:

```
. ~db2inst1/sql1lib/db2profile
```

where *db2inst1* is the user created during DB2 installation.

Using Java Transaction API drivers

Instructions are available for using Java Transaction API (JTA) drivers on particular operating systems. See your operating system documentation for more information.

The goal of this section is to provide information about the steps that make DB2 work well with applications utilizing XA classes -- that is, those whose *dataSourceClasses* implement *javax.sql.XADataSource*.

Using Java Transaction API drivers for DB2 on Windows NT systems

To enable JTA drivers for DB2 on Windows NT systems, follow these steps:

1. Bind the necessary packages to the database. From the **DB2 Command Line Processor** window, issue the following commands:

```
db2=> connect to mydb2jta
db2=> bind db2home\bnd\@db2cli.lst
db2=> bind db2home\bnd\@db2ubind.lst
db2=> disconnect mydb2jta
```

where *mydb2jta* is the name of the database to enable for the JTA, and *db2home* is the DB2 root installation directory path (for example, *D:\ProgramFiles\SQLLIB\bnd\@db2cli.lst*).

2. Specify the following settings when you use an IBM WebSphere Application Server administrative client (such as the administrative console) to configure a JDBC driver:
 - **Server class path** = %DB2_ROOT%/Sql1lib/java/db2java.zip
 - **Implementation class name** = COM.ibm.db2.jdbc.DB2XADataSource

Using Java Transaction API drivers for DB2 on UNIX systems

To enable JTA drivers on UNIX systems, follow these steps:

1. Stop all DB2 services.
2. Stop the IBM WebSphere Application Server administrative service.
3. Stop any other processes that use the *db2java.zip* file.
4. Make sure that you already enabled JDBC 2.0.
5. Start the DB2 services.
6. Bind the necessary packages to the database. From the DB2 command-line process or window, issue the following commands:

```
db2=> connect to mydb2jta
db2=> bind db2home\bnd\@db2cli.lst
db2=> bind db2home\bnd\@db2ubind.lst
db2=> disconnect mydb2jta
```

7. Specify the following settings when you use an IBM WebSphere Application Server administrative client (such as the administrative console) to configure a JDBC driver:
 - **Server class path** = \$INSTHOME/sql1lib/java12/db2java.zip
For example, if *\$INSTHOME* is */home/test*, the path will be */home/test/sql1lib/java12/db2java.zip*
 - **Implementation class name** = COM.ibm.db2.jdbc.DB2XADataSource

For Oracle 8.1.7 two phase commit support

You can use the Oracle 8.1.7 thin driver for JTA two-phase support with the following restrictions:

- The thin driver that comes shipped with 8.1.7 might or might not work. Future patches from Oracle might work as well, but are not tested. The driver that was available from the Oracle Technology Network Web site as of February 20, 2001 does work and is the recommended driver. Later versions on this Web site are expected to work, but are not tested.

To obtain the driver from the Oracle support Web site, visit:

<http://technet.oracle.com/>

You need to be a registered user for the Oracle Technology Network to get the driver from this site. Contact Oracle for access. After you have access download the 8.1.7 driver for the platforms you use and follow the instructions for installing the new driver.

- You must use the 8.1.7 driver with 8.1.7 databases, 8.1.6 databases do not support the `recover()` and `forget()` methods and other problems are encountered running with 8.1.6. Oracle does not support JTA with 8.1.6.
- For Oracle, you can only use JTA with container-managed persistence (CMP) beans.
- For the bean to create the table, you must start the bean with the JTA set to *false*. After the bean creates the table, you can set the JTA back to *true*.
- Configure an entity bean that accesses Oracle with JTA set to *true* as follows:
 - Click **deployment descriptor properties** > **Transactions** > Remote tab. Set the Transaction Attribute to *TX_REQUIRED*.
 - Click **Isolation** > Remote tab. Set the Isolation Level to *TRANSACTION_READ_COMMITTED*.
- Configure a session bean that is used with an entity bean that accesses Oracle with JTA set to *true* as follows:
 - Click **deployment descriptor properties** > **Transactions** > Remote tab. Set the Transaction Attribute to *TX_BEAN_MANAGED*.
 - Click **Isolation** > Remote tab. Set the Isolation Level to *TRANSACTION_READ_COMMITTED*.

Using Java Transaction API drivers for Sybase products on AIX systems

To enable Java Transaction API (JTA) drivers for use with Sybase products on the AIX operating system, follow these steps:

1. Enable the Data Transaction Manager (DTM) by issuing these commands (one per line) at a command prompt:

```
isql -Usa -Ppassword -Sservername
sp_configure "enable DTM", 1
go
```

2. Stop the Sybase Adaptive Server database and start it again.

3. Grant the appropriate role authorization to the enterprise bean user at a command prompt:

```
isql -Usa -Ppassword -Sservername
grant role dtm_tm_role to EJB
go
```

Notes about Sybase Java Transaction API drivers

Do not use a Sybase Java Transaction API (JTA) connection in an enterprise bean method with an unspecified transaction context. A Sybase JTA connection does not support the local transaction mode. The implication is that you must use the Sybase JTA connection in a global transaction context.

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 then run the `ddl` file on your database server to create the tables.

Following is an example of how to use such a `Table.ddl` file for DB2, on Windows and z/OS:

1. The container-managed bean (CMP) enterprise bean JAR file has a *Table.ddl* file that the assembly tool generates. Extract the *Table.ddl* file to a working directory such as *C:\temp*.
2. Run this command -- **C:\temp>db2cmd** A new command window appears in which you enter the following commands.
3. Run this command -- **C:\temp>db2 connect to dbname**
4. Run this command -- **C:\temp>db2 -tf Table.ddl** The command runs and creates tables for your CMP enterprise bean.
5. Run this command -- **C:\temp>db2 disconnect all**

Note: If you use DB2 on Unix, use these same commands. Simply run them from a user with permissions for DB2, rather than from a DB2 command window.

Installing J2EE Connector resource adapters

1. Click **Resources**.
2. Click **Resource Adapters**.
3. Select the *scope* at which you want to define this resource adapter. (This scope becomes the scope of your connection factory.) You can choose cell, node, cluster, or server. For more information, see "Administrative console scope settings" in the information center.
4. Click **Install RAR**. The Install RAR button opens a dialog that enables you to install a J2EE Connector Architecture (JCA) connector and create a resource adapter for it. You can also use the **New** button, but the New button creates only a new resource adapter (the JCA connector must already be installed on the system).

Note: When installing a RAR file using this dialog, the scope you define on the Resource Adapters page has no effect on where the RAR file is installed. You can install RAR files only at the *node* level. The node on which the file is installed is determined by the scope on the **Install RAR** page. (The scope you set on the Resource Adapters page determines the scope of the new resource adapters, which you can install at the server, node, or cell level.)

5. Browse to find the appropriate RAR file.
 - If your RAR file is located on your local workstation, select **Local path** and browse to find the file.
 - If your RAR file is located on your server, select **Server path** and specify the fully qualified path to the file.
6. Click **Next**.
7. Enter the resource adapter name and any other properties needed under *General Properties*. If you install a J2C Resource Adapter that includes *Native path* elements, consider the following: If you have more than one native path element, and one of the native libraries (native library A) is dependent on another library (native library B), then you must copy native library B to a *system* directory. Because of limitations on Windows NT and most Unix platforms, an attempt to load a native library does not look in the current directory.
8. Click **OK**.

Installing resource adapters within applications:

1. Assemble an application with resource adapter archive (RAR) modules in it. See Assembling applications.
2. Install the application following the steps in Installing a new application. In the **Map modules to servers** step, specify target servers or clusters for each RAR file. Be sure to map all other modules that use the resource adapters defined in the RAR modules to the same targets. Also, specify the Web servers as targets that serve as routers for requests to this application. The plug-in configuration file (*plugin-cfg.xml*) for each Web server is generated based on the applications that are routed through it.

Note: When installing a RAR file onto a server, WebSphere Application Server looks for the manifest (MANIFEST.MF) for the connector module. It looks first in the *connectorModule.jar* file for the RAR 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.

To ensure that the installed connector module finds the classes and resources that it needs, check the **Class path** setting for the RAR using the console. For more information, see “Resource Adapter settings” on page 522 and “WebSphere relational resource adapter settings” on page 393.

3. Click **Finish** > **Save** to save the changes.
4. Create connection factories for the newly installed application.
 - a. Open the administrative console.
 - b. Click **Applications** > **Enterprise Applications** > *application name*.
 - c. Click **Connector Modules** in the Related Items section of the page.
 - d. Click *filename.rar*.
 - e. Click **Resource adapter** in the Additional Properties section of the page.
 - f. Click **J2C Connection Factories** in the Additional Properties section of the page.
 - g. Click on an existing connection factory to update it, or **New** to create a new one.

If you install a J2C Resource Adapter that includes *Native path* elements, consider the following: If you have more than one native path element, and one of the native libraries (native library A) is dependent on another library (native library B), then you must copy native library B to a *system* directory. Because of limitations on Windows NT and most Unix platforms, an attempt to load a native library does not look in the current directory.

After you create and save the connection factories, you can modify the resource references defined in various modules of the application and specify the Java Naming and Directory Interface (JNDI) names of the connection factories wherever appropriate.

Note: A given native library can only be loaded one time for each instance of the Java virtual machine (JVM). Because each application has its own classloader, separate applications with embedded RAR files cannot both use the same native library. The second application receives an exception when it tries to load the library.

If any application deployed on the application server uses an embedded RAR file that includes native path elements, then you must always ensure that you shut down the application server cleanly, with no outstanding transactions. If the application server does not shut down cleanly it performs *recovery* upon server restart and loads any required RAR files and native libraries. On completion of recovery, do not attempt any application-related work. Shut down the server and restart it. No further recovery is attempted by the application server on this restart, and normal application processing can proceed.

Resource Adapters collection:

This page contains the list of installed and configured resource adapters and is used to install new resource adapters, create additional configurations of installed resource adapters or delete resource adapter configurations.

A resource adapter is an implementation of the J2EE Connector Architecture (JCA) Specification that provides access for applications to resources outside of the server or provides access for an enterprise information system (EIS) to applications on the server. It can provide application access to resources such as DB2, CICS, SAP and PeopleSoft. It can provide an EIS with the ability to communicate with message-driven beans that are configured on the server. Some resource adapters are provided by IBM; however, third party vendors can provide their own resource adapters. A resource adapter implementation is provided in a resource adapter archive file; this file has an extension of *.rar*. A resource adapter can be

provided as a standalone adapter or as part of an application, in which case it is referred to as an embedded adapter. Use this panel to install a standalone resource adapter archive file. Embedded adapters are installed as part of the application installation. This panel can be used to work with either kind of adapter.

To view this administrative console page, click **Resources >Resource Adapters**.

Scope:

Specifies the level to which this resource adapter is visible. For general information, see *Administrative console scope settings* in the Related Reference section.

Some considerations that you should keep in mind for this particular panel are:

- Changing the scope enables you to see which resource adapter definitions exist at that level.
- Changing the scope **does not** have any effect on installation. Installations are always done under a scope of **node**, no matter what you set the scope to.
- When you create a new resource adapter from this panel, you must change the scope to what you want it to be **before** clicking **New**.

Name:

Specifies the name of the resource adapter.

A string with no spaces meant to be a meaningful text identifier for the resource adapter.

Data type	String
------------------	--------

Resource Adapter settings:

Use this page to specify settings for a Resource Adapter.

A resource adapter is an implementation of the J2EE Connector Architecture (JCA) Specification that provides access for applications to resources outside of the server or provides access for an enterprise information system (EIS) to applications on the server. It can provide application access to resources such as DB2, CICS, SAP and PeopleSoft. It can provide an EIS with the ability to communicate with message driven beans that are configured on the server. Some resource adapters are provided by IBM; however, third party vendors can provide their own resource adapters. A resource adapter implementation is provided in a resource adapter archive file; this file has an extension of *.rar*. A resource adapter can be provided as a standalone adapter or as part of an application, in which case it is referred to as an embedded adapter. Use this panel to install a standalone resource adapter archive file. Embedded adapters are installed as part of the application installation.

To view this administrative console page, click **Resources >Resource Adapters > resource_adapter**.

Scope:

Specifies the level to which this resource definition is visible. For general information, see *Administrative console scope settings* in the Related Reference section.

The Scope field is a read only string field that shows where the particular definition for a resource adapter is located. This is set either when the resource adapter is installed (which can only be at the node level) or when a new resource adapter definition is added.

Name:

Specifies the name of the resource adapter definition.

This property is required.

A string with no spaces meant to be a meaningful text identifier for the resource adapter.

Data type String

Description:

Specifies a text description of the resource adapter.

A free-form text string to describe the resource adapter and its purpose.

Data type String

Archive path:

Specifies the path to the RAR file containing the module for this resource adapter.

This property is required.

Data type String

Class path:

Specifies a list of paths or JAR file names which together form the location for the resource adapter classes.

This includes any additional libraries needed by the resource adapter. The resource adapter code base itself is automatically added to the class path, but if anything outside the RAR is needed it can be specified here.

Data type String

Native path:

Specifies a list of paths which forms the location for the resource adapter native libraries.

The resource adapter code base itself is automatically added to the class path, but if anything outside the RAR is needed it can be specified here.

Data type String

ThreadPool Alias:

Specifies the name of a thread pool that is configured in the server that is used by the resource adapter's Work Manager.

If there is no thread pool configured in the server with this name, the default configured thread pool instance, named *Default*, is used. This property is only necessary if this resource adapter uses Work Manager.

Data type

String

Pretesting pooled connections to ensure validity

When a database fails, pooled connections that are not valid might exist in the free pool. This scenario is likely to occur when you have a failingConnectionOnly purge policy, which mandates that only failing connections be removed from the pool. Whether the remaining connections in the pool are valid varies with the cause of the failure. Connection pretesting is a way to test connections from the free pool before giving them to the client.

If your application uses pooled connections, you can enable the PreTest Connections feature in the administrative console to help prevent your application from obtaining connections that are no longer valid.

The feature is particularly useful for routine database outages. Because these outages are usually scheduled for periods of low use, connections to the database are likely to be in the free pool rather than in active use. Active connections are not pretested; pretesting impedes performance during normal operation. Pretesting ensures that users do not waste time trying to resume connections that became bad before the outage.

1. In the administrative console, click **Resources > JDBC providers**.
2. Select a provider and click **Data Sources** under Additional properties.
3. Select a data source and click **WebSphere Application Server data source properties** under Additional properties.
4. Select the **PreTest Connections** check box.
5. Type a value for the PreTest Connection Retry Interval, which is measured in seconds. This property determines the frequency with which a new connection request is made after a pretest operation fails.
6. Type a valid SQL statement for the PreTest SQL String. Use a reliable SQL command, with minimal performance impact; this statement is processed each time a connection is obtained from the free pool.

For example, you might specify `SELECT COUNT(*) FROM TESTTABLE`. (For an Oracle database, use `SELECT USER FROM DUAL`.)

Creating and configuring a JDBC provider and data source

Determine which version of data source you need. If you are using the Enterprise JavaBean (EJB) 1.0 specification or the Java Servlet 2.2 specification, you need the version 4.0 data source; see the topic [Data Sources \(Version 4\)](#). If you are using more advanced releases of these specifications, see the topic [Data Source collection](#).

1. Create a JDBC provider.

From the administrative console, see [Creating a JDBC provider using the administrative console](#).

OR

Using the wsadmin scripting client, see ["Configuring a JDBC provider using scripting"](#) in the information center.

OR

Using the Java Management Extensions (JMX) API, see [Creating a JDBC provider and data source using the Java Management Extensions API](#).

2. Create a data source.

From the administrative console, see [Creating a data source using the administrative console](#).

OR

Using the wsadmin scripting client, see ["Configuring new data sources using scripting"](#) in the information center. (For V4 data sources, see ["Configuring new WAS40 data sources using scripting."](#))

OR

Using the JMX API, see [Creating a JDBC provider and data source using the Java Management Extensions API](#).

3. Bind the resource reference. See [Binding to a data source](#)
4. Test the connection (for non-container-managed persistence usage). See [Test connection](#).

Note: When you save the data source configuration, it is saved in the *resource.xml* file. You should not need to modify the **jdbc-resource-provider-templates.xml**. However, special consideration should be taken if you need to update the **jdbc-resource-provider-templates.xml** file. For details, consult “J2EE Connector Architecture migration tips” on page 443.

Verifying a connection:

Many connection problems can be easily fixed by verifying some configuration parameters. This article provides a checklist of steps that you must complete to enable a successful connection. Click on the link for more information on a specific step.

If your connection is still not successful after completing these steps and reviewing the applicable information, check the SystemOut.log for warning or exception messages. Then use the technical support search function to find known problems.

1. Create the authentication data alias.
2. Create the JDBC provider.
3. Create a data source.
4. Save the data source.
5. If you created a new authentication alias, restart the server for which you need to verify connectivity.
6. Test the connection

You can test your connection from the data source collection view or the data source details view. Access either view in the administrative console, and then select a connection from the list. Click the **Test Connection** button on the connection.

Creating and configuring a JDBC provider using the administrative console:

An application installed on WebSphere Application Server accesses a relational database through a JDBC provider, which is essentially a system-level software driver. For at-a-glance information on which provider type is appropriate for your database configuration and application requirements, see the JDBC provider table.

You can easily establish a JDBC provider from the administrative console.

1. Open the administrative console.
2. Click **Resources > JDBC Providers**.
3. Select the *scope* of your definition. (This scope becomes the scope of your data source.) You can choose cell, node, cluster, or server. For more information, see “Administrative console scope settings” in the information center.
4. Click **New**.

Note: If Java script is disabled for your browser, you do not see the three drop-down lists that are described in the next three steps (for database type, provider type, and implementation type). Instead, you see a single drop-down box that lists *all* JDBC provider choices simultaneously (inclusive of every database, provider, and implementation type).

5. Use the first drop-down list to select the database type of the JDBC provider you need to create. If the list of supported JDBC provider types does not include the JDBC provider that you want to use, select the **User-Defined JDBC Provider**. Then consult the JDBC provider vendor’s documentation for information on specific properties required for data sources associated with this provider, and skip to step eight of this list.

6. From the second drop-down list, select your JDBC provider type.
7. From the third drop-down list, select the implementation type necessary for your application. If your application does not require that connections support two-phase commit transactions, choose **Connection Pool Data Source**. Choose **XA Data Source**, however, if your application requires connections that support two-phase commit transactions. Applications using this data source configuration have the benefit of container-managed transaction recovery.
8. Click **Next** to view the general property settings page for your JDBC provider.
9. Ensure that all required properties have valid values. For more information, see JDBC Provider settings.
10. Click **Apply** to view the page with your new JDBC provider settings. Note that two active data source links now appear under the **Additional Properties** heading on this page. To set up a data source, click the link that corresponds to the type required by your application, the Version 4 data source or the later version data source. (For more information, refer to the section entitled "Choice of data source" in the "Data sources" on page 400 topic.)
11. Click **OK** to return to the JDBC providers page, where your new JDBC provider appears in the list.

For more information about creating a data source for association with your JDBC provider, see "Creating and configuring a data source using the administrative console" on page 531.

JDBC Provider collection:

Use this page to view a JDBC provider.

To view this administrative console page, click **Resources > JDBC Providers** in the console navigation tree.

Notice the *Scope* of your JDBC provider. If you pick anything other than the default of *Node* the provider might not be available in other scope contexts. New items created in this view are created within the selected scope.

Name:

Specifies a text identifier for this provider.

For example, this field can be *DB2 JDBC Provider (XA)*.

Data type String

Description:

Specifies a text string describing this provider.

Data type String

JDBC provider settings:

Use this page to create or modify JDBC provider settings.

To view this administrative console page, click **Resources > JDBC Providers > JDBC_provider**.

Name:

Specifies the name of the resource provider.

Data type String

Description:

Specifies a text description for the resource provider.

Data type String

Class path:

Specifies a list of paths or JAR file names which together form the location for the resource provider classes.

For example, *D:/SQLLIB/java/db2java.zip*.

Class path entries are separated by using the ENTER key and must not contain path separator characters (such as ';' or ':'). Class paths contain variable (symbolic) names which you can substitute using a variable map. Check the driver installation notes for the specific required JAR file names.

Data type String

Native Library Path:

Specifies a list of paths that forms the location for the resource provider native libraries.

Native path entries are separated by using the ENTER key and must not contain path separator characters (such as ';' or ':'). Native paths can contain variable (symbolic) names which you can substitute using a variable map.

Data type String

Implementation class name:

Specifies the Java class name of the JDBC driver implementation.

This class is available in the driver file mentioned in the class path description above. For example, *COM.ibm.db2.jdbc.DB2XADataSource*.

Note: If you modify the implementation class name of the JDBC provider after you have created the provider, you might disconnect the provider from the template used to create it. As a result, data sources created from this JDBC provider do not have an associated template; you must manually configure a working data source through setting custom properties.

Data type String

New JDBC Provider:

Use this page to create a new JDBC provider.

To view this administrative console page, click **Resources > JDBC Providers > New**.

Step 1: Select the database type:

Choose a supported database type.

If the list of supported database types does not include the type that you want to use, select **User-Defined**. You might need to consult the documentation for the database for more information on specific properties that database requires.

Data type Drop-down list

Step 2: Select the JDBC provider type:

Choose a supported JDBC Provider type.

Only JDBC provider types that are appropriate for the database type you selected in step 1 will appear in the list. For at-a-glance information on which provider type is appropriate for your database configuration and application requirements, see the JDBC provider table.

Data type Drop-down list

Step 3: Select the implementation type:

Choose a supported implementation type.

Only the implementation types supported by the JDBC provider you selected in step 2 will appear in the list.

Note: If two choices appear on the implementation type list, select **Connection Pool DataSource** if your application runs in a single phase or local transaction. Otherwise, choose **XA DataSource** to run in a global transaction.

Data type Drop-down list

JDBC provider summary:

Database type	JDBC Provider	Transaction support	Version and other considerations
DB2 on Windows, UNIX, or workstation-based LINUX	DB2 Universal JDBC Provider	One phase only	
	DB2 Universal JDBC Provider (XA)	One and two phase	
	DB2 legacy CLI-based Type 2 JDBC Provider	One phase only	
	DB2 legacy CLI-based Type 2 JDBC Provider (XA)	One and two phase	

Database type	JDBC Provider	Transaction support	Version and other considerations
DB2 UDB for iSeries	DB2 UDB for iSeries (Native)	One phase only	Recommended for use with WebSphere Application Server run on iSeries
	DB2 UDB for iSeries (Native XA)	One and two phase	Recommended for use with WebSphere Application Server run on iSeries
	DB2 UDB for iSeries (Toolbox)	One phase only	Recommended for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 UDB for iSeries (Toolbox XA)	One and two phase	Recommended for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 legacy CLI-based Type 2 JDBC Provider	One phase only	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect driver (available from DB2)
DB2 legacy CLI-based Type 2 JDBC Provider (XA)	One and two phase	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect driver (available from DB2)	

Database type	JDBC Provider	Transaction support	Version and other considerations
DB2 on z/OS	DB2 for z/OS Local JDBC Provider (RRS)	One and two phase	<i>Only</i> for use with WebSphere Application Server run on z/OS
	DB2 Universal JDBC Provider	One phase only	<i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 Universal JDBC Provider (XA)	One and two phase	<i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 legacy CLI-based Type 2 JDBC Provider	One phase only	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect program (available from DB2)
	DB2 legacy CLI-based Type 2 JDBC Provider (XA)	One and two phase	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect program (available from DB2)
Cloudscape	Cloudscape JDBC Provider	One phase only	- Not for use in clustering environment: Cloudscape is accessible from a single JVM only - Does not support Version 4 data sources
	Cloudscape JDBC Provider (XA)	One and two phase	- Not for use in clustering environment: Cloudscape is accessible from a single JVM only - Does not support Version 4 data sources
	Cloudscape Network Server using Universal JDBC driver	One phase only	Does not support Version 4 data sources
Informix	Informix JDBC Driver	One phase only	
	Informix JDBC Driver (XA)	One and two phase	
Sybase	Sybase JDBC Driver	One phase only	
	Sybase JDBC Driver (XA)	One and two phase	
Oracle	Oracle JDBC Driver	One phase only	
	Oracle JDBC Driver (XA)	One and two phase	

Database type	JDBC Provider	Transaction support	Version and other considerations
MS SQL Server	DataDirect ConnectJDBC type 4 driver for MS SQL Server	One phase only	Only for use with the corresponding driver from DataDirect Technologies
	DataDirect ConnectJDBC type 4 driver for MS SQL Server (XA)	One and two phase	Only for use with the corresponding driver from DataDirect Technologies
	WebSphere embedded ConnectJDBC driver for MS SQL Server	One phase only	- Not available for Application Server on z/OS - Cannot be used outside of WebSphere Application Server environment
	WebSphere embedded ConnectJDBC driver for MS SQL Server (XA)	One and two phase	- Not available for Application Server on z/OS - Cannot be used outside of WebSphere Application Server environment

Creating and configuring a data source using the administrative console:

After you create a JDBC provider, you must create a data source to access the backend data store. Application components use the data source to access connection instances to your database; a connection pool is associated with each data source. The product supports two different versions of data source:

- Version 4.0, for use with the Enterprise JavaBeans (EJB) 1.0 specification and the Java Servlet 2.2 specification
 - The latest standard version for use with more advanced releases of these specifications
1. Open the administrative console.
 2. Click **Resources > JDBC Providers**.
 3. Choose the JDBC resource provider under which you want to create your data source. The detail page for this provider is displayed.
 4. Under Additional Properties, click the **Data Sources** link that is appropriate for your application. The Data sources or Data sources (Version 4) page is displayed.
 5. Click **New** to display the Data source settings page.
 6. Verify that all the required properties have valid values.

For data sources of the latest standard version:

- a. Select a DataStoreHelper class name from the list entitled DataStoreHelpers provided by WebSphere Application Server, or leave the default selection as is. If you want to use a data store helper other than those available in the drop-down list, click **Specify a user-defined DataStoreHelper**. Type a fully qualified class name in the field that is provided.
- b. The next section of properties varies according to the database selection, provider type, and implementation that you chose for your JDBC provider. These properties are either required or highly recommended for your data source. Provide valid values for these settings if you do not want to accept the default values.
- c. Click **Component-managed Authentication Alias** if your database requires a user ID and password for a connection. This alias is used only when the application resource reference is using res-auth = Application.

Important:(For components with res-auth=Container) Both the Container-managed Authentication Alias and Mapping-Configuration Alias settings are deprecated. They are superseded by the

specification of a login configuration on the resource-reference mapping at deployment time. You must now use this login setting to define the aliases at deployment.

- d. If you chose XA Data Source as the implementation type of your JDBC provider, you need to specify the alias used during transaction recovery processing. An additional section entitled Authentication Alias for XA Recovery is available. Select either **Use Application Authentication Alias** to use the same value that you chose for component-managed authentication, or select **Specify**: to choose a different alias from the drop-down list.
7. Click **Apply** to view a page with your new data source settings. Additional properties and Related items sections are now available on this page. Additional properties contains the Connection pool, Custom properties, and WebSphere Application Server data source properties choices. (If you are using a Version 4 data source, however, you see only the first two choices.)
 - a. Click on the first link to define settings that affect the behavior of the Java 2 Connector (J2C) connection pool manager.
 - a. Go to the Custom properties page to view and modify additional properties that the database vendor might require for the connection of its product to an application server.
 - b. Use the WebSphere Application Server data source properties page to input settings that exclusively affect the WebSphere Application Server connection to the database.
 - c. The Related items section (applicable only to later version data sources, not Version 4 data sources) contains the J2C Authentication data entries choice. Here, you can specify a list of user IDs and passwords for J2C security to use.
8. Click **Save**.
9. Return to the data source page to confirm that your new data source is displayed in the list.

You are now ready to install the application for which you configured the data sources. During installation, you can bind resource references to these data sources.

Data source collection:

Use this page to create or modify a data source.

To view this administrative console page, click **Resources > JDBC Providers > JDBC_provider > Data sources**.

Note: If you are using the Enterprise JavaBean (EJB) 1.0 specification and the Java Servlet 2.2 specification, you must use the **Data sources (Version 4)** console page.

Name:

Specifies the display name of this data source.

Data type String

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name for this data source.

Data type String

Description:

Specifies a text description of the data source.

Data type String

Category:

Specifies a string that you can use to classify or group a data source.

Data type String

Data source settings:

Use this page to create a data source for association with your JDBC provider. Think of the data source as a pooled set of connections necessary for conducting transactions between your application and database.

To view this administrative console page, click **Resources** > **JDBC Providers** > *JDBC_provider* > **Data sources** > **New** (if you are creating a new data source) or > *data_source* (if you are viewing an established data source).

Note: If your application uses an Enterprise JavaBean (EJB) 1.1 or a Java Servlet 2.2 module, you must use the **Data sources (Version 4)** > *data_source* console page.

Name:

Specifies the display name for the data source.

Valid characters for this name include letters and numbers, but NOT most of the special characters. For example you can set this field to *Test Data Source*. But any name starting with a period (•) or containing special characters (\ / , ; " * ? < > | = + & % ' `) is not a valid name.

Data type String

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name.

Distributed computing environments often employ naming and directory services to obtain shared components and resources. Naming and directory services associate names with locations, services, information, and resources.

Naming services provide name-to-object mappings. Directory services provide information on objects and the search tools required to locate those objects.

There are many naming and directory service implementations, and the interfaces to them vary. JNDI provides a common interface that is used to access the various naming and directory services.

For example, you can use the name *jdbc/markSection*.

If you leave this field blank a JNDI name is generated from the name of the data source. For example, a data source name of *markSection* generates a JNDI name of *jdbc/markSection*.

After you set this value, save it, and restart the server, you can see this string when you run the dump name space tool.

Data type String

Container-managed persistence:

Specifies if this data source is used for container-managed persistence of enterprise beans.

If this field is checked, a CMP Connector Factory that corresponds to this data source is created for the relational resource adapter.

Data type	Checkbox
Default	Enabled (The field is checked.)

Description:

Specifies a text description for the resource.

Data type	String
------------------	--------

Category:

Specifies a category string you can use to classify or group the resource.

Data type	String
------------------	--------

Data store helper class name:

Specifies the name of the DataStoreHelper implementation class that extends the capabilities of your selected JDBC driver implementation class to perform database-specific functions.

WebSphere Application Server provides a set of DataStoreHelper implementation classes for each of the JDBC provider drivers it supports. These implementation classes are in the package `com.ibm.websphere.rsadapter`. For example, if your JDBC provider is DB2, then your default DataStoreHelper class is `com.ibm.websphere.rsadapter.DB2DataStoreHelper`. The administrative console page you are viewing, however, might make multiple DataStoreHelper class names available to you in a drop-down list; be sure to select the one required by your database configuration. Otherwise, your application might not work correctly. If you want to use a DataStoreHelper other than those displayed in the drop-down list, select **Specify a user-defined DataStoreHelper** and type a fully qualified class name. Refer to the Information Center topic "Example: Developing your own DataStoreHelper class."

Data type	Drop-down list or string (if user-defined DataStoreHelper is selected)
------------------	---

Important data source properties: These properties are specific to the data source that corresponds to your selected JDBC provider. They are either required by the data source, or are especially useful for the data source. You can find a complete list of the properties required for all supported JDBC providers in the topic "Vendor-specific data sources minimum required settings" in the Information Center.

Component-managed Authentication Alias:

This alias is used for database authentication at run time.

The **Component-managed Authentication Alias** is only used when the application resource reference is using `res-auth = Application`.

If your database (for example, Cloudscape) does not support *user ID* and *password*, then do not set the alias in the component-managed authentication alias or container-managed authentication alias fields. Otherwise, you see the warning message in the system log to indicate that the user and password are not valid properties. (This message is only a warning message; the data source is still created successfully.)

If you do not set an alias (component-managed or otherwise), and your database requires the user ID and password to get a connection, then you receive an exception during run time.

Data type Drop-down list

Container-managed Authentication Alias (deprecated):

Specifies authentication data (a string path converted to userid and password) for container-managed sign-on to the resource.

Note: Beginning with WebSphere Application Server Version 6.0, the container-managed authentication alias is superseded by the specification of a login configuration on the resource-reference mapping at deployment time, for components with *res-auth=Container*.

Choose from aliases defined under **Security>JAAS Configuration> J2C Authentication Data**.

To define a new alias not already appearing in the pick list:

- Click **Apply** to expose Related Items.
- Click **J2C Authentication Data Entries**.
- Define an alias.
- Click the connection factory name at the top of the *J2C Authentication Data Entries* page to return to the connection factory page.
- Select the alias.

Data type Drop-down list

Mapping-Configuration Alias (deprecated):

Allows users to select from the **Security > JAAS Configuration > Application Logins Configuration** list.

Note: Beginning with WebSphere Application Server Version 6.0, the Mapping-Configuration Alias is superseded by the specification of a login configuration on the resource-reference mapping at deployment time, for components with *res-auth=Container*.

The **DefaultPrincipalMapping** JAAS configuration maps the authentication alias to the userid and password. You may define and use other mapping configurations.

Data type Drop-down list

Authentication Alias for XA Recovery:

This optional field is used to specify the authentication alias that should be used during XA recovery processing.

If the resource adapter does not support XA transactions, then this field will not be displayed. The default value will come from the selected alias for application authentication (if specified).

Use Component-managed Authentication Alias

Selecting this radio button specifies that the alias set for Component-managed Authentication is used at XA recovery time.

Data type Radio button

Specify:

Selecting this radio button enables you to choose an authentication alias from a drop-down list of

configured aliases.

Data type

Radio button

WebSphere Application Server data source properties collection:

Use this page to view the WebSphere Application Server data source properties. These properties apply to the WebSphere Application Server connection, rather than to the database connection.

To view this administrative console page, click **Resources > JDBC Providers > JDBC_provider > Data sources > data_source > WebSphere Application Server connection properties.**

Statement Cache Size:

Specifies the number of free statements that are cached per connection.

The WebSphere Application Server data source optimizes the processing of prepared statements. A prepared statement is a precompiled SQL statement that is stored in a prepared statement object. This object is used to efficiently execute the given SQL statement multiple times.

If the cache is not large enough, useful entries are discarded to make room for new entries. To determine the largest value for your cache size to avoid any cache discards, add the number of uniquely prepared statements and callable statements (as determined by the *sql* string, concurrency, and the scroll type) for each application that uses this data source on a particular server. This value is the maximum number of possible prepared statements that are cached on a given connection over the life of the server. Setting the cache size to this value means you never have cache discards. In general, the more statements your application has, the larger the cache should be. For example, if the application has 5 SQL statements, set the statement cache size to 5, so that each connection has 5 statements.

You can also use the Tivoli Performance Viewer to minimize cache discards. Use a standard workload that represents a typical number of incoming client requests, use a fixed number of iterations, and use a standard set of configuration settings. **Note:** The higher the statement cache, the more system resources are delayed. Therefore, if you set the number too high, you could lack resources because your system is not able to open that many prepared statements.

In test applications, tuning the statement cache improved throughput by 10-20%. However, because of potential resource limitations, this might not always be possible.

Data type

Integer

Default

Depends on the database. Most are 10. Informix Version 7.3, 9.2, or 9.3 without latest fix must be 0. A default of 0 means there is no cache statement.

Enable Multithreaded Access Detection: If checked, the application server detects the existence of access by multiple threads.

Enable WebSphere Connection Pooling: If checked, the application server sets up connect pools for this datasource.

Enable Database Reauthentication: If checked, there is not be an exact match on connections retrieved out of the WebSphere Application Server connection pool (that is, connection pool search criteria do not include user name and password). Instead, the reauthentication of connection is done in the *doConnectionSetupPerTransaction()* of the DataStoreHelper class. Note that WebSphere Application Server runtime does NOT provide connection reauthentication implementation. Therefore, when this box is checked you MUST extend the DataStoreHelper class to provide implementation of the *doConnectionSetupPerTransaction()* method where the reauthentication takes place. Failure to do that

results in wrong connections being handed out to users. For more information, refer to the Javadoc API documentation for *com.ibm.websphere.rsadapter.DataStoreHelper#doConnectionSetupPerTransaction(...)*.

Connection reauthentication can help improve performance by reducing the overhead of opening and closing connections, particularly for applications that always request connections with different user names and passwords.

Enable JMS One Phase Optimization Support: If checked, the application server allows JMS to get optimized connections from this data source. This property prevents JDBC applications from sharing connections with CMP applications.

PreTest Connections: If checked, the application server tries to connect to this data source before it attempts to send data to or receive data from this data source. If you select this property, you can specify how often, in seconds, the application server retries to make a connection if the initial attempt fails.

PreTest Connection Retry Interval: When **PreTest Connection** is checked, use this property to specify how long, in seconds, the application server waits before retrying to make a connection if the initial attempt fails.

PreTest SQL String:

Specifies the string of data that the application server sends to the database to test the connection.

Data type Integer

Data sources (Version 4):

Use this page to view the settings of a Version 4.0 style data source.

These Version 4.0 data sources use the WebSphere Application Server Version 4.0 Connection Manager architecture. All EJB 1.1 modules must use a Version 4.0 data source.

To view this administrative console page, click **Resources** > **JDBC Providers** > *JDBC_provider* > **Data sources (Version 4)**.

Name:

Specifies a text identifier of the data source.

Data type String

JNDI Name:

Specifies the Java Naming and Directory Interface (JNDI) name of the data source.

Data type String

Description:

Specifies a text description of the data source.

Data type String

Category:

Specifies a text string that you can use to classify or group the data source.

Data type String

Data source (Version 4) settings:

Use this page to create a Version 4.0 style data source. This data source uses the WebSphere Application Server Version 4.0 connection manager architecture. All of your EJB1.x modules must use this data source.

To view this administrative console page, click **Resources > JDBC Providers > JDBC_provider > Data Sources (Version 4) > data_source**.

Scope:

Specifies the level to which this resource definition is visible -- the cell, node, or server level.

Resources such as JDBC providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

Note that no matter what the scope of a defined resource, the resource's properties only apply at an individual server level. For example, if you define the scope of a data source at the cell level, all users in that cell can look up and use that data source, which is unique within that cell. However, resource property settings are local to each server in the cell. For example, if you define *max connections* to 10, then each server in that cell can have 10 connections.

Cell The most general scope. Resources defined at the cell scope are visible from all nodes and servers, unless they are overridden. To view resources defined in the cell scope, do not specify a server or a node name in the scope selection form.

Node The default scope for most resource types. Resources defined at the node scope override any duplicates defined at the cell scope and are visible to all servers on the same node, unless they are overridden at a server scope on that node. To view resources defined in a node scope, do not specify a server, but select a node name in the scope selection form.

Server The most specific scope for defining resources. Resources defined at the server scope override any duplicate resource definitions defined at the cell scope or parent node scope and are visible only to a specific server. To view resources defined in a server scope, specify a server name as well as a node name in the scope selection form.

When resources are created, they are always created into the current scope selected in the panel. To view resources in other scopes, specify a different node or server in the scope selection form.

Data type String

Name:

Specifies the display name for the resource.

For example, you can set this field to *Test Data Source*.

Data type String

JNDI Name:

Specifies the Java Naming and Directory Interface (JNDI) name.

Distributed computing environments often employ naming and directory services to obtain shared components and resources. Naming and directory services associate names with locations, services, information, and resources.

Naming services provide name-to-object mappings. Directory services provide information on objects and the search tools required to locate those objects.

There are many naming and directory service implementations, and the interfaces to them vary. JNDI provides a common interface that is used to access the various naming and directory services.

For example, you can use the name *jdbc/markSection*.

If you leave this field blank a JNDI name is generated from the name of the data source. For example, a data source name of *markSection* generates a JNDI name of *jdbc/markSection*.

After you set this value, save it, and restart the server, you can see this string when you run the dump name space tool.

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

Database Name:

Specifies the name of the database that this data source accesses.

For example, you can call the database *SAMPLE*.

Data type String

Default User ID:

Specifies the user name to use for connecting to the database.

For example, you can use the ID *db2admin*.

Data type String

Default Password:

Specifies the password used for connecting to the database.

For example, you can use the password *db2admin*.

Data type String

Custom properties collection:

Use this page to view and configure the custom properties of a J2EE resource provider.

You can configure custom property collections for numerous resource types. According to the resource type with which a collection is associated, your ability to add, delete, and modify individual properties and settings varies. Begin the configuration process by clicking on the *Required* field to sort those column values in descending order. All of the required (true) values are then sorted at the beginning of the page. Be sure to set all required properties.

Name:

Specifies the property name.

You must ensure that the resource provider has the setting for this name.

Data type String

Value:

Specifies the property value.

Data type Variable; see “Custom property settings” for more information.

Description:

Specifies text to describe any bounds or well-defined values for this property.

Data type String

Required:

Specifies whether this property is required for the resource provider.

Data type Boolean or Check box

Custom property settings:

Use this page to view and set custom properties that might be required for resource providers and resource factories.

According to the resource type with which a property collection is associated, your ability to modify individual property settings varies. Therefore, consider the following descriptions as a general reference for custom property settings. (The administrative console page that you are using to configure your custom property may only allow you to modify a subset of the following settings.)

Required:

Specifies properties that are required for this resource.

Data type Check box

Name:

Specifies the name associated with this property (PortNumber, ConnectionURL, etc).

Data type String

Value:

Specifies the value associated with this property in this property set.

Data type Determined by the **Type** setting, which you select from a drop-down list. If the type is `java.lang.String` then the value is of type String; if the type is `java.lang.Integer`, then the value is of type Integer; and so on.

Description:

Specifies text to describe any bounds or well-defined values for this property.

Data type String

Type:

Specifies the fully qualified Java data type of this property .

There are specific types that are valid:

- `java.lang.Boolean`
- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Double`
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Character`

Data type Drop-down list

Custom Properties (Version 4) collection:

Use this page to view properties for a Version 4.0 data source.

To view this administrative console page, click **Resources** > **JDBC Providers** > *JDBC_provider* > **Data Sources (Version 4)** > *data_source* > **Custom Properties**

Name:

Specifies the name of the custom property

Data type String

Value:

Specifies the value of the custom property.

Data type Integer

Description:

Specifies text to describe any bounds or well-defined values for this property.

Data type String

Required:

Specifies properties that are required for this resource.

Data type String

Custom property (Version 4) settings:

Use this page to add properties for a Version 4.0 data source.

To view this administrative console page, click **Resources > JDBC Providers > JDBC_provider > Data Sources (Version 4) > data_source > Custom Properties > custom_property**.

Scope:

Specifies the level to which this resource definition is visible -- the cell, node, or server level.

Resources such as JDBC providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

Note that no matter what the scope of a defined resource, the resource's properties only apply at an individual server level. For example, if you define the scope of a data source at the cell level, all users in that cell can look up and use that data source, which is unique within that cell. However, resource property settings are local to each server in the cell. For example, if you define *max connections* to 10, then each server in that cell can have 10 connections.

Cell The most general scope. Resources defined at the cell scope are visible from all nodes and servers, unless they are overridden. To view resources defined in the cell scope, do not specify a server or a node name in the scope selection form.

Node The default scope for most resource types. Resources defined at the node scope override any duplicates defined at the cell scope and are visible to all servers on the same node, unless they are overridden at a server scope on that node. To view resources defined in a node scope, do not specify a server, but select a node name in the scope selection form.

Server The most specific scope for defining resources. Resources defined at the server scope override any duplicate resource definitions defined at the cell scope or parent node scope and are visible only to a specific server. To view resources defined in a server scope, specify a server name as well as a node name in the scope selection form.

When resources are created, they are always created into the current scope selected in the panel. To view resources in other scopes, specify a different node or server in the scope selection form.

Data type String

Required:

Specifies properties that are required for this resource.

Data type Check box

Name:

Specifies the name associated with this property (PortNumber, ConnectionURL, etc).

Data type String

Value:

Specifies the value associated with this property in this property set.

Data type Integer

Description:

Specifies text to describe any bounds or well-defined values for this property.

Data type String

Type:

Specifies the fully qualified Java type of this property (java.lang.Integer, java.lang.Byte).

Data type String

Creating and configuring a JDBC provider and data source using the Java Management Extensions API:

If your application requires access to a JDBC connection pool from a J2EE 1.3 or 1.4 level WebSphere Application Server component, you can create the necessary JDBC provider and data source objects using the Java Management Extensions (JMX) API exclusively. Alternatively, you can use the JMX API in combination with the WSadmin - scripting tool.

Note: Use the JMX API to create only data sources for which the product does *not* provide a template. For every JDBC provider WebSphere Application Server supports, the product provides a corresponding data source template. You can create supported providers and associated data sources through the administrative console, or by using the WSadmin - scripting tool. For a complete list of supported JDBC providers (and therefore a complete list of data sources that must be created using a template), refer to the topic “Vendor-specific data sources minimum required settings” on page 580.

These steps outline the general procedure for using the JMX API to create a JDBC provider and data source, on WebSphere Application Server running on Windows platforms:

1. Set your classpath.

You need two JAR files in your classpath -- *wsexception.jar* and *wasjmx.jar*. The following command is an example for setting your classpath (shown on 2 lines for publication):

```
set classpath=%classpath%;D:\WebSphere\AppServer\lib\wsexception.jar;  
D:\WebSphere\AppServer\lib\wasjmx.jar
```

2. Look up the host and get an administration client handle.
3. Get a configuration service handle.
4. Update the *resource.xml* file using the configuration service as desired.
 - a. Add a JDBC provider.
 - b. Add the data source.
 - c. Add the connection factory. (This step is necessary only for data sources that must support container-managed persistence.)
5. Reload the *resource.xml* file to bind the newly created data source into the JNDI namespace. Perform this step if you want to use the newly created data source right away without restarting the application server.
 - a. Locate the DataSourceConfigHelper MBean using the name.
 - b. Put together the signature and parameters for the call.
 - c. Invoke the reload() call.

Example: Using the Java Management Extensions API to create a JDBC driver and data source for container-managed persistence:

```
//  
// "This program may be used, executed, copied, modified and distributed without royalty for the  
// purpose of developing, using, marketing, or distributing."  
//  
// Product 5630-A36, (C) COPYRIGHT International Business Machines Corp., 2001, 2002  
// All Rights Reserved * Licensed Materials - Property of IBM  
//  
import java.util.*;  
import javax.sql.*;  
import javax.transaction.*;  
import javax.management.*;  
  
import com.ibm.websphere.management.*;  
import com.ibm.websphere.management.configservice.*;  
import com.ibm.ws.exception.WsException;  
  
/**  
 * Creates a node scoped resource.xml entry for a DB2 XA datasource.  
 * The datasource created is for CMP use.  
 *  
 * We need following to run (shown on multiple lines for publication):  
 * set classpath=%classpath%;D:\WebSphere\AppServer\lib\wsexception.jar;  
 * D:\WebSphere\AppServer\lib\wasjmx.jar;D:\$WAS_HOME\lib\wasx.jar  
 */  
public class CreateDataSourceCMP {  
  
    String dsName = "markSection"; // ds display name , also jndi name and CF name  
    String dbName = "SECTION"; // database name  
    String authDataAlias = "db2admin"; // an authentication data alias  
    String uid = "db2admin"; // userid  
    String pw = "db2admin"; // password  
    String dbclasspath = "D:/SQLLIB/java/db2java.zip"; // path to the db driver  
  
    /**  
     * Main method.  
     */  
    public static void main(String[] args) {  
        CreateDataSourceCMP cds = new CreateDataSourceCMP();  
    }  
}
```

```

try {
    cds.run(args);
} catch (com.ibm.ws.exception.WsException ex) {
    System.out.println("Caught this " + ex );
    ex.printStackTrace();
    //ex.getCause().printStackTrace();
} catch (Exception ex) {
    System.out.println("Caught this " + ex );
    ex.printStackTrace();
}
}

/**
 * This method creates the datasource using JMX.
 * The datasource created here is only written into resources.xml.
 * It is not bound into namespace until the server is restarted, or an application started
 */
public void run(String[] args) throws Exception {

    try {
        // Initialize the AdminClient.
        Properties adminProps = new Properties();
        adminProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
        adminProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
        adminProps.setProperty(AdminClient.CONNECTOR_PORT, "8880");
        AdminClient adminClient = AdminClientFactory.createAdminClient(adminProps);

        // Get the ConfigService implementation.
        com.ibm.websphere.management.configservice.ConfigServiceProxy configService =
            new com.ibm.websphere.management.configservice.ConfigServiceProxy(adminClient);

        Session session = new Session();

        // Use this group to add to the node scoped resource.xml.
        ObjectName node1 = ConfigServiceHelper.createObjectName(null, "Node", null);
        ObjectName[] matches = configService.queryConfigObjects(session, null, node1, null);
        node1 = matches[0]; // use the first node found

        // Use this group to add to the server1 scoped resource.xml.
        ObjectName server1 = ConfigServiceHelper.createObjectName(null, "Server", "server1");
        matches = configService.queryConfigObjects(session, null, server1, null);
        server1 = matches[0]; // use the first server found

        // Create the JDBCProvider
        String providerName = "DB2 JDBC Provider (XA)";
        System.out.println("Creating JDBCProvider " + providerName );

        // Prepare the attribute list
        AttributeList provAttrs = new AttributeList();
        provAttrs.add(new Attribute("name", providerName));
        provAttrs.add(new Attribute("implementationClassName",
            "COM.ibm.db2.jdbc.DB2XADataSource"));
        provAttrs.add(new Attribute("description","DB2 JDBC2-compliant XA Driver"));

        //create it
        ObjectName jdbcProv =
            configService.createConfigData(session,node1,"JDBCProvider",
                "resources.jdbc:JDBCProvider",provAttrs);

        // now plug in the classpath
        configService.addElement(session,jdbcProv,"classpath",dbcclasspath,-1);

        // Search for RRA so we can link it to the datasource
        ObjectName rra =
            ConfigServiceHelper.createObjectName(null, "J2CResourceAdapter", null);

```

```

matches = configService.queryConfigObjects(session, node1, rra, null);
rra = matches[0]; // use the first J2CResourceAdapter segment for builtin_rra

// Prepare the attribute list
AttributeList dsAttrs = new AttributeList();
dsAttrs.add(new Attribute("name", dsName));
dsAttrs.add(new Attribute("jndiName", "jdbc/" + dsName));
dsAttrs.add(new Attribute("datasourceHelperClassname",
    "com.ibm.websphere.rsadapter.DB2DataStoreHelper"));
dsAttrs.add(new Attribute("statementCacheSize", new Integer(10)));
dsAttrs.add(new Attribute("relationalResourceAdapter",
    rra)); // this is where we make the link to "builtin_rra"
dsAttrs.add(new Attribute("description", "JDBC Datasource for mark section CMP 2.0 test"));
dsAttrs.add(new Attribute("authDataAlias",authDataAlias));

// Create the datasource
System.out.println(" ** Creating datasource");
ObjectName dataSource =
    configService.createConfigData(session,jdbcProv,"DataSource",
    "resources.jdbc:DataSource",dsAttrs);

// Add a propertySet.
AttributeList propSetAttrs = new AttributeList();
ObjectName resourcePropertySet =
    configService.createConfigData(session,dataSource,"propertySet","",propSetAttrs);

// Add resourceProperty databaseName
AttributeList propAttrs1 = new AttributeList();
propAttrs1.add(new Attribute("name", "databaseName"));
propAttrs1.add(new Attribute("type", "java.lang.String"));
propAttrs1.add(new Attribute("value", dbName));

configService.addElement(session,resourcePropertySet,"resourceProperties",propAttrs1,-1);

// Now Create the corresponding J2CResourceAdapter Connection Factory object.
ObjectName jra = ConfigServiceHelper.createObjectName(null,"J2CResourceAdapter",null);

// Get all the J2CResourceAdapter, and I want to add my datasource
System.out.println(" ** Get all J2CResourceAdapter's");
ObjectName[] jras = configService.queryConfigObjects(session, node1, jra, null);

int i=0;

for (;i<jras.length;i++) {
    System.out.println(ConfigServiceHelper.getConfigDataType(jras[i])+ " " + i + " = "
        + jras[i].getKeyProperty(SystemAttributes._WEBSHERE_CONFIG_DATA_DISPLAY_NAME)
        + "\nFrom scope ="
        + jras[i].getKeyProperty(SystemAttributes._WEBSHERE_CONFIG_DATA_ID));
    // quit on the first builtin_rra
    if (jras[i].getKeyProperty(SystemAttributes._WEBSHERE_CONFIG_DATA_DISPLAY_NAME)
        .equals("WebSphere Relational Resource Adapter")) {
        break;
    }
}

if (i >= jras.length) {
    System.out.println(
        "Did not find builtin_rra J2CResourceAdapter object creating CF anyways" );
} else {
    System.out.println(
        "Found builtin_rra J2CResourceAdapter object at index " + i + " creating CF" );
}

// Prepare the attribute list
AttributeList cfAttrs = new AttributeList();
cfAttrs.add(new Attribute("name", dsName + "_CF"));

```

```

cfAttrs.add(new Attribute("authMechanismPreference","BASIC_PASSWORD"));
cfAttrs.add(new Attribute("authDataAlias",authDataAlias));
cfAttrs.add(new Attribute("cmpDatasource",
    dataSource )); // this is where we make the link to DataSource's xmi:id
ObjectName cf =
    configService.createConfigData(session,jras[i],"CMPConnectorFactory",
        "resources.jdbc:CMPCConnectorFactory",cfAttrs);

// ===== start Security section
System.out.println("Creating an authorization data alias " + authDataAlias);

// Find the parent security object
ObjectName security = ConfigServiceHelper.createObjectName(null, "Security", null);
ObjectName[] securityName =
    configService.queryConfigObjects(session, null, security, null);
security=securityName[0];

// Prepare the attribute list
AttributeList authDataAttrs = new AttributeList();
authDataAttrs.add(new Attribute("alias", authDataAlias));
authDataAttrs.add(new Attribute("userId", uid));
authDataAttrs.add(new Attribute("password", pw));
authDataAttrs.add(new Attribute("description","Auto created alias for datasource"));

//create it
ObjectName authDataEntry =
    configService.createConfigData(session,security,"authDataEntries",
        "JAASAuthData",authDataAttrs);

// ===== end Security section

// Save the session
System.out.println("Saving session" );
configService.save(session, false);

// reload resources.xml to bind the new datasource into the name space
reload(adminClient,true);
} catch (Exception ex) {
    ex.printStackTrace(System.out);
    throw ex;
}
}

/**
 * Get the DataSourceConfigHelperMbean and call reload() on it
 *
 * @param adminClient
 * @param verbose true - print messages to stdout
 */
public void reload(AdminClient adminClient,boolean verbose) {
    if (verbose) {
        System.out.println("Finding the Mbean to call reload()");
    }
}

// First get the Mbean
ObjectName handle = null;
try {
    ObjectName queryName = new ObjectName("WebSphere:type=DataSourceCfgHelper,*");
    Set s = adminClient.queryNames(queryName, null);
    Iterator iter = s.iterator();
    if (iter.hasNext()) handle = (ObjectName)iter.next();
} catch (MalformedObjectNameException mone) {
    System.out.println("Check the program variable queryName" + mone);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
    System.out.println("Cannot connect to the application server" + ce);
}
}

```

```

    if (verbose) {
        System.out.println("Calling reload()");
    }
    Object result = null;
    try {
        result = adminClient.invoke(handle, "reload", new Object[] {}, new String[] {});
    } catch (MBeanException mbe) {
        if (verbose) {
            System.out.println("\tMbean Exception calling reload" + mbe);
        }
    } catch (InstanceNotFoundException infe) {
        System.out.println("Cannot find reload ");
    } catch (Exception ex) {
        System.out.println("Exception occurred calling reload()" + ex);
    }
    if (result==null && verbose) {
        System.out.println("OK reload()");
    }
}
}
}

```

Example: Using the Java Management Extensions API to create a JDBC driver and data source for bean-managed persistence, session beans, or servlets:

```

//
// "This program may be used, executed, copied, modified and distributed without royalty for the
// purpose of developing, using, marketing, or distributing."
//
// Product 5630-A36, (C) COPYRIGHT International Business Machines Corp., 2001, 2002
// All Rights Reserved * Licensed Materials - Property of IBM
//
import java.util.*;
import javax.sql.*;
import javax.transaction.*;
import javax.management.*;

import com.ibm.websphere.management.*;
import com.ibm.websphere.management.configservice.*;
import com.ibm.ws.exception.WsException;

/**
 * Creates a node scoped resource.xml entry for a DB2 XA datasource.
 * The datasource created is for BMP use.
 *
 * We need following to run (shown on 2 lines for publication):
 * set classpath=%classpath%;D:\WebSphere\AppServer\lib\wsexception.jar;
 *   D:\WebSphere\AppServer\lib\wasjmx.jar;D:\$WAS_HOME\lib\wasx.jar
 */
public class CreateDataSourceBMP {

    String dsName = "markSection"; // ds display name , also jndi name and CF name
    String dbName = "SECTION"; // database name
    String authDataAlias = "db2admin"; // an authentication data alias
    String uid = "db2admin"; // userid
    String pw = "db2admin"; // password
    String dbclasspath = "D:/SQLLIB/java/db2java.zip"; // path to the db driver

    /**
     * Main method.
     */
    public static void main(String[] args) {

```

```

CreateDataSourceBMP cds = new CreateDataSourceBMP();

try {
    cds.run(args);
} catch (com.ibm.ws.exception.WsException ex) {
    System.out.println("Caught this " + ex );
    ex.printStackTrace();
    //ex.getCause().printStackTrace();
} catch (Exception ex) {
    System.out.println("Caught this " + ex );
    ex.printStackTrace();
}
}

/**
 * This method creates the datasource using JMX.
 *
 * The datasource created here is only written into resources.xml.
 * It is not bound into namespace until the server is restarted, or an application started
 */
public void run(String[] args) throws Exception {

    try {
        // Initialize the AdminClient.
        Properties adminProps = new Properties();
        adminProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
        adminProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
        adminProps.setProperty(AdminClient.CONNECTOR_PORT, "8880");
        AdminClient adminClient = AdminClientFactory.createAdminClient(adminProps);

        // Get the ConfigService implementation.
        com.ibm.websphere.management.configservice.ConfigServiceProxy configService =
            new com.ibm.websphere.management.configservice.ConfigServiceProxy(adminClient);

        Session session = new Session();

        // Use this group to add to the node scoped resource.xml.
        ObjectName node1 = ConfigServiceHelper.createObjectName(null, "Node", null);
        ObjectName[] matches = configService.queryConfigObjects(session, null, node1, null);
        node1 = matches[0]; // use the first node found

        // Use this group to add to the server1 scoped resource.xml.
        ObjectName server1 = ConfigServiceHelper.createObjectName(null, "Server", "server1");
        matches = configService.queryConfigObjects(session, null, server1, null);
        server1 = matches[0]; // use the first server found

        // Create the JDBCProvider
        String providerName = "DB2 JDBC Provider (XA)";
        System.out.println("Creating JDBCProvider " + providerName );

        // Prepare the attribute list
        AttributeList provAttrs = new AttributeList();
        provAttrs.add(new Attribute("name", providerName));
        provAttrs.add(
            new Attribute("implementationClassName", "COM.ibm.db2.jdbc.DB2XADatasource"));
        provAttrs.add(new Attribute("description", "DB2 JDBC2-compliant XA Driver"));

        //create it
        ObjectName jdbcProv =
            configService.createConfigData(session, node1, "JDBCProvider",
                "resources.jdbc:JDBCProvider", provAttrs);
        // now plug in the classpath
        configService.addElement(session, jdbcProv, "classpath", dbclasspath, -1);
    }
}

```

```

// Search for RRA so we can link it to the datasource
ObjectName rra = ConfigServiceHelper.createObjectName(null, "J2CResourceAdapter", null);
matches = configService.queryConfigObjects(session, node1, rra, null);
rra = matches[0]; // use the first J2CResourceAdapter segment for builtin_rra

// Prepare the attribute list
AttributeList dsAttrs = new AttributeList();
dsAttrs.add(new Attribute("name", dsName));
dsAttrs.add(new Attribute("jndiName", "jdbc/" + dsName));
dsAttrs.add(new Attribute("datasourceHelperClassname",
    "com.ibm.websphere.rsadapter.DB2DataStoreHelper"));
dsAttrs.add(new Attribute("statementCacheSize", new Integer(10)));
dsAttrs.add(new Attribute(
    "relationalResourceAdapter", rra)); // this is where we make the link to "builtin_rra"
dsAttrs.add(new Attribute("description", "JDBC Datasource for mark section CMP 2.0 test"));
dsAttrs.add(new Attribute("authDataAlias",authDataAlias));

// Create the datasource
System.out.println(" ** Creating datasource");
ObjectName dataSource =
    configService.createConfigData(session,jdbcProv,"DataSource",
    "resources.jdbc:DataSource",dsAttrs);

// Add a propertySet.
AttributeList propSetAttrs = new AttributeList();
ObjectName resourcePropertySet =
    configService.createConfigData(session,dataSource,"propertySet","",propSetAttrs);

// Add resourceProperty databaseName
AttributeList propAttrs1 = new AttributeList();
propAttrs1.add(new Attribute("name", "databaseName"));
propAttrs1.add(new Attribute("type", "java.lang.String"));
propAttrs1.add(new Attribute("value", dbName));

configService.addElement(session,resourcePropertySet,"resourceProperties",propAttrs1,-1);

// ===== start Security section
System.out.println("Creating an authorization data alias " + authDataAlias);

// Find the parent security object
ObjectName security = ConfigServiceHelper.createObjectName(null, "Security", null);
ObjectName[] securityName =
    configService.queryConfigObjects(session, null, security, null);
security=securityName[0];

// Prepare the attribute list
AttributeList authDataAttrs = new AttributeList();
authDataAttrs.add(new Attribute("alias", authDataAlias));
authDataAttrs.add(new Attribute("userId", uid));
authDataAttrs.add(new Attribute("password", pw));
authDataAttrs.add(new Attribute("description","Auto created alias for datasource"));

//create it
ObjectName authDataEntry =
    configService.createConfigData(session,security,"authDataEntries",
    "JAASAuthData",authDataAttrs);
// ===== end Security section

// Save the session
System.out.println("Saving session" );
configService.save(session, false);

// reload resources.xml
reload(adminClient,true);

```



```

    } catch (Exception ex) {
        ex.printStackTrace(System.out);
        throw ex;
    }
}

/**
 * Get the DataSourceConfigHelperMbean and call reload() on it
 *
 * @param adminClient
 * @param verbose true - print messages to stdout
 */
public void reload(AdminClient adminClient,boolean verbose) {
    if (verbose) {
        System.out.println("Finding the Mbean to call reload()");
    }

    // First get the Mbean
    ObjectName handle = null;
    try {
        ObjectName queryName = new ObjectName("WebSphere:type=DataSourceCfgHelper,*");
        Set s = adminClient.queryNames(queryName, null);
        Iterator iter = s.iterator();
        if (iter.hasNext()) handle = (ObjectName)iter.next();
    } catch (MalformedObjectNameException mone) {
        System.out.println("Check the program variable queryName" + mone);
    } catch (com.ibm.websphere.management.exception.ConnectorException ce) {
        System.out.println("Cannot connect to the application server" + ce);
    }

    if (verbose) {
        System.out.println("Calling reload()");
    }
    Object result = null;
    try {
        result = adminClient.invoke(handle, "reload", new Object[] {}, new String[] {});
    } catch (MBeanException mbe) {
        if (verbose) {
            System.out.println("\tMbean Exception calling reload" + mbe);
        }
    } catch (InstanceNotFoundException infe) {
        System.out.println("Cannot find reload ");
    } catch (Exception ex) {
        System.out.println("Exception occurred calling reload()" + ex);
    }

    if (result==null && verbose) {
        System.out.println("OK reload()");
    }
}
}

```

Example: Creating a JDBC provider and data source using Java Management Extensions API and the scripting tool: The following code is a JACL (WSadmin - scripting tool) script used to create a data source.

Note: Use this script to create only data sources for which the product does *not* provide a template. For every JDBC provider WebSphere Application Server supports, the product provides a corresponding data source template. See the topic "Creating configuration objects using the wsadmin tool" in the information center for instructions on how to use the `createUsingTemplate` command to establish these data sources. For a complete list of supported JDBC providers (and therefore a complete list of data sources that must be created using a template), refer to the topic "Vendor-specific data sources minimum required settings" on page 580.

This script:

- Creates a data source *fvtDS_1*
- Creates a 4.0 data source *fvtDS_3*
- Creates a container-managed persistence (CMP) data source linked to *fvtDS_1*

```
#AWE -- Set up XA DB2 data sources, both Version 4.0 and J2EE Connector
        architecture (JCA)-compliant data sources
```

```
#UPDATE THESE VALUES:
```

```
#The classpath that will be used by your database driver
set driverClassPath "c:/sqllib/java/db2java.zip"
```

```
set server "server1"
```

```
set fvtbase "c:/wssb/fvtbase"
```

```
#Users and passwords..
```

```
set defaultUser1 "dbuser1"
set defaultPassword1 "dbpwd1"
set aliasName "alias1"
```

```
set databaseName1 "jtest1"
set databaseName2 "jtest2"
```

```
#END OF UPDATES
```

```
puts "Add an alias alias1"
set cell [$AdminControl getCell]
set sec [$AdminConfig getid /Cell:$cell/Security:/]
```

```
#-----
# Create a JAASAuthData object for component-managed authentication
#-----
```

```
puts "create JAASAuthData object for alias1"
```

```
set alias_attr [list alias $aliasName]
set desc_attr [list description "Alias 1"]
set userid_attr [list userId $defaultUser1]
set password_attr [list password $defaultPassword1]
set attrs [list $alias_attr $desc_attr $userid_attr $password_attr]
```

```
set authdata [$AdminConfig create JAASAuthData $sec $attrs]
$AdminConfig save
```

```
puts "Installing DB2 datasource for XA"
```

```
puts "Finding the old JDBCProvider.."
#Remove the old jdbc provider...
set jps [$AdminConfig list JDBCProvider]
foreach jp $jps {
  set jpname [lindex [lindex [$AdminConfig show $jp {name}] 0] 1]
  if {($jpname == "FVTProvider")} {
    puts "Removing old JDBC Provider"
    $AdminConfig remove $jp
    $AdminConfig save
  }
}
```

```
#Get the server name...
```

```
puts "Finding the server $server"
set servlist [$AdminConfig list Server]
set servsize [llength $servlist]
foreach srvr $servlist {
  set sname [lindex [lindex [$AdminConfig show $srvr {name}] 0] 1]
  if {($sname == $server)} {
    puts "Found server $srvr"
```

```

    set serv $srvr
  }
}

puts "Finding the Resource Adapter"
set rsadapter [$AdminConfig list J2CResourceAdapter $serv]

#Now create a JDBC Provider for the data sources
puts "Creating the provider for COM.ibm.db2.jdbc.DB2XADDataSource"
set attrs1 [subst {{classpath $driverClassPath} {
  implementationClassName COM.ibm.db2.jdbc.DB2XADDataSource} {
  name "FVTProvider2"} {description "DB2 JDBC Provider"}}]
set provider1 [$AdminConfig create JDBCProvider $serv $attrs1]

#Create the first data source
puts "Creating the datasource fvtDS_1"
set attrs2 [subst {{name fvtDS_1} {description "FVT DataSource 1"}}]
set ds1 [$AdminConfig create DataSource $provider1 $attrs2]

#Set the properties for the data source.
set propSet1 [$AdminConfig create J2EEResourcePropertySet $ds1 {}]

set attrs3 [subst {{name databaseName} {type java.lang.String} {value $databaseName1}}]
$AdminConfig create J2EEResourceProperty $propSet1 $attrs3

set attrs10 [subst {{jndiName jdbc/fvtDS_1} {statementCacheSize 10}{
  datasourceHelperClassname com.ibm.websphere.rsadapter.DB2DataStoreHelper} {
  relationalResourceAdapter $rsadapter} {authMechanismPreference "BASIC_PASSWORD"} {
  authDataAlias $aliasName}}]
$AdminConfig modify $ds1 $attrs10

#Create the connection pool object...
$AdminConfig create ConnectionPool $ds1 {{connectionTimeout 1000} {maxConnections 30} {
  minConnections 1} {agedTimeout 1000} {reapTime 2000} {unusedTimeout 3000} }

#Now create the 4.0 data sources..
puts "Creating the 4.0 datasource fvtDS_3"
set ds3 [$AdminConfig create WAS40DataSource $provider1 {{name fvtDS_3} {
  description "FVT 4.0 DataSource"}}]

#Set the properties on the data source
set propSet3 [$AdminConfig create J2EEResourcePropertySet $ds3 {}]

#These attributes should be the same as fvtDS_1
set attrs4 [subst {{name user} {type java.lang.String} {value $defaultUser1}}]
set attrs5 [subst {{name password} {type java.lang.String} {value $defaultPassword1}}]
$AdminConfig create J2EEResourceProperty $propSet3 $attrs3
$AdminConfig create J2EEResourceProperty $propSet3 $attrs4
$AdminConfig create J2EEResourceProperty $propSet3 $attrs5
set attrs10 [subst {{jndiName jdbc/fvtDS_3} {databaseName $databaseName1}}]
$AdminConfig modify $ds3 $attrs10

$AdminConfig create WAS40ConnectionPool $ds3 {{orphanTimeout 3000} {connectionTimeout 1000} {
  minimumPoolSize 1} {maximumPoolSize 10} {idleTimeout 2000}}

#Now add a CMP connection factory for the JCA-compliant data source. This step is not
#necessary for Version 4 data sources, as they contain built-in CMP connection factories.
puts "Creating the CMP Connector Factory for fvtDS_1"
set attrs12 [subst {{name "FVT DS 1_CF"} {authMechanismPreference BASIC_PASSWORD} {
  cmpDataSource $ds1} {authDataAlias $aliasName}}]
set cf1 [$AdminConfig create CMPConnectorFactory $rsadapter $attrs12]

#Set the properties for the data source.

```

```
$AdminConfig create MappingModule $cf1 {{mappingConfigAlias "DefaultPrincipalMapping"} {  
  authDataAlias "alias1"}}}
```

```
$AdminConfig save
```

Test connection service:

WebSphere Application Server provides a test connection service for testing connections to the data sources that you configure for database access.

If the definition of your data source includes WebSphere variables, see the "Configuring WebSphere variables" topic to verify that you define them correctly. A variable cannot be found exception results from attempted use of a data source that is configured with undefined variables.

Activating the test connection service

There are three ways to activate the test connection service: through the administrative console, the wsadmin tool, or a Java stand-alone program. Each process invokes the same methods on the same MBean.

Administrative console

WebSphere Application Server allows you to test a connection from the administrative console by simply pushing a button: the *Data source collection*, *Data source settings*, *Version 4 data source collection*, and *Version 4 data source settings* pages all have **Test Connection** buttons. After you define and save a data source, you can click this button to ensure that the parameters in the data source definition are correct. On the collection page, you can select several data sources and test them all at once. Note that there are certain conditions that must be met first. For more information, see *Testing a connection with the administrative console*.

WsAdmin tool

The wsadmin tool provides a scripting interface to a full range of WebSphere Application Server administration activities. Because the Test Connection functionality is implemented as a method on an MBean, and wsadmin can invoke MBean methods, wsadmin can be utilized to test connections to data sources. You have two options for testing a data source connection through wsadmin:

The *AdminControl* object of wsadmin has a testConnection operation that tests the configuration properties of a data source object. For information, see *Testing a connection using wsadmin*.

You can also test a connection by invoking the MBean operation. Use "Example: Testing data source connection using wsadmin" in the information center as a guide for this technique.

Java stand-alone program

Finally, you can test a connection by executing the testConnection() method on the DataSourceCfgHelper MBean. This method allows you to pass the configuration ID of the configured data source. The Java program connects to a running Java Management Extensions (JMX) server to access the MBean. In a base installation of Application Server, you connect to the JMX server running in the application server, usually on port 8880.

The return value from this invocation is either 0, a positive number, or an exception. 0 indicates that the operation completed successfully, with no warnings. A positive number indicates that the operation completed successfully, with the number of warnings. An exception indicates that the test of the connection failed.

You can find an example of this code in *Example: Test a connection using testConnection(ConfigID)*.

Testing a connection with the administrative console:

After you have defined and saved a data source, you can click the **Test Connection** button to ensure that the parameters in the data source definition are correct. On the collection panel, you can select multiple data sources and test them all at once. Be sure that the following conditions are met before using the Test Connection button.

1. Depending on your specific needs, a valid *Authentication Data alias* might need to exist and be supplied on the data source panels.
2. If you are testing a connection using a WebSphere Application Server Version 4.0 type of data source, ensure that the *user* and *password* information is filled in.
3. If you used a WebSphere Environment entry for the class path or other fields, such as `#{DB2_JDBC_DRIVER_PATH}/db2java.zip`, make sure that you assign it a value in the *WebSphere Variables* page. Note that if you add a new WebSphere environment variable, or modify it, the process (application server) might need restarting.

4. Click **Test Connection**.

A Test Connection operation can have three different outcomes, each resulting in a different message being displayed in the messages panel of the page on which you press the Test Connection button.

- a. The test can complete successfully, meaning that a connection is successfully obtained to the database using the configured data source parameters. The resulting message states: Test Connection for data source *DataSourceName* on process *ProcessName* at node *NodeName* was successful.
- b. The test can complete successfully with warnings. This means that while a connection is successfully obtained to the database, warnings were issued. The resulting message states: Test Connection for data source *DataSourceName* on process *ProcessName* at node *NodeName* was successful with warning(s). View the JVM Logs for more details.

The **View the JVM Logs** text is a hyperlink that takes you to the JVM Logs console screen for the process.

- c. The test can fail. A connection to the database with the configured parameters is not obtained. The resulting message states: Test Connection failed for data source *DataSourceName* on process *ProcessName* at node *NodeName* with the following exception: *ExceptionText*. View the JVM Logs for more details.

Again, the text for **View the JVM Logs** is a hyperlink to the appropriate logs screen.

Testing a connection using wsadmin:

The *AdminControl* object of *wsadmin* has a *testConnection* operation that tests the configuration properties of a data source object. It takes a *configuration ID* as an argument.

Note: There is no way to pass a user ID and password to this invocation. This invocation can only be used for databases that do not require a user ID and password to make a connection (such as DB2 on a Windows machine), or for data sources that have a component-managed or container-managed authentication alias set on the data source object.

1. Invoke the *getid()* method for your data source.
2. Set the value of the *configuration id* to a variable.

```
set myds [AdminConfig getid /JDBCProvider:mydriver/DataSource:mydatasrc/]
```

where */JDBCProvider:mydriver/DataSource:mydatasrc/* is the data source you want to test. After you have the configuration ID of the data source, you can test the connection to the database.

3. Test the connection to the database.

```
$AdminControl testConnection $myds
```

Example: Test a connection using testConnection(ConfigID): This program uses JMX to connect to a running server and invoke the *testConnection* method on the *DataSourceCfgHelper* MBean.

```

/**
 * Description
 * Resource adapter test program to make sure that the MBean interfaces work.
 * Following interfaces are tested
 *
 * --- testConnection()
 *
 * We need following to run
 * C:\src>java -Djava.ext.dirs=
 *   C:\WebSphere\AppServer\lib;C:\WebSphere\AppServer\java\jre\lib\ext testDSGUI
 * must include jre for log.jar and mail.jar, else get class not found exception
 *
 */

import java.util.Iterator;
import java.util.Locale;
import java.util.Properties;
import java.util.Set;

import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.RuntimeMBeanException;
import javax.management.RuntimeOperationsException;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.ws.rsadapter.exceptions.DataStoreAdapterException;

public class testDSGUI {

    //Use port 8880 for base installation or port 8879 for ND installation
    String port = "8880";
    // String port = "8879";
    String host = "localhost";
    final static boolean verbose = true;

    // eg a configuration ID for DataSource declared at the node level for base
    private static final String resURI = "cells/cat/nodes/cat:resources.xml#DataSource_1";

    // eg a 4.0 DataSource declared at the node level for base
    // private static final String resURI =
    //   "cells/cat/nodes/cat:resources.xml#WAS40DataSource_1";

    // eg Cloudscape DataSource declared at the server level for base
    //private static final String resURI =
    //   "cells/cat/nodes/cat/servers/server1/resources.xml#DataSource_6";

    // eg node level DataSource for ND
    //private static final String resURI =
    //   "cells/catNetwork/nodes/cat:resources.xml#DataSource_1";

    // eg server level DataSource for ND
    //private static final String resURI =
    //   "cells/catNetwork/nodes/cat/servers/server1:resources.xml#DataSource_4";

    // eg cell level DataSource for ND
    //private static final String resURI = "cells/catNetwork:resources.xml#DataSource_1";

    public static void main(String[] args) {
        testDSGUI cds = new testDSGUI();
        cds.run(args);
    }
}

```

```

/**
 * This method tests the ResourceMbean.
 *
 * @param args
 * @exception Exception
 */
public void run(String[] args) {

    try {

System.out.println("Connecting to the application server.....");

        /*****
        /** Initialize the AdminClient
        *****/
Properties adminProps = new Properties();
adminProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
adminProps.setProperty(AdminClient.CONNECTOR_HOST, host);
adminProps.setProperty(AdminClient.CONNECTOR_PORT, port);
AdminClient adminClient = null;
try {
    adminClient = AdminClientFactory.createAdminClient(adminProps);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
    System.out.println("NLS: Cannot make a connection to the application server\n");
    ce.printStackTrace();
    System.exit(1);
}

        /*****
        /** Locate the Mbean
        *****/
ObjectName handle = null;
try {
    // Send in a locator string
    // eg for a Baseinstallation this is enough
    ObjectName queryName = new ObjectName("WebSphere:type=DataSourceCfgHelper,*");

    // for ND you need to specify which node/process you would like to test from
    // eg run in the server
//ObjectName queryName = new ObjectName(
//    "WebSphere:cell=catNetwork,node=cat,process=server1,type=DataSourceCfgHelper,*");
//    // eg run in the node agent
//ObjectName queryName =
//    // new ObjectName(
//    //    "WebSphere:cell=catNetwork,node=cat,process=nodeagent,type=DataSourceCfgHelper,*");
//    // eg run in the Deployment Manager
//ObjectName queryName =
//    // new ObjectName(
//    //    "WebSphere:cell=catNetwork,node=catManager,process=dmgr,type=DataSourceCfgHelper,*");
Set s = adminClient.queryNames(queryName, null);
Iterator iter = s.iterator();
while (iter.hasNext()) {
    // use the first MBean that is found
    handle = (ObjectName) iter.next();
    System.out.println("Found this ->" + handle);
}
if (handle == null) {
    System.out.println("NLS: Did not find this MBean>>" + queryName);
    System.exit(1);
}
} catch (MalformedObjectNameException mone) {
    System.out.println("Check the program variable queryName" + mone);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
    System.out.println("Cannot connect to the application server" + ce);
}

        /*****

```



```

        /**          Build parameters to pass to Mbean          */
        /*****
String[] signature = { "java.lang.String" };
Object[] params = { resURI };
Object result = null;

        if (verbose) {
System.out.println("\nTesting connection to the database using " + handle);
}

try {
        /*****
        /** Start to test the connection to the database          */
        /*****
result = adminClient.invoke(handle, "testConnection", params, signature);
} catch (MBeanException mbe) {
// ***** all user exceptions come in here
if (verbose) {
Exception ex = mbe.getTargetException(); // this is the real exception from the Mbean
System.out.println("\nNLS:Mbean Exception was received contains " + ex);
ex.printStackTrace();
System.exit(1);
}
} catch (InstanceNotFoundException infe) {
System.out.println("Cannot find " + infe);
} catch (RuntimeMBeanException rme) {
Exception ex = rme.getTargetException();
ex.printStackTrace(System.out);
throw ex;
} catch (Exception ex) {
System.out.println("\nUnexpected Exception occurred: " + ex);
ex.printStackTrace();
}

        /*****
        /** Process the result. The result will be the number of warnings          */
        /** issued. A result of 0 indicates a successful connection with          */
        /** no warnings.          */
        /*****

//A result of 0 indicates a successful connection with no warnings.
System.out.println("Result= " + result);

} catch (RuntimeOperationsException roe) {
Exception ex = roe.getTargetException();
ex.printStackTrace(System.out);
} catch (Exception ex) {
System.out.println("General exception occurred");
ex.printStackTrace(System.out);
}
}
}

```

Configuring J2EE Connector connection factories in the administrative console

1. Click **Resources**.
2. Click **Resource Adapters**.
3. Select a resource adapter under Resource Adapters.
4. Click **J2C Connection Factories** under Additional Properties .
5. Click **New**.
6. Specify *General Properties* .
7. Select the authentication preference.

8. Select aliases for **component-managed authentication**, **container-managed authentication**, or both. If none are available, or you want to define a different one, click **Apply > J2C Authentication Data Entries** under Related Items.
 - a. Click **J2C Auth Data Entries** under Related Items.
 - b. Click **New**.
 - c. Specify General Properties.
 - d. Click **OK**.
9. Click **OK**.
10. Click the J2C connection factory you just created.
11. Under *Additional Properties* click **Connection Pool**.
12. Change any values desired by clicking the property name.
13. Click **OK**.
14. Click **Custom Properties** under *Additional Properties*.
15. Click any property name to change its value. Note that **UserName** and **Password** if present, are overridden by the **component-managed authentication** alias you specified in a previous step.
16. Click **Save**.

Connection pool settings:

Use this page to configure connection pool settings.

This administrative console page is common to a range of resource types; for example, JDBC data sources and JMS queue connection factories. To view this page, the path depends on the type of resource, but generally you select an instance of the resource provider, then an instance of the resource type, then click **Connection Pool**. For example: click **Resources > JDBC Providers > JDBC_provider > Data Sources > data_source > Connection Pool**.

Connection Timeout:

Specifies the interval, in seconds, after which a connection request times out and a `ConnectionWaitTimeoutException` is thrown.

This value indicates the number of seconds a request for a connection waits when there are no connections available in the free pool and no new connections can be created, usually because the maximum value of connections in the particular connection pool has been reached. For example, if Connection Timeout is set to 300, and the maximum number of connections are all in use, the pool manager waits for 300 seconds for a physical connection to become available. If a physical connection is not available within this time, the pool manager initiates a `ConnectionWaitTimeout` exception. It usually does not make sense to retry the `getConnection()` method; if a longer wait time is required you should increase the Connection Timeout setting value. If a `ConnectionWaitTimeout` exception is caught by the application, the administrator should review the expected connection pool usage of the application and tune the connection pool and database accordingly.

If the Connection Timeout is set to 0, the pool manager waits as long as necessary until a connection becomes available. This happens when the application completes a transaction and returns a connection to the pool, or when the number of connections falls below the value of Maximum Connections, allowing a new physical connection to be created.

If Maximum Connections is set to 0, which enables an infinite number of physical connections, then the Connection Timeout value is ignored.

Data type	Integer
Units	Seconds

Default	180
Range	0 to max int

Maximum Connections:

Specifies the maximum number of physical connections that you can create in this pool.

These are the physical connections to the backend resource. Once this number is reached, no new physical connections are created and the requester waits until a physical connection that is currently in use returns to the pool, or a `ConnectionWaitTimeout` exception is issued.

For example, if the Maximum Connections value is set to 5, and there are five physical connections in use, the pool manager waits for the amount of time specified in Connection Timeout for a physical connection to become free.

If Maximum Connections is set to 0, the connection pool is allowed to grow infinitely. This also has the side effect of causing the Connection Timeout value to be ignored.

If multiple standalone application servers use the same data source, there is one pool for each application server. If clones are used, one data pool exists for each clone. Knowing the number of data pools is important when configuring the database maximum connections.

You can use the Tivoli Performance Viewer to find the optimal number of connections in a pool. If the number of concurrent waiters is greater than 0, but the CPU load is not close to 100%, consider increasing the connection pool size. If the Percent Used value is consistently low under normal workload, consider decreasing the number of connections in the pool.

Data type	Integer
Default	10
Range	0 to max int

Minimum Connections:

Specifies the minimum number of physical connections to maintain.

If the size of the connection pool is at or below the minimum connection pool size, the Unused Timeout thread does not discard physical connections. However, the pool does not create connections solely to ensure that the minimum connection pool size is maintained. Also, if you set a value for Aged Timeout, connections with an expired age are discarded, regardless of the minimum pool size setting.

For example if the Minimum Connections value is set to 3, and one physical connection is created, the Unused Timeout thread does not discard that connection. By the same token, the thread does not automatically create two additional physical connections to reach the Minimum Connections setting.

Data type	Integer
Default	1
Range	0 to max int

Reap Time:

Specifies the interval, in seconds, between runs of the pool maintenance thread.

For example, if Reap Time is set to 60, the pool maintenance thread runs every 60 seconds. The *Reap Time* interval affects the accuracy of the *Unused Timeout* and *Aged Timeout* settings. The smaller the

interval, the greater the accuracy. If the pool maintenance thread is enabled, set the Reap Time value less than the values of Unused Timeout and Aged Timeout. When the pool maintenance thread runs, it discards any connections remaining unused for longer than the time value specified in Unused Timeout, until it reaches the number of connections specified in *Minimum Connections*. The pool maintenance thread also discards any connections that remain active longer than the time value specified in Aged Timeout.

The Reap Time interval also affects performance. Smaller intervals mean that the pool maintenance thread runs more often and degrades performance.

To disable the pool maintenance thread set Reap Time to 0, or set both Unused Timeout and Aged Timeout to 0. The recommended way to disable the pool maintenance thread is to set Reap Time to 0, in which case Unused Timeout and Aged Timeout are ignored. However, if Unused Timeout and Aged Timeout are set to 0, the pool maintenance thread runs, but only physical connections which timeout due to non-zero timeout values are discarded.

Data type	Integer
Units	Seconds
Default	180
Range	0 to max int

Unused Timeout:

Specifies the interval in seconds after which an unused or idle connection is discarded.

Set the Unused Timeout value higher than the Reap Timeout value for optimal performance. Unused physical connections are only discarded if the current number of connections exceeds the Minimum Connections setting. For example, if the unused timeout value is set to 120, and the pool maintenance thread is enabled (Reap Time is not 0), any physical connection that remains unused for two minutes is discarded. Note that accuracy of this timeout, as well as performance, is affected by the Reap Time value. See Reap Time for more information.

Data type	Integer
Units	Seconds
Default	1800
Range	0 to max int

Aged Timeout:

Specifies the interval in seconds before a physical connection is discarded.

Setting *Aged Timeout* to 0 supports active physical connections remaining in the pool indefinitely. Set the Aged Timeout value higher than the Reap Timeout value for optimal performance. For example, if the Aged Timeout value is set to 1200, and the Reap Time value is not 0, any physical connection that remains in existence for 1200 seconds (20 minutes) is discarded from the pool. Note that accuracy of this timeout, as well as performance, are affected by the Reap Time value. See Reap Time for more information.

Data type	Integer
Units	Seconds
Default	0
Range	0 to max int

Purge Policy:

Specifies how to purge connections when a *stale connection* or *fatal connection error* is detected.

Valid values are **EntirePool** and **FailingConnectionOnly**. JCA data sources can have either option. WebSphere Version 4.0 data sources always have a purge policy of **EntirePool**.

Data type
Default
Range

String
EntirePool

EntirePool

All connections in the pool are marked stale. Any connection not in use is immediately closed. A connection in use is closed and issues a stale connection Exception during the next operation on that connection. Subsequent getConnection() requests from the application result in new connections to the database opening. When using this purge policy, there is a slight possibility that some connections in the pool are closed unnecessarily when they are not stale. However, this is a rare occurrence. In most cases, a purge policy of EntirePool is the best choice.

FailingConnectionOnly

Only the connection that caused the stale connection exception is closed. Although this setting eliminates the possibility that valid connections are closed unnecessarily, it makes recovery from an application perspective more complicated. Because only the currently failing connection is closed, there is a good possibility that the next getConnection() request from the application can return a connection from the pool that is also stale, resulting in more stale connection exceptions.

The connection pretest function attempts to insulate an application from pooled connections that are not valid. When a backend resource, such as a database, goes down, pooled connections that are not valid might exist in the free pool. This is especially true when the purge policy is failingConnectionOnly; in this case, the failing connection is removed from the pool. Depending on the failure, the remaining connections in the pool might not be valid.

Connection pool advanced settings:

Use this page to specify connection pooling related settings.

Properties-shared partitions, free pool partitions, and free pool distribution table size are properties related to reducing the time a thread needs to wait for a synchronization lock.

Partition support is always enabled. The default values of 0 should be used enabling the connection pool to pick the best values for performance. In some cases where large multiprocessor systems are used, adjusting the partition support properties might help performance.

To view this administrative console page, click **Resources > Resource Adapters > resource_adapter > J2C Connection Factories > connection_factory > Connection Pool > Advanced Connection Pool**.

Number of shared partitions:

Specifies the number of partitions that are created in each of the shared pools.

Data type	integer
Default value	0
Range	0 to max int

Number of free pool partitions:

Specifies the number of partitions that are created in each of the free pools.

Data type	integer
Default value	0
Range	0 to max int

Free pool distribution table size:

The free pool distribution table size is used for better distribution of the Subject and CRI hash values within a hash table to minimize collisions for faster retrieval of a matching free connection.

If there are many incoming requests with varying credentials, this value can help with the distribution of finding a free pool for a connection for that user. Larger values are more common for installations that have many different credentials accessing the resource. Smaller values (1) should be used if the same credentials apply to all incoming requests for the resource.

Data type	integer
Default value	0
Range	0 to max int

Surge threshold:

Specifies the number of connections created before surge protection is activated.

Surge protection is designed to prevent overloading of a data source when too many connections are created at the same time. Surge protection is controlled by two properties, *surge threshold* and *surge creation interval*.

The surge threshold property specifies the number of connections created before surge protection is activated. After you reach the specified number of connections, you enter *surge mode*.

The surge creation interval property specifies the amount of time, in seconds, between the creation of connections when in surge mode.

For example, assume the follow settings:

- maxConnections = 50
- surgeThreshold = 10
- surgeCreationInterval = 30 seconds

If the connection pool receives 15 connection requests, 10 connections are created at about the same time. The 11th connection is created 30 seconds after the first 10 connections. The 12th connection is created 30 seconds after the 11th connection. Connections continue to be created every 30 seconds until there are no more new connections needed or you reach the maxConnections value.

Surge connection support starts if the surge threshold is > -1 and the surge creation interval is > 0. The surge threshold property has a default value of -1, which indicates that it is turned off.

wsadmin examples

```
$AdminControl getAttribute $objectname surgeCreationInterval
$AdminControl setAttribute $objectname surgeCreationInterval 30
$AdminControl getAttribute $objectname surgeThreshold
$AdminControl setAttribute $objectname surgeThreshold 15
```

Data type	integer
Default value	-1
Range	-1 to max int

Surge creation interval:

Specifies the amount of time between connection creates when you are in surge protection mode.

If the number of connections specified in the surge threshold property have been made, each request for a new connection must wait to be created on the surge creation interval. This property has a default value of 20, which indicates that at least 20 seconds should pass between connections being created. Valid values for this property are any positive integer.

Data type	integer
Default value	20
Range	0 to max int

Stuck timer time:

A *stuck* connection is an active connection that is not responding or returning to the connection pool. If the pool appears to be stuck (you have reached the stuck threshold), a resource exception is given to all new connection requests until the pool is unstuck. The stuck timer time property is the interval for the timer. This is how often the connection pool checks for stuck connections. The default value is 5 seconds.

If an attempt to change the stuck time, stuck timer time, or stuck threshold properties using the wsadmin scripting tool fails, an `IllegalStateException` occurs. The pool cannot have any active requests or active connections during this request. For the stuck connection support to start, all three stuck property values must be greater than 0 and maximum connections must be greater than 0.

Also, the stuck timer time, if it is set, must be less than the stuck time value. In fact, it is suggested that the stuck timer time should be one-quarter to one-sixth the value of stuck time so that the connection pool checks for stuck connections 4 to 6 times before a connection is declared stuck. This reduces the likelihood of false positives

wsadmin examples

```
$AdminControl getAttribute $objectname stuckTime
$AdminControl setAttribute $objectname stuckTime 30
$AdminControl getAttribute $objectname stuckTimerTime
$AdminControl setAttribute $objectname stuckTimerTime 15
$AdminControl getAttribute $objectname stuckThreshold
$AdminControl setAttribute $objectname stuckThreshold 10
```

Data type	integer
Default value	5
Range	0 to max int

Stuck time:

A *stuck* connection is an active connection that is not responding or returning to the connection pool. If the pool appears to be stuck (you have reached the stuck threshold), a resource exception is given to all new connection requests until the pool is unstuck. The stuck time property is the interval, in seconds, allowed for a single active connection to be in use to the backend resource before it is considered to be stuck.

Data type	integer
Default value	0
Range	0 to max int

Stuck threshold:

A *stuck* connection is an active connection that is not responding or returning to the connection pool. If the pool appears to be stuck (you have reached the stuck threshold), a resource exception is given to all new connection requests until the pool is unstuck. An application can explicitly catch this exception and continue processing. The pool will continue to periodically check for stuck connections when the number of stuck connections is past the threshold. If the number of stuck connections drops below the stuck threshold, the pool will detect this during its periodic checks and enable the pool to begin servicing requests again. The stuck threshold is the number of connections that need to be considered stuck for the pool to be in stuck mode.

Data type	integer
Default value	0
Range	0 to max int

Connection pool (Version 4) settings:

Use this page to create a connection pool for a Version 4.0 data source.

To view this administrative console page, click **Resources > JDBC Providers > JDBC_provider > Data Sources (Version 4) > data_source > Connection Pool**.

Scope:

Specifies the level to which this resource definition is visible -- the cell, node, or server level.

Resources such as JDBC providers, namespace bindings, or shared libraries can be defined at multiple scopes, with resources defined at more specific scopes overriding duplicates which are defined at more general scopes.

Note that no matter what the scope of a defined resource, the resource's properties only apply at an individual server level. For example, if you define the scope of a data source at the cell level, all users in that cell can look up and use that data source, which is unique within that cell. However, resource property settings are local to each server in the cell. For example, if you define *max connections* to 10, then each server in that cell can have 10 connections.

Cell The most general scope. Resources defined at the cell scope are visible from all nodes and servers, unless they are overridden. To view resources defined in the cell scope, do not specify a server or a node name in the scope selection form.

Node The default scope for most resource types. Resources defined at the node scope override any duplicates defined at the cell scope and are visible to all servers on the same node, unless they are overridden at a server scope on that node. To view resources defined in a node scope, do not specify a server, but select a node name in the scope selection form.

Server The most specific scope for defining resources. Resources defined at the server scope override

any duplicate resource definitions defined at the cell scope or parent node scope and are visible only to a specific server. To view resources defined in a server scope, specify a server name as well as a node name in the scope selection form.

When resources are created, they are always created into the current scope selected in the panel. To view resources in other scopes, specify a different node or server in the scope selection form.

Data type String

Minimum Pool Size:

Specifies the minimum number of connections to maintain in the pool.

The minimum pool size can affect the performance of an application. Smaller pools require less overhead when the demand is low because fewer connections are held open to the database. When the demand is high, the first applications experience a slow response because new connections are created if all others in the pool are in use.

Data type Integer
Default 1
Range Any non-negative integer.

Maximum Pool Size:

Specifies the maximum number of connections to maintain in the pool.

If the maximum number of connections is reached and all connections are in use, additional requests for a connection wait up to the number of seconds specified as the connection timeout. The maximum pool size can affect the performance of an application. Larger pools require more overhead when demand is high because there are more connections open to the database at peak demand. These connections persist until idled out of the pool. If the maximum value is smaller, longer wait times or possible connection timeout errors during peak times can occur. Ensure that the database can support the maximum number of connections in the application server, in addition to any load that it has outside of the application server.

Data type Integer
Default 10
Range Any positive integer

Connection Timeout:

Specifies the maximum number of seconds an application waits for a connection from the pool before timing out and issuing a `ConnectionWaitTimeout` exception to the application.

Setting this value to 0 disables the connection timeout.

Data type Integer
Units Seconds
Default 180
Range Any non-negative integer

Idle Timeout:

Specifies the maximum number of seconds that an idle (unallocated) connection can remain in the pool before being removed to free resources.

Connections need to idle out of the pool because keeping connections open to the database can cause database memory problems. However, not all connections are idled out of the pool, even if they are older than the Idle Timeout setting. A connection is not idled if removing the connection would cause the pool to shrink below its minimum size. Setting this value to 0 disables the idle timeout.

Data type	Integer
Units	Seconds
Default	1800
Range	Any non-negative integer

Orphan Timeout:

Specifies the maximum number of seconds that an application can hold a connection without using it before the connection returns to the pool

If there is no activity on an allocated connection for longer than the Orphan Timeout setting, the connection is marked for orphaning. After another Orphan Timeout number of seconds, if the connection still has no activity, the connection returns to the pool. If the application tries to use the connection again, it is issued a stale connection exception. Connections that are enlisted in a transaction are not orphaned. Setting this value to 0 disables the orphan timeout.

Data type	Integer
Units	Seconds
Default	1800
Range	Any non-negative integer

Statement Cache Size:

Specifies the number of cached prepared statements to keep per connection.

The largest value you would need to set your cache size to if you do not want any cache discards is determined as follows: for each application that uses this data source on a particular server, add up the number of unique prepared statements (as determined by the *sql* string, concurrency, and the scroll type). This is the maximum number of possible prepared statements that can be cached on a given connection over the life of the server. Setting the cache size to this value means you never have cache discards. This provides better performance. However, because of potential resource limitations, this might not always be possible.

Data type	Integer
Default	10
Range	Any non-negative integer

Auto Connection Cleanup:

Specifies whether or not the connection pooling software automatically closes connections from this data source at the end of a transaction.

The default is *false*, which indicates that when a transaction completes, WebSphere Application Server closes the connection and returns it to the pool. Any use of the connection after the transaction has ended results in a stale connection exception because the connection is closed and has returned to the pool. This mechanism ensures that connections are not held indefinitely by the application. If the value is set to *true*, the connection is not returned to the pool at the end of a transaction. In this case, the application must return the connection to the pool by calling *close()*. If the application does not close the connection, the pool can run out of connections for other applications to use.

Data type
Default

Check box
False (clear)

Configuring connection factories for resource adapters within applications:

1. Click **Applications**.
2. Click **Install New Application**.
3. Browse to find the appropriate EAR file, which contains an RAR file.
4. Click **Next**.
5. Select **resource ref mapping to a J2C Connection Factory**, then click **Next**.
6. After the application installs, click **Applications**.
7. Select the application just installed.
8. Click **Connector Modules** under Related Items.
9. Select an RAR file name on the Connector Modules page.
10. Click **Resource Adapter** under Additional Properties.
11. Click **J2C Connection Factories** under Additional Properties.
12. Click **New**.
13. Specify General Properties.
14. Select a **Component-managed authentication alias** if any application components with *Application* or *Per connection factory* authentication specified in the resource reference are going to be getting connections from this connection factory using the empty-argument `getConnection()` method. For resources supporting XA, you can optionally specify an Authentication alias for XA recovery. If a desired alias is not available, or you want to define a different one, click **Apply > J2C Authentication Data Entries** under Related Items.
 - a. Click **J2C Auth Data Entries** under Related Items.
 - b. Click **New**.
 - c. Specify General Properties.
 - d. Click **OK**.
15. Click **OK**.
16. Click the J2C connection factory you just created.
17. Click **Connection Pool** under Additional Properties .
18. Change any values desired by clicking on the property name.
19. Click **OK**.
20. Click **Custom Properties** under Additional Properties.
21. Click any property name to change its value. Note that **UserName** and **Password** if present, are overridden by the **component-managed authentication** alias you specified in a previous step.
22. Click **Save**.

J2C Connection Factories collection:

Use this page to select a connection factory, which represents one set of connection configuration values.

Application components such as enterprise beans have resource reference descriptors that refer to the connection factory, not the resource adapter. The connection factory is really a configuration properties list holder. In addition to the arbitrary set of configuration properties defined by the vendor of the resource adapter, there are several standard configuration properties that apply to the connection factory. These standard properties are used by the Java 2 Connectors connection pool manager in the application server run time and are not known by the vendor-supplied resource adapter code.

To view this administrative console page, click **Resources > Resource Adapters > resource_adapter > J2C Connection Factories**.

Name:

Specifies a list of the connection factory display names.

Data type String

JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name of this connection factory.

Data type String

Description:

Specifies a text description of this connection factory.

Data type String

Category:

Specifies a string that you can use to classify or group this connection factory.

Data type String

J2C Connection Factories settings:

Use this page to specify settings for a connection factory.

To view this administrative console page, click **Resources > Resource Adapters > resource_adapter > J2C Connection Factories > connection_factory**.

Name:

Specifies a list of connection factory display names.

This is a required property.

Data type String

JNDI Name:

Specifies the JNDI name of this connection factory.

For example, the name could be *eis/myECIConnection*.

After you set this value, save it and restart the server. You can see this string when you run the *dumpNameSpace* tool. This is a required property. If you do not specify a JNDI name, it is filled in by default using the Name field.

Data type String
Default *eis/display name*

Description:

Specifies a text description of this connection factory.

Data type String

Connection Factory Interface:

Specifies the fully qualified name of the Connection Factory Interfaces supported by the resource adapter.

This is a required property. For new objects, the list of available classes is provided by the resource adapter in a drop-down list. After you create the connection factory, the field is a read only text field.

Data type Drop-down list or text

Category:

Specifies a string that you can use to classify or group this connection factory.

Data type String

Authentication Alias for XA Recovery:

This optional field is used to specify the authentication alias that should be used during XA recovery processing.

If the resource adapter does not support XA transactions, then this field will not be displayed. The default value will come from the selected alias for application authentication (if specified).

Use Component-managed Authentication Alias

Selecting this radio button specifies that the alias set for component-managed authentication is used at XA recovery time.

Data type Radio button

Specify:

Selecting this radio button enables you to choose an authentication alias from a drop-down list of configured aliases.

Data type Radio button

Component-managed Authentication Alias:

Specifies authentication data for component-managed signon to the resource.

Choose from aliases defined under **Security>JAAS Configuration> J2C Authentication Data**.

To define a new alias not already appearing in the pick list:

- Click **Apply** to expose Related Items.
- Click **J2C Authentication Data Entries**.
- Define an alias.
- Click the connection factory name at the top of the *J2C Authentication Data Entries* page to return to the connection factory page.

- Select the alias.

Data type Pick-list

Container-managed Authentication Alias (deprecated):

Specifies authentication data (a string path converted to userid and password) for container-managed signon to the resource.

Note: Beginning with WebSphere Application Server Version 6.0, the container-managed authentication alias is superseded by the specification of a login configuration on the resource-reference mapping at deployment time, for components with *res-auth=Container*.

Choose from aliases defined under **Security>JAAS Configuration> J2C Authentication Data**.

To define a new alias not already appearing in the pick list:

- Click **Apply** to expose Related Items.
- Click **J2C Authentication Data Entries**.
- Define an alias.
- Click the connection factory name at the top of the *J2C Authentication Data Entries* page to return to the connection factory page.
- Select the alias.

Data type Pick-list

Authentication Preference (deprecated):

Specifies the authentication mechanisms defined for this connection factory.

Note: Beginning with WebSphere Application Server Version 6.0, the authentication preference is superseded by the combination of the `<res-auth>` application component deployment descriptor setting and the specification of a login configuration on the resource-reference mapping at deployment time.

This setting specifies which of the authentication mechanisms defined for the corresponding resource adapter applies to this connection factory. Common values, depending on the capabilities of the resource adapter, are: *KERBEROS*, *BASIC_PASSWORD*, and *None*.

If *None* is chosen, the application component is expected to manage authentication (`<res-auth>Application</res-auth>`). In this case, the user ID and password are taken from one of the following:

- The component-managed authentication alias
- UserName, Password Custom Properties
- Strings passed on the getConnection method

For example, if two authentication mechanism entries are defined for a resource adapter in the *ra.xml* document:

- `<authentication-mechanism-type>BasicPassword</authentication-mechanism-type>`
- `<authentication-mechanism-type>Kerbv5</authentication-mechanism-type>`

the authentication preference specifies the mechanism to use for container-managed authentication. An exception is issued during server startup if a mechanism that is not supported by the resource adapter is selected.

Data type Pick-list
Default BASIC_PASSWORD

Mapping-Configuration Alias (deprecated):

Allows users to select from the **Security > JAAS Configuration > Application Logins Configuration** list.

Note: Beginning with WebSphere Application Server Version 6.0, the Mapping-Configuration Alias is superseded by the specification of a login configuration on the resource-reference mapping at deployment time, for components with *res-auth=Container*.

The **DefaultPrincipalMapping** JAAS configuration maps the authentication alias to the userid and password. You may define and use other mapping configurations.

Data type Pick-list

J2C Connection Factory advanced settings:

Use this page to specify settings for a connection factory.

To view this administrative console page, click **Resources > Resource Adapters > resource_adapter > J2C Connection Factories > connection_factory > Advanced Connection Factory Properties**.

Manage cached handles:

Specifies whether cached handles (handles held in inst vars in a bean) should be tracked by the container.

Data type Checkbox

Log missing transaction contexts:

Specifies whether or not the container logs that there is a missing transaction context when a connection is obtained.

Data type Checkbox

Connection factory JNDI name tips: Distributed computing environments often employ naming and directory services to obtain shared components and resources. Naming and directory services associate names with locations, services, information, and resources.

Naming services provide name-to-object mappings. Directory services provide information on objects and the search tools required to locate those objects. There are many naming and directory service implementations, and the interfaces to them vary.

Java Naming and Directory Interface (JNDI) provides a common interface that is used to access the various naming and directory services. After you have set this value, saved it, and restarted the server, you should be able to see this string when you invoke name space dump tool.

For WebSphere Application Server specifically, when you create a data source the default JNDI name is set to *jdbc/data_source_name*. When you create a connection factory, its default name is *eis/j2c_connection_factory_name*. You can, of course, override these values by specifying your own.

In addition, if you click the checkbox for the *Use this data source for container managed persistence (CMP)* option when you create the data source, another reference is created with the name of *eis/jndi_name_of_datasource_CMP*. For example, if a data source has a JNDI name of

jdbc/myDatasource, the CMP JNDI name is *eis/jdbc/myDatasource_CMP*. This name is used internally by CMP and is provided simply for informational purposes.

When creating a connection factory or data source, a JNDI name is given by which the connection factory or data source can be looked up by a component. Preferably an "indirect" name with the *java:comp/env* prefix should be used and must be used in future releases. An "indirect" name makes any resource-reference data associated with the application available to the connection management runtime, to better manage resources based on the *res-auth*, *res-isolation-level*, *res-sharing-scope*, and *res-resolution-control* settings.

Though you can use a direct JNDI name, this naming method is deprecated in Version 6 of WebSphere Application Server. Application Server assigns default values to the resource-reference data when you use this method. An informational message, resembling the following, is logged to document the defaults:

```
J2CA0294W: Deprecated usage of direct JNDI lookup of resource jdbc/IOPEntity.  
The following default values are used: [Resource-ref settings]
```

```
res-auth:                1 (APPLICATION)  
res-isolation-level:    0 (TRANSACTION_NONE)  
res-sharing-scope:      true (SHAREABLE)  
loginConfigurationName: null  
loginConfigProperties:  null  
[Other attributes]
```

```
res-resolution-control: 999 (undefined)  
isCMP1_x:                false (not CMP1.x)  
isJMS:                   false (not JMS)
```

These default values can lead to unexpected behavior in your resources. For example, an application component (such as a JAAS login module) that accesses a resource with container-managed authentication data might fail to authenticate with the backend resource. Because the *res-auth* setting is assigned the default level of *Application*, rather than *Container*, the application server cannot find it.

This message can occur when you try using the fully qualified names of resources when looking up resources through Java Naming Directory Interface (JNDI). The J2EE programming model recommends the use of resource references and the local JNDI *java:comp/env* context. To correct this problem, modify the application to use the preferred J2EE programming model with resource references and the local JNDI *java:comp/env* context.

Security of lookups with component managed authentication

External Java clients (stand alone clients or servers from other cells) with Java Naming and Directory Interface (JNDI) access can look up a Java 2 Connector (J2C) resource such as a data source or Java Message Service (JMS) queue. However, they are not permitted to take advantage of the component managed *authentication alias* defined on the resource. This alias is a default value used when the *user* and *password* are not supplied on the *getConnection()* call. Therefore, if an external client needs to get a connection, it must assume responsibility for the authentication by passing it through arguments on the *getConnection()* call.

Any client running in the WebSphere Application Server process (such as a Servlet or an enterprise bean) within the same cell that can look up a resource in the JNDI namespace can obtain connections without explicitly providing authentication data on the *getConnection()* call. In this case, if the component's *res-auth* setting is **Application**, authentication is taken from the component-managed authentication alias defined on the connection factory. With *res-auth* set to **Container**, authentication is taken from the login configuration defined on the component's resource-reference. It is important to note that J2C authentication alias is per **cell**. An enterprise bean or Servlet in one application server cannot look up a resource in another server process which is in a different cell, because the alias would not be resolved.

Configuring data access for the Application Client

Configuring data access for the Application Client involves specifying the resource reference and associated database information required for data access. This specification is done as part of the assembly and deployment steps for the Application Client.

There are two tools needed to configure data sources used by J2EE application clients:

- An assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer for defining the resource reference in the deployment descriptor; and
- The Application Client Resource Configuration Tool (ACRCT) for defining the connection to the database in the client deployment environment.

Data access from an application client uses the JDBC driver connection functions directly from the client side. It does not take advantage of the additional pooling support available in the WebSphere Application Server run time. Configuring data access for an application client does not require configuration of a JDBC provider and data source on the WebSphere Application Server server machine.

If you want to take advantage of the pooling and additional database functions provided by WebSphere Application Server, it is recommended that your client application utilize an enterprise bean running on the server side to perform data access.

Defining an application client resource reference using an assembly tool

1. Assemble your application client module as described in “Assembling application clients” on page 184.
2. Create a new resource reference:
 - a. In a Project Explorer view, right-click your application client module and click **Open With > Deployment Descriptor Editor**.
 - b. On the **References** tab, click **Add > Resource reference > Next**.
 - c. On the Resource Reference page, enter the **Name** of this resource reference. The Application Client for WebSphere Application Server run time uses this name for two purposes: to bind the object into the *java:comp/env* portion of the JNDI namespace, and to find client specific configuration information. If the code for the Application Client performs a lookup for *java:comp/env/jdbc/myDB*, the name of the resource reference should be *jdbc/myDB*.
 - d. For **Type**, select *javax.sql.DataSource* for JDBC connections.
 - e. For **Authentication**, select *Application* if your client application intends to provide authentication information. If the Application Client for WebSphere Application Server run time provides the authentication information (as configured by the Application Client Resource Configuration tool), select *Container*.
 - f. Ignore the **Sharing scope** setting; it is unused in an application client resource reference. All Application Client resources are not shared.
 - g. Click **Finish**.
 - h. Close the deployment descriptor and save your changes.

The JNDI name field appears under **WebSphere Bindings** after your add the reference.

Client configuration with the ACRCT:

There are two client resources for you to configure in the Application Client Resource Configuration Tool (ACRCT) to enable data access from an application client: a data source provider and a data source.

Notes

Note: The following WebSphere objects, which can be bound into the server name space, are not supported on the client:

- Java 2 Connector (J2C) objects

- Connection manager objects

The WebSphere Application Server Client does not provide client database drivers. If your client application uses a database directly, rather than using an enterprise bean, you must provide the database drivers on the client machine. This action can involve contacting your database vendor to acquire client database driver code and licenses.

Instead of accessing the database directly, it is recommended that your client application use an enterprise bean. Accessing a database through an enterprise bean eliminates the need to have database drivers on the client machine because the database access is handled by the enterprise bean running on the WebSphere Application Server. Enterprise beans can also take advantage of the additional database functions provided by the WebSphere Application Server run time.

1. Configure a new data source provider as described in Configuring new data source providers. This provider describes the JDBC database implementation for your client application.
2. Enter the following information on the **General** tab:
 - a. A **name** for this data source provider.
 - b. Optional: A **description**.
 - c. The **classpath** to the data source provider implementation classes or JAR files. This is optional if the implementation classes or JAR files are already in the class path configuration of the client.
 - d. The name of the **implementation class**. For example, for DB2 this value is *COM.ibm.db2.jdbc.DB2DataSource*. Remember this class must implement the *javax.sql.DataSource* class. The ACRCT does not verify this class and you receive an error when you run your client application if the class does not implement *javax.sql.DataSource*.

Use the **Custom** tab to configure non-standard properties of the data source provider. This panel enables you to enter property-value pairs. During run time the *implementation class name* is created and any custom properties added on this panel are set on the newly created data source object using reflection. Any properties configured on this panel must have an appropriate set method on the data source class. For example, assume there is a property called *use2Phase* and its value should be 1. On the custom panel you enter the value *use2Phase* into the **name** column and the value 1 into the **value** column. The Application Client for WebSphere Application Server run time then uses reflection to find a property on the data source class called, typically *setUse2Phase* and call that method passing the value of 1. See your database product documentation for valid properties on your data source implementation.

3. Click **OK**.
4. Configure a new data source as described in Configuring new data sources for application clients. This describes the client properties of the database your client application uses.
5. Enter the following information on the **General** tab:
 - a. A **Name**. This field is required and identifies a name for the Application Client Resource Configuration Tool to use. This name is **not** used by your client application program.
 - b. Optional: A **description**.
 - c. The **JNDI name**. This field is required and must match the value entered in the **Name** field on the Add Resource Reference page of the assembly tool. In the example above, set this value to *jdbc/myDB*.
 - d. Optional: The **Database Name**.
 - e. Optional: Your *userid* in the **User** field.
 - f. Optional: Your *password* in the **Password** field. This password does not display.
 - g. Your password again to confirm in the **Re-Enter password** field. Note: The **User** and **Password** fields are used only when the **Authentication** field on the Add Resource Reference page of the assembly tool is set to *Container*.

Configuring Cloudscape Version 5.1.60x

Cloudscape provides the following two separate frameworks for running Cloudscape with WebSphere Application Server:

- **Embedded:** This framework is the same as the one for Cloudscape Version 5.0. To use this framework, define a Cloudscape JDBC provider. See the Cloudscape section in the minimum required settings article for more information.

You must use the embedded framework if you are running XA. Cloudscape does not support XA on Network Server.

- **Network Server:** This framework was a new feature in Cloudscape Version 5.1, and removes these limitations that existed in earlier versions of Cloudscape:
 - inability to access a remote Cloudscape instance
 - only one JVM can boot up the same database instance

The following steps describe how to configure and run the Network Server framework.

1. Start the Network Server on the machine that hosts the database instance.

To start the Network Server, run the **startNetworkServer.bat** file, which is located in the `WAS_HOME/cloudscape/bin/networkserver` directory. On UNIX platforms, the file is **startNetworkServer.sh**.

2. Update the **db2j.properties** file, which is located in the `WAS_HOME/cloudscape` directory, if necessary. Cloudscape should work without any modifications to this file.

Use the entries in the **db2j.properties** file to turn on trace, change the port number on which Network Server listens, and enable other functions of the Network Server framework. The default port number on which the Network Server listens is port 1527.

For more information on this file, see the Cloudscape documentation at www.ibm.com/software/data/cloudscape/pubs/collateral.html.

3. Define a Cloudscape Network Server using Universal JDBC driver to connect Cloudscape with WebSphere Application Server using the Network Server framework.

4. Stop the Network Server by invoking the **stopNetworkServer.bat** file.

You can find this file in the `WAS_HOME/cloudscape/bin/networkserver` directory. On UNIX platforms, the file is **stopNetworkServer.sh**.

5. Review additional tools available in the Network Server framework.

Find these tools in the `WAS_HOME/cloudscape/bin/networkserver` directory. These tools are:

- `sysinfo`
- `cview`
- `ij`
- `dropSYSIBM` Use this tool to drop the SYSIBM schema and its contents.

6. Create a SYSIBM schema.

If you do not create the SYSIBM schema, you cannot see the datatypes when you create tables using the `cview` graphical user interface. The **db2j.drda.loadSYSIBM** property in the **db2j.properties** file controls whether the schema is created on the first connection to the database. The **db2j.drda.loadSYSIBM** property default value is true.

Note: When you run `ij`, surround the dbname by double quotation marks (" ") if it includes the full path name; for example: `ij> connect "c:temp;create=true"`

This is ' ' ' ' without spaces.

Cloudscape Version 5.1.60x post installation instructions:

After installing Cloudscape Version 5.1.60x, you must complete the following steps before you can access the database.

1. Upgrade or migrate any existing database instances.
 - a. Backup an existing database.

You must complete a backup in case you have to access the previous version of Cloudscape. After you migrate a database, you cannot access your old database unless you perform a backup.

b. Migrate an existing database by doing the following:

- Set the **connectionAttributes** custom property to **upgrade=true**.

The data source is located in the WebSphere Application Server administrative console under the JDBC providers.

- If you are using the cview interface, located in the WAS_HOME/cloudscape51/bin/embedded directory, click **yes** when you see the *upgrade database* prompt.

Note: Ensure you migrate **defaultDB**, which is located in the WAS_HOME/bin/DefaultDB directory.

2. Set or change the class path definitions in any existing JDBC providers, which are defined to use Cloudscape. Cloudscape JAR files will not load when WebSphere Application Server is active.

Use the WebSphere Application Server environment variable

\${CLOUDSCAPE_JDBC_DRIVER_PATH}\db2j.jar to point to the new version of Cloudscape.

The **CLOUDSCAPE_JDBC_DRIVER_PATH** environment variable is defined in WebSphere Application Server with a value of WAS_HOME/cloudscape/lib.

3. If the application server is running Cloudscape as a persistent store for UDDI in previous versions, additional steps are necessary.

The server SystemOut log might issue this message:

The data source class name com.ibm.db2j.jdbc.db2jConnectionPoolDataSource could not be found.

This is because the Cloudscape JAR file has moved to from its location in version 5.x to a new location in version 5.1x.

To correct this situation, do the following:

- a. Upgrade the database to Cloudscape 5.1.60x.
- b. Rerun the install script, **or** edit the class path field in the data source.

DB2 tuning parameters

DB2 has many parameters that you can configure to optimize database performance. For complete DB2 tuning information, refer to the DB2 UDB Administration Guide: Performance.

DB2 logging

- **Description:** DB2 has corresponding log files for each database that provides services to administrators, including viewing database access and the number of connections. For systems with multiple hard disk drives, you can gain large performance improvements by setting the log files for each database on a different hard drive from the database files.
- **How to view or set:** At a DB2 command prompt, issue the command: **db2 update db cfg for [database_name] using newlogpath [fully_qualified_path]**.
- **Default value:** Logs reside on the same disk as the database.
- **Recommended value:** Use a separate high-speed drive, preferably performance enhanced through RAID.

For more information about using AIX with DB2 see "Tuning operating systems" in the information center.

DB2 configuration advisor

Located in the DB2 Control Center, this advisor calculates and displays recommended values for the DB2 buffer pool size, the database, and the database manager configuration parameters, with the option of applying these values. See more information about the advisor in the online help facility within the Control Center.

Use TCP/IP sockets for DB2 on Linux

- **Description:** On Linux platforms, whether the DB2 server resides on a local machine with WebSphere Application Server or on a remote machine, configure the DB2 application databases to use TCP/IP sockets for communications with the database.

- **How to view or set:** Locate the directions for configuring DB2 on Linux in the *Installing your application serving environment* PDF. This document specifies setting DB2COMM for TCP/IP and corresponding changes required in the etc/service file.
- **Default value:** Shared memory for local databases
- **Recommended value:** On Linux, change the specification for the DB2 application databases and any session databases from shared memory to TCP/IP sockets.

Number of connections to DB2 - MaxAppls and MaxAgents

When configuring the data source settings for the databases, confirm the DB2 MaxAppls setting is greater than the maximum number of connections for the data source. If you are planning to establish clones, set the MaxAppls value as the maximum number of connections multiplied by the number of clones. The same relationship applies to the session manager number of connections. The MaxAppls setting must be equal to or greater than the number of connections. If you are using the same database for session and data sources, set the MaxAppls value as the sum of the number of connection settings for the session manager and the data sources.

For example, MaxAppls = (# of connections set for data source + # of connections in session manager) x # of clones.

After calculating the MaxAppls settings for the WebSphere database and each of the application databases, verify that the MaxAgents setting for DB2 is equal to or greater than the sum of all of the MaxAppls values, for example, MaxAgents = sum of MaxAppls for all databases.

DB2 buffpage

- **Description:** Improves database system performance. Buffpage is a database configuration parameter. A buffer pool is a memory storage area where database pages containing table rows or index entries are temporarily read and changed. Data is accessed much faster from memory than from disk.
- **How to view or set:** To view the current value of buffpage for database *x*, issue the DB2 command **get db cfg for x** and look for the value **BUFFPAGE**. To set **BUFFPAGE** to a value of *n*, issue the DB2 command **update db cfg for x** using **BUFFPAGE n** and set **NPAGES** to -1 as follows:

```
db2 <-- go to DB2 command mode, otherwise the following "select" does not work as is
connect to x <-- (where x is the particular DB2 database name)
select * from syscat.bufferpools
    (and note the name of the default, perhaps: IBMDEFAULTBP)
    (if NPAGES is already -1, there is no need to issue following command)
alter bufferpool IBMDEFAULTBP size -1
(re-issue the above "select" and NPAGES now equals -1)
```

You can collect a snapshot of the database while the application is running and calculate the buffer pool hit ratio as follows:

1. Collect the snapshot:
 - a. Issue the **update monitor switches using bufferpool on** command.
 - b. Make sure that bufferpool monitoring is on by issuing the **get monitor switches** command.
 - c. Clear the monitor counters with the **reset monitor all** command.
 2. Run the application.
 3. Issue the **get snapshot for all databases** command before all applications disconnect from the database, otherwise statistics are lost.
 4. Issue the **update monitor switches using bufferpool off** command.
 5. Calculate the hit ratio by looking at the following database snapshot statistics:
 - Buffer pool data logical reads
 - Buffer pool data physical reads
 - Buffer pool index logical reads
 - Buffer pool index physical reads
- **Default value:** 250
 - **Recommended value:** Continue increasing the value until the snapshot shows a satisfactory hit rate.

The buffer pool hit ratio indicates the percentage of time that the database manager did not need to load a page from disk to service a page request. That is, the page was already in the buffer pool. The greater the buffer pool hit ratio, the lower the frequency of disk input and output. Calculate the buffer pool hit ratio as follows:

- P = buffer pool data physical reads + buffer pool index physical reads
- L = buffer pool data logical reads + buffer pool index logical reads
- Hit ratio = $(1-(P/L)) * 100\%$

DB2 query optimization level

- **Description:** Sets the amount of work and resources that DB2 puts into optimizing the access plan. When a database query is executed in DB2, various methods are used to calculate the most efficient access plan. The range is from zero to 9. An optimization level of 9 causes DB2 to devote a lot of time and all of its available statistics to optimizing the access plan.
- **How to view or set:** The optimization level is set on individual databases and can be set with either the command line or with the DB2 Control Center. Static SQL statements use the optimization level specified on the **prep** and **bind** commands. If the optimization level is not specified, DB2 uses the default optimization as specified by the `dft_queryopt` setting. Dynamic SQL statements use the optimization class specified by the current query optimization special register, which is set using the SQL Set statement. For example, the following statement sets the optimization class to 1:

```
Set current query optimization = 1
```

If the current query optimization register is not set, dynamic statements are bound using the default query optimization class.

- **Default value:** 5
- **Recommended value:** Set the optimization level for the needs of the application. Use high levels only when there are very complicated queries.

DB2 reorgchk

- **Description:** Obtains the current statistics for data and rebinding. Use this parameter because SQL statement performance can deteriorate after many updates, deletes or inserts.
- **How to view or set:** Use the DB2 **reorgchk update statistics on table all** command to issue runstats on all user and system tables for the database to which you are currently connected. Rebind packages using the **bind** command. If runstats exists, issue the **db2 -v "select tname, nleaf, nlevels, stats_time from sysibm.sysindexes"** command on DB2 CLP. If no runstats exist, nleaf and nlevels are -1, and stats_time has an empty entry, for example "-". If runstats was previously done, the real-time stamp from completion of the runstats also displays under stats_time. If you think the time shown for the previous runstats is too old, execute runstats again.
- **Default value:** None
- **Recommended value:** None

DB2 MinCommit

- **Description:** Delays the writing of log records to a disk until a minimum number of commits is performed, reducing the database manager overhead associated with writing log records. For example, if MinCommit is set to 2, a second commit causes output to the transaction log for the first and second commits. The exception occurs when a one-second timeout forces the first commit to be output if a second commit does not occur within one second.

In test applications, up to 90% of the disk input and output was related to the DB2 transaction log. Changing MinCommit from 1 to 2 reduced the results to 45%.

Adjust this parameter if the disk input and output wait is more than 5% and there is DB2 transaction log activity from multiple sources. When a lot of activity occurs from multiple sources, it is less likely that a single commit has to wait for another commit, or the one second timeout. Do not adjust this parameter if you have an application with a single thread performing a series of commits because each commit can hit the one second delay.

- **How to view or set:**

1. Issue the DB2 **get db cfg for xxxxxx** command, where xxxxxx is the name of the application database, to list database configuration parameters.
2. Look for "Group commit count (MINCOMMIT)".
3. Set a new value by issuing the DB2 **update db cfg for xxxxxx using mincommit n** command, where xxxxxx is the name of the application database and n is a value between 1 and 25 inclusive.

The new setting takes effect immediately.

The following are several metrics that are related to DB2 MinCommit:

- The disk input and output wait can be observed on AIX with the command **vmstat 5**. This shows statistics every 5 seconds. Look for the *wa* column under the CPU area.
 - The percentage of time a disk is active can be observed on AIX with the command **iostat 5**. This shows statistics every 5 seconds. Look for the *%tm_act* column.
 - The DB2 **get snapshot for db on xxxxxx** command, where xxxxxx is the name of the application database, shows counters for log pages read and log pages written.
- **Default value:** 1.
 - **Recommended value:** 1 or 2, if the circumstance permits.

DB2 Deadlock Event Monitor

- **Description:** For DB2 V8 or later, deadlock event monitor is turned on by default when a database is created. This event monitor captures critical information about all connections involved in deadlock when the particular event occurs. Although there is not much overhead on faster SMP systems, it can be turned off.
- **How to view or set:** In DB2 command window, issue following commands:

```
db2 connect to [db name]
db2 set event monitor db2detaildeadlock state 0
db2 drop event monitor db2detaildeadlock
db2 connect reset
db2 terminate
```

- **Default value:** Event monitor is ON by default.
- **Recommended value:** Turn OFF event monitor, if it is not required.

For more information about using AIX with DB2 see "Tuning operating systems."

Vendor-specific data sources minimum required settings

Use this table as an at-a-glance reference of JDBC providers that can be defined for use with WebSphere Application Server Version 6.x. A list that contains detailed descriptions for all of these providers follows the table.

Database type	JDBC Provider	Transaction support	Version and other considerations
DB2 on Windows, UNIX, or workstation-based LINUX	DB2 Universal JDBC Provider	One phase only	
	DB2 Universal JDBC Provider (XA)	One and two phase	
	DB2 legacy CLI-based Type 2 JDBC Provider	One phase only	
	DB2 legacy CLI-based Type 2 JDBC Provider (XA)	One and two phase	

Database type	JDBC Provider	Transaction support	Version and other considerations
DB2 UDB for iSeries	DB2 UDB for iSeries (Native)	One phase only	Recommended for use with WebSphere Application Server run on iSeries
	DB2 UDB for iSeries (Native XA)	One and two phase	Recommended for use with WebSphere Application Server run on iSeries
	DB2 UDB for iSeries (Toolbox)	One phase only	Recommended for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 UDB for iSeries (Toolbox XA)	One and two phase	Recommended for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 legacy CLI-based Type 2 JDBC Provider	One phase only	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect driver (available from DB2)
	DB2 legacy CLI-based Type 2 JDBC Provider (XA)	One and two phase	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect driver (available from DB2)

Database type	JDBC Provider	Transaction support	Version and other considerations
DB2 on z/OS	DB2 for z/OS Local JDBC Provider (RRS)	One and two phase	<i>Only</i> for use with WebSphere Application Server run on z/OS
	DB2 Universal JDBC Provider	One phase only	<i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 Universal JDBC Provider (XA)	One and two phase	<i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX
	DB2 legacy CLI-based Type 2 JDBC Provider	One phase only	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect program (available from DB2)
	DB2 legacy CLI-based Type 2 JDBC Provider (XA)	One and two phase	- <i>Only</i> for use with WebSphere Application Server run on Windows, UNIX, or workstation-based LINUX - Requires the DB2 Connect program (available from DB2)
Cloudscape	Cloudscape JDBC Provider	One phase only	- Not for use in clustering environment: Cloudscape is accessible from a single JVM only - Does not support Version 4 data sources
	Cloudscape JDBC Provider (XA)	One and two phase	- Not for use in clustering environment: Cloudscape is accessible from a single JVM only - Does not support Version 4 data sources
	Cloudscape Network Server using Universal JDBC driver	One phase only	Does not support Version 4 data sources
Informix	Informix JDBC Driver	One phase only	
	Informix JDBC Driver (XA)	One and two phase	
Sybase	Sybase JDBC Driver	One phase only	
	Sybase JDBC Driver (XA)	One and two phase	
Oracle	Oracle JDBC Driver	One phase only	
	Oracle JDBC Driver (XA)	One and two phase	

Database type	JDBC Provider	Transaction support	Version and other considerations
MS SQL Server	DataDirect ConnectJDBC type 4 driver for MS SQL Server	One phase only	Only for use with the corresponding driver from DataDirect Technologies
	DataDirect ConnectJDBC type 4 driver for MS SQL Server (XA)	One and two phase	Only for use with the corresponding driver from DataDirect Technologies
	WebSphere embedded ConnectJDBC driver for MS SQL Server	One phase only	- Not available for Application Server on z/OS - Cannot be used outside of WebSphere Application Server environment
	WebSphere embedded ConnectJDBC driver for MS SQL Server (XA)	One and two phase	- Not available for Application Server on z/OS - Cannot be used outside of WebSphere Application Server environment

Detailed JDBC provider list

The following list contains a description of every JDBC provider that can be defined for use with WebSphere Application Server Version 6.x. It also shows the supported data source classes and their required properties. Specific fields are designated for the *user* and *password* properties. Inclusion of a property in the list does not imply that you should add it to the data source properties list. Rather, inclusion in the list means that a value is typically required for that field.

Use these links to find your provider information:

- DB2 on Windows, UNIX, or workstation-based LINUX
- DB2 UDB for iSeries
- DB2 on z/OS, connecting to Application Server on Windows, UNIX, or workstation-based LINUX
- Cloudscape
- Informix
- Sybase
- Oracle
- MS SQL Server

DB2 on Windows, UNIX, or workstation-based LINUX

1. DB2 Universal JDBC Provider

The DB2 Universal JDBC Driver is an architecture-neutral JDBC driver for distributed and local DB2 access. Because the Universal Driver architecture is independent of any particular JDBC driver connectivity or target platform, it allows both Java connectivity (Type 4) or Java Native Interface (JNI) based connectivity (Type 2) in a single driver instance to DB2.

This JDBC driver allows applications to use both JDBC and Structured Query Language in Java (SQLJ) access.

The DB2 Universal JDBC Driver Provider supports one phase data source:

`com.ibm.db2.jcc.DB2ConnectionPoolDataSource`

Requires JDBC driver files:

- **db2jcc.jar** After you install DB2, you can find this jar file in the DB2 *java* directory. For Type 4 JDBC driver support from a client machine where DB2 is not installed, copy this file to the local machine. If you install any fixes or upgrades to DB2, you must update this file as well. You must

also set the **DB2UNIVERSAL_JDBC_DRIVER_PATH** path variable to point to the **db2jcc.jar** file. See the Cloudscape section for more information on the **DB2UNIVERSAL_JDBC_DRIVER_PATH** path variable.

Note: To find out the version of the universal driver you are using, issue this DB2 command:

```
java com.ibm.db2.jcc.DB2Jcc -version
```

The output for the above example is:

```
IBM DB2 JDBC Universal Driver Architecture 2.2.xx
```

- **db2jcc_license_cu.jar** This is the DB2 Universal JDBC driver license file that allows access to the DB2 Universal database. Use this jar file or the next one to gain access to the database. This jar file ships with WebSphere Application Server in a directory defined by **\${UNIVERSAL_JDBC_DRIVER_PATH}** environment variable.
- **db2jcc_license_cisuz.jar** This is the DB2 Universal JDBC driver license file that allows access to the following databases:
 - DB2 Universal
 - DB2 for iSeries
 - DB2 for z/OS
 - SQLDS

The **db2jcc_license_cisuz.jar** does not ship with Websphere Application Server and should be located in the same directory as the **db2jcc.jar** file, so that the **DB2UNIVERSAL_JDBC_DRIVER_PATH** points to both.

The classpath for this provider is set as follows:

```
<classpath>${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar </classpath>  
<classpath>${UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu.jar</classpath>  
<classpath>${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cisuz.jar</classpath>
```

Note: The license jar files are independent of each other; therefore, order does not matter.

Requires **DataStoreHelper** class:

```
com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper
```

Requires a valid authentication alias.

Requires properties:

- **databaseName** This is an actual database name if the **driverType** is set to **4**, or a locally cataloged database name if the **driverType** is set to **2**.
- **driverType** The JDBC connectivity type of a data source. *There are two permitted values: 2 and 4.* If you want to use Universal JDBC Type 2 driver, set this value to 2. If you want to use Universal JDBC Type 4 driver, set this value to 4.
- **serverName** The TCP/IP address or host name for the Distributed Relational Database Architecture (DRDA) server. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.
- **portNumber** The TCP/IP port number where the DRDA server resides. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.

2. DB2 Universal JDBC Provider (XA)

The DB2 Universal JDBC Provider (XA) is an architecture-neutral JDBC provider for distributed and local DB2 access. Whether you use this provider for Java connectivity or Java Native Interface (JNI) based connectivity depends on the version of DB2 you are running. Application Server Version 6.0 minimally requires DB2 8.1 Fix Pack 6. This version of DB2 only supports XA connectivity over the Java Native Interface (JNI) based connectivity (Type 2) driver. In order to use XA connectivity with the Type 4 driver, DB2 8.1 Fix Pack 7 or higher is required.

The DB2 Universal JDBC Driver (XA) supports two phase transactions and the more advanced data source option offered by Application Server (as opposed to the other option, Version 4 data sources). This driver also allows applications to use both JDBC and SQLJ access.

The DB2 Universal JDBC Driver Provider supports the two phase data source:

`com.ibm.db2.jcc.DB2XADataSource`

Requires JDBC driver files:

- **db2jcc.jar** After you install DB2, you can find this .jar file in the DB2 java directory. For Type 4 JDBC driver support from a client machine where DB2 is not installed, copy this file to the local machine. If you install any fixes or upgrades to DB2, you must update this file as well. You must also set the **DB2UNIVERSAL_JDBC_DRIVER_PATH** environment variable to point to the db2jcc.jar file. See the Cloudscape section for more information on the **DB2UNIVERSAL_JDBC_DRIVER_PATH** environment variable.

Note: To find the level of universal driver you are using, issue the following DB2 command:

```
java com.ibm.db2.jcc.DB2Jcc -version
```

example output of the above:

```
IBM DB2 JDBC Universal Driver Architecture 2.2.xx
```

- **db2jcc_license_cu.jar** This is the DB2 Universal JDBC driver license file that allows access to the DB2 Universal database. Use this jar file or the next one to gain access to the database. This jar file ships with WebSphere Application Server in the *WAS_HOME/universalDriver/lib* directory.
- **db2jcc_license_cisuz.jar** This is the DB2 Universal JDBC driver license file that allows access to the following databases:
 - DB2 Universal
 - DB2 for iSeries
 - DB2 for z/OS
 - SQLDS

You must use the right license jar file to access a specific database backend.

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **databaseName** This is an actual database name if the **driverType** is set to **4**, or a locally cataloged database name if the **driverType** is set to **2**.
- **driverType** The JDBC connectivity type of a data source. *There are two permitted values: 2 and 4.* If you want to use Universal JDBC Type 2 XA driver, set this value to 2. If you want to use Universal JDBC Type 4 XA driver (which requires DB2 8.1 Fix Pack 7 or higher), set this value to 4.
- **serverName** The TCP/IP address or host name for the Distributed Relational Database Architecture (DRDA) server. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.
- **portNumber** The TCP/IP port number where the DRDA server resides. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.

3. DB2 legacy CLI-based Type 2 JDBC Driver

The DB2 legacy CLI-based Type 2 JDBC Driver Provider is built on top of DB2 CLI (Call Level Interface). It uses the DB2 CLI interface to communicate with DB2 UDB servers.

DB2 legacy CLI-based Type 2 JDBC Driver supports one phase data source:

`COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`

Requires JDBC driver files: **db2java.zip** (Note: If you run SQLJ in DB2 Version 8, **db2jcc.jar** is also required.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2DataStoreHelper`

Does not require a valid authentication alias if Application Server is running on the same machine as the database. Otherwise, connectivity through this driver does require an alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

4. DB2 legacy CLI-based Type 2 JDBC Driver (XA)

The DB2 legacy CLI-based Type 2 JDBC Driver (XA) is built on top of DB2 CLI (Call Level Interface). It uses the DB2 CLI interface to communicate with DB2 UDB servers.

DB2 legacy CLI-based Type 2 JDBC Driver (XA) supports two phase data source:

COM.ibm.db2.jdbc.DB2XADataSource

Requires JDBC driver files: **db2java.zip** (Note: If you run SQLJ in DB2 Version 8, **db2jcc.jar** is also required.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2DataStoreHelper`

Does not require a valid authentication alias if Application Server is running on the same machine as the database. Otherwise, connectivity through this driver does require an alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

For more information on DB2, visit the DB2 Web site at: <http://www.ibm.com/software/data/db2/>.

For information on configuring WebSphere Application Server for DB2 access, see the "Configuring DB2" article in the information center.

DB2 UDB for iSeries

1. DB2 UDB for iSeries (Native)

The iSeries Developer Kit for Java contains this Type 2 JDBC driver that is built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R2, or later releases.

DB2 UDB for iSeries (Native V5R2 and later) supports one phase data source:

com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource

Requires JDBC driver files: **db2_classes.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

Does not require an authentication alias.

Requires properties:

- **databaseName** The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

2. DB2 UDB for iSeries (Native XA)

The iSeries Developer Kit for Java contains this XA-compliant Type 2 JDBC driver built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R2 or later releases.

DB2 UDB for iSeries (Native XA - V5R2 and later) supports two phase data source:

com.ibm.db2.jdbc.app.UDBXADataSource

Requires JDBC driver files: **db2_classes.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

Does not require an authentication alias.

Requires properties:

- **databaseName** The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

3. DB2 UDB for iSeries (Toolbox)

This JDBC driver, also known as iSeries Toolbox driver for Java, is provided in the DB2 for iSeries database server. Use this driver for remote DB2 connections on iSeries. We recommend you use this driver instead of the IBM Developer Kit for Java JDBC Driver to access remote DB2 UDB for iSeries systems.

DB2 UDB for iSeries (Toolbox) supports one phase data source:

com.ibm.as400.access.AS400JDBCConnectionPoolDataSource

Requires JDBC driver files: **jt400.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

Does not require an authentication alias if WebSphere Application Server and DB2 UDB for iSeries are installed in the same server. If they are installed in different servers, the user ID and password are required.

Requires properties:

- **serverName** The name of the server from which the data source obtains connections. Example: *myserver.mydomain.com*.

4. DB2 UDB for iSeries (Toolbox XA)

This XA compliant JDBC driver, also known as iSeries Toolbox XA compliant driver for Java, is provided in the DB2 for iSeries database server. Use this driver for remote DB2 connections on iSeries. We recommend you use this driver instead of the IBM Developer Kit for Java JDBC Driver to access remote DB2 UDB for iSeries systems.

DB2 UDB for iSeries (Toolbox XA) supports two phase data source:

com.ibm.as400.access.AS400JDBCXADataSource

Requires JDBC driver files: **jt400.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

Does not require an authentication alias if WebSphere Application Server and DB2 UDB for iSeries are installed in the same server. If they are installed in different servers, the user ID and password are required.

Requires properties:

- **serverName** The name of the server from which the data source obtains connections. Example: *myserver.mydomain.com*.

5. DB2 legacy CLI-based Type 2 JDBC Driver

The DB2 legacy CLI-based Type 2 JDBC Driver Provider is built on top of DB2 CLI (Call Level Interface). It uses the DB2 CLI interface to communicate with DB2 UDB servers. This provider is intended for *remote* connections to DB2 running on iSeries; for use with Application Server on Windows, UNIX, or workstation-based LINUX, it therefore requires the DB2 Connect Driver (which is available from DB2).

DB2 legacy CLI-based Type 2 JDBC Driver supports one phase data source:

COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource

Requires JDBC driver files: **db2java.zip** (Note: If you run SQLJ in DB2 Version 8, **db2jcc.jar** is also required.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2DataStoreHelper`

Does not require a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

6. **DB2 legacy CLI-based Type 2 JDBC Driver (XA)**

The DB2 legacy CLI-based Type 2 JDBC Driver (XA) is built on top of DB2 CLI (Call Level Interface). It uses the DB2 CLI interface to communicate with DB2 UDB servers. This provider is intended for *remote* connections to DB2 running on iSeries; for use with Application Server on Windows, UNIX, or workstation-based LINUX, it therefore requires the DB2 Connect Driver (which is available from DB2).

DB2 legacy CLI-based Type 2 JDBC Driver (XA) supports two phase data source:

COM.ibm.db2.jdbc.DB2XADataSource

Requires JDBC driver files: **db2java.zip** (Note: If you run SQLJ in DB2 Version 8, **db2jcc.jar** is also required.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2DataStoreHelper`

Does not require a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

7. **DB2 UDB for iSeries (Native - Version 5 Release 1 and earlier)** -- Deprecated

This JDBC provider is deprecated because it corresponds to a version of the iSeries operating system that WebSphere Application Server Version 6.x does not support. You must now use iSeries V5R2 or a later release of the iSeries operating system, for which the WebSphere Application Server administrative console lists one native iSeries DB2 non-XA provider: DB2 UDB for iSeries (Native).

The iSeries Developer Kit for Java contains this Type 2 JDBC driver that is built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R1, or earlier releases.

DB2 UDB for iSeries (Native V5R1 and earlier) supports one phase data source:

com.ibm.db2.jdbc.app.DB2StdConnectionPoolDataSource

Requires JDBC driver files: **db2_classes.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

Does not require an authentication alias.

Requires properties:

- **databaseName** The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

8. **DB2 UDB for iSeries (Native XA - Version 5 Release 1 and earlier)** -- Deprecated

This JDBC provider is deprecated because it corresponds to a version of the iSeries operating system that WebSphere Application Server Version 6.x does not support. You must now use iSeries V5R2 or a later release of the iSeries operating system, for which the administrative console lists one native iSeries DB2 XA provider: DB2 UDB for iSeries (Native XA).

The iSeries Developer Kit for Java contains this XA-compliant Type 2 JDBC driver built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R1, or earlier releases.

DB2 UDB for iSeries (Native XA - V5R1 and earlier) supports two phase data source:

com.ibm.db2.jdbc.app.DB2StdXADataSource

Requires JDBC driver files: **db2_classes.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

Does not require an authentication alias.

Requires properties:

- **databaseName** The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

For more information on DB2 UDB for iSeries, visit the DB2 Web site at:
<http://www.ibm.com/software/data/db2/>

DB2 on z/OS, connecting to Application Server on Windows, UNIX, or workstation-based LINUX

1. DB2 Universal JDBC Provider

The DB2 Universal JDBC Driver is an architecture-neutral JDBC driver for distributed and local DB2 access. Because the Universal Driver architecture is independent of any particular JDBC driver connectivity or target platform, it allows both Java connectivity (Type 4) or Java Native Interface (JNI) based connectivity (Type 2) in a single driver instance to DB2. Starting with WebSphere Application Server Version 5.0.2, the product now supports both Type 2 and Type 4 JDBC drivers. To use the Type 4 driver, you must install DB2 Version 8.1 or a later version. To use the Type 2 driver, you must install DB2 Version 8.1 Fix Pack 2 or a later version.

This JDBC driver allows applications to use both JDBC and Structured Query Language in Java (SQLJ) access.

The DB2 Universal JDBC Driver Provider supports one phase data source:

com.ibm.db2.jcc.DB2ConnectionPoolDataSource

Requires JDBC driver files:

- **db2jcc.jar** After you install DB2, you can find this jar file in the DB2 *java* directory. For Type 4 JDBC driver support from a client machine where DB2 is not installed, copy this file to the local machine. If you install any fixes or upgrades to DB2, you must update this file as well. You must also set the **DB2UNIVERSAL_JDBC_DRIVER_PATH** path variable to point to the **db2jcc.jar** file. See the Cloudscape section for more information on the **DB2UNIVERSAL_JDBC_DRIVER_PATH** path variable.

Note: To find out the version of the universal driver you are using, issue this DB2 command:

```
java com.ibm.db2.jcc.DB2Jcc -version
```

The output for the above example is:

```
IBM DB2 JDBC Universal Driver Architecture 2.2.xx
```

- **db2jcc_license_cu.jar** This is the DB2 Universal JDBC driver license file that allows access to the DB2 Universal database. Use this jar file or the next one to gain access to the database. This jar file ships with WebSphere Application Server in a directory defined by **\${UNIVERSAL_JDBC_DRIVER_PATH}** environment variable.
- **db2jcc_license_cisuz.jar** This is the DB2 Universal JDBC driver license file that allows access to the following databases:
 - DB2 Universal
 - DB2 for iSeries
 - DB2 for z/OS
 - SQLDS

The **db2jcc_license_cisuz.jar** does not ship with Websphere Application Server and should be located in the same directory as the **db2jcc.jar** file, so that the **DB2UNIVERSAL_JDBC_DRIVER_PATH** points to both.

The classpath for this provider is set as follows:

```
<classpath>${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar </classpath>
<classpath>${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu.jar</classpath>
<classpath>${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cisuz.jar</classpath>
```

Note: The license jar files are independent of each other; therefore, order does not matter.

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2UniversalDataSourceHelper`

Requires a valid authentication alias.

Requires properties:

- **databaseName** This is an actual database name if the **driverType** is set to **4**, or a locally cataloged database name if the **driverType** is set to **2**.
- **driverType** The JDBC connectivity type of a data source. *There are two permitted values: 2 and 4.* If you want to use Universal JDBC Type 2 driver, set this value to 2. If you want to use Universal JDBC Type 4 driver, set this value to 4.
- **serverName** The TCP/IP address or host name for the Distributed Relational Database Architecture (DRDA) server. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.
- **portNumber** The TCP/IP port number where the DRDA server resides. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.

2. DB2 Universal JDBC Provider (XA)

The DB2 Universal JDBC Driver (XA) is an architecture-neutral JDBC driver for distributed and local DB2 access. In WebSphere Application Server Version 5.0.2, this driver only supports Java Native Interface (JNI) based connectivity (Type 2) in a single driver instance to DB2. To use this driver, you must install DB2 Version 8.1 Fix Pack 2 or a later version. This driver supports two phase transactions and the WebSphere Application Server Version 5.0 data source. This driver allows applications to use both JDBC and SQLJ access.

The DB2 Universal JDBC Driver Provider supports the two phase data source:

`com.ibm.db2.jcc.DB2XADataSource`

Requires JDBC driver files:

- **db2jcc.jar** After you install DB2, you can find this .jar file in the DB2 java directory. For Type 4 JDBC driver support from a client machine where DB2 is not installed, copy this file to the local machine. If you install any fixes or upgrades to DB2, you must update this file as well. You must also set the **DB2UNIVERSAL_JDBC_DRIVER_PATH** environment variable to point to the `db2jcc.jar` file. See the Cloudscape section for more information on the **DB2UNIVERSAL_JDBC_DRIVER_PATH** environment variable.

Note: To find the level of universal driver you are using, issue the following DB2 command:

```
java com.ibm.db2.jcc.DB2Jcc -version
```

example output of the above:

```
IBM DB2 JDBC Universal Driver Architecture 2.2.xx
```

- **db2jcc_license_cu.jar** This is the DB2 Universal JDBC driver license file that allows access to the DB2 Universal database. Use this jar file or the next one to gain access to the database. This jar file ships with WebSphere Application Server in the `WAS_HOME/universalDriver/lib` directory.
- **db2jcc_license_cisuz.jar** This is the DB2 Universal JDBC driver license file that allows access to the following databases:
 - DB2 Universal
 - DB2 for iSeries
 - DB2 for z/OS
 - SQLDS

You must use the right license jar file to access a specific database backend.

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2UniversalDataSourceHelper`

Requires a valid authentication alias.

Requires properties:

- **databaseName** This is a locally cataloged database name.
- **driverType** This is the JDBC connectivity type of a data source. *If you are running a version of DB2 prior to DB2 V8.1 FP6, you are restricted to using only the type 2 driver.*

- **serverName** The TCP/IP address or host name for the Distributed Relational Database Architecture (DRDA) server. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.
- **portNumber** The TCP/IP port number where the DRDA server resides. Provide a value for this property only if your **driverType** is set to **4**. This property is not required if your **driverType** is set to **2**.

3. DB2 legacy CLI-based Type 2 JDBC Driver

The DB2 legacy CLI-based Type 2 JDBC Driver Provider is built on top of DB2 CLI (Call Level Interface). It uses the DB2 CLI interface to communicate with DB2 UDB servers. For use with Application Server on Windows, UNIX, or workstation-based LINUX, this provider requires DB2 Connect (which is available from DB2).

DB2 legacy CLI-based Type 2 JDBC Driver supports one phase data source:

COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource

Requires JDBC driver files: **db2java.zip** (Note: If you run SQLJ in DB2 Version 8, **db2jcc.jar** is also required.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2DataStoreHelper`

Does not require a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

4. DB2 legacy CLI-based Type 2 JDBC Driver (XA)

The DB2 legacy CLI-based Type 2 JDBC Driver (XA) is built on top of DB2 CLI (Call Level Interface). It uses the DB2 CLI interface to communicate with DB2 UDB servers. For use with Application Server on Windows, UNIX, or workstation-based LINUX, this provider requires DB2 Connect (which is available from DB2).

DB2 legacy CLI-based Type 2 JDBC Driver (XA) supports two phase data source:

COM.ibm.db2.jdbc.DB2XADataSource

Requires JDBC driver files: **db2java.zip** (Note: If you run SQLJ in DB2 Version 8, **db2jcc.jar** is also required.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.DB2DataStoreHelper`

Does not require a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

For more information on DB2 for z/OS, visit the DB2 Web site at: <http://www.ibm.com/software/data/db2/>.

Cloudscape

1. Cloudscape JDBC Provider

The Cloudscape JDBC Provider provides the JDBC access to the Cloudscape database. This Cloudscape JDBC driver used the embedded framework. You cannot use any Version 4.0 data sources with Cloudscape.

Cloudscape JDBC Provider supports one phase data source:

com.ibm.db2j.jdbc.DB2jConnectionPoolDataSource

Requires JDBC driver files: **db2j.jar**.

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.CloudscapeDataStoreHelper`

Does not require a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections. If you do not specify a fully qualified path name, Application Server uses the default location of WAS_HOME./cloudscape (or the equivalent default for a UNIX or LINUX environment).
 - Example database path name for Windows: `c:\temp\sampleDB`
 - Example database path name for UNIX or LINUX: `/tmp/sampleDB`

If no database currently exists for the path name you want to specify, simply append `;create=true` to the path name to create a database dynamically. (For example: `c:\temp\sampleDB;create=true`)

2. Cloudscape JDBC Provider (XA)

The Cloudscape JDBC Provider (XA) provides the XA-compliant JDBC access to the Cloudscape database. This Cloudscape JDBC driver uses the embedded framework. You cannot use any Version 4.0 data sources with Cloudscape.

Cloudscape JDBC Provider (XA) supports two phase data source:

`com.ibm.db2j.jdbc.DB2jXADataSource`

Requires JDBC driver files: **db2j.jar**

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.CloudscapeDataStoreHelper`

Does not require a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections. If you do not specify a fully qualified path name, Application Server uses the default location of WAS_HOME./cloudscape (or the equivalent default for a UNIX or LINUX environment).
 - Example database path name for Windows: `c:\temp\sampleDB`
 - Example database path name for UNIX or LINUX: `/tmp/sampleDB`

If no database currently exists for the path name you want to specify, simply append `;create=true` to the path name to create a database dynamically. (For example: `c:\temp\sampleDB;create=true`)

3. Cloudscape Network Server using Universal JDBC driver

This Cloudscape driver takes advantage of the Network Server support that the DB2 universal Type 4 JDBC driver provides. You cannot use any Version 4.0 data sources with Cloudscape.

Cloudscape uses the DB2 Universal Driver when using the Network Server. It supports one phase data source:

`com.ibm.db2.jcc.DB2ConnectionPoolDataSource`

Requires JDBC driver files:

- **db2jcc.jar** If you install and run DB2, you must use the **db2jcc.jar** file that comes with DB2. To do that, the classpath in the JDBC template for Cloudscape network server is set to be:

```
<classpath>${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar</classpath>
```

```
<classpath>${CLOUDSCAPE_JDBC_DRIVER_PATH}/db2j.jar</classpath>
```

```
<classpath>${CLOUDSCAPE51_JDBC_DRIVER_PATH}/db2j.jar</classpath>
```

```
<classpath>${UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar</classpath>
```

which means that the `db2jcc.jar` from DB2 always takes precedence. Note that this also means that you must set the DB2 environment variable **DB2UNIVERSAL_JDBC_DRIVER_PATH** in WebSphere when you set up your DB2 datasources. This is instead of hard coding the path of the `db2jcc.jar` for DB2 datasources.

- **db2jcc_license_cu.jar** This file is the DB2 Universal JDBC license file that provides access to the Cloudscape databases using the **Network Server** framework. Use this file to gain access to the database. This file ships with WebSphere and is located in `${UNIVERSAL_JDBC_DRIVER_PATH}`.

Note: **UNIVERSAL_JDBC_DRIVER_PATH** is a WebSphere environment variable that is already defined to the location in WebSphere Application Server where the license jar file above is

located, and will only be used if the **DB2UNIVERSAL_JDBC_DRIVER_PATH** is not set. DB2 users should ensure that **DB2UNIVERSAL_JDBC_DRIVER_PATH** is set to avoid loading multiple versions of the db2jcc.jar file.

Note: **DB2UNIVERSAL_JDBC_DRIVER_PATH** is a WebSphere environment variable that you must set to point to the location of db2jcc.jar file (that comes with DB2). This variable is set only if you create a db2 provider. See the DB2 section for more information on the **DB2UNIVERSAL_JDBC_DRIVER_PATH** path variable.

Note: Cloudscape requires only db2jcc_license_c.jar; however, WebSphere Application Server uses db2jcc_license_cu.jar because this works for both DB2 UDB and Cloudscape.

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.CloudscapeNetworkServerDataStoreHelper`

Note: The administrative console incorrectly lists the DB2UniversalDataStoreHelper as the default value for the **DataStoreHelper** class. You must change the default value to `com.ibm.websphere.rsadapter.CloudscapeNetworkServerDataStoreHelper`. Also change the custom properties, using the instructions in the customer property section.

Requires a valid authentication alias.

Requires properties:

- **databaseName** The name of the database from which the data source obtains connections. If you do not specify a fully qualified path name, Application Server uses the default location of `WAS_HOME./cloudscape` (or the equivalent default for a UNIX or LINUX environment).
 - Example database path name for Windows: `c:\temp\sampleDB`
 - Example database path name for UNIX or LINUX: `/tmp/sampleDB`

If no database currently exists for the path name you want to specify, simply append `;create=true` to the path name to create a database dynamically. (For example: `c:\temp\sampleDB;create=true`)

- **driverType** Only the Type 4 driver is allowed.
- **serverName** The TCP/IP address or the host name for the Distributed Relational Database Architecture (DRDA) server.
- **portNumber** The TCP/IP port number where the DRDA server resides. The default value is port **1527**.
- **retrieveMessagesfromServerOnGetMessage** This property is required by WebSphere Application Server, not the database. The default value is **false**. You must set the value of this property to **true**, to enable text retrieval using the `SQLException.getMessage()` method.

See the Cloudscape setup instructions for more information on configuring the Cloudscape Network Server.

For more information on IBM Cloudscape, visit the Cloudscape Web site at:
<http://www.ibm.com/software/data/cloudscape/>

Informix

1. Informix JDBC Driver

The Informix JDBC Driver is a Type 4 JDBC driver that provides JDBC access to the Informix database.

Informix JDBC Driver supports one phase data source:

`com.informix.jdbcx.IfxConnectionPoolDataSource`

Requires JDBC driver files:

`ifxjdbc.jar`
`ifxjdbcx.jar`

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.InformixDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the Informix instance on the server. Example: *ol_myserver*.
- **portNumber** The port on which the instances listen. Example: *1526*.
- **ifxIFXHOST** Either the IP address or the host name of the machine that is running the Informix database to which you want to connect. Example: *myserver.mydomain.com*.
- **databaseName** The name of the database from which the data source obtains connections. Example: *Sample*.
- **informixLockModeWait** Although not required, this property enables you to set the number of seconds that Informix software waits for a lock. By default, Informix code throws an exception if it cannot immediately acquire a lock. Example: *2*.

2. Informix JDBC Driver (XA)

The Informix JDBC Driver (XA) is a Type 4 JDBC driver that provides XA-compliant JDBC access to the Informix database.

Informix JDBC Driver (XA) supports two phase data source:

`com.informix.jdbcx.IfXXADataSource`

Requires JDBC driver files:

`ifxjdbc.jar`
`ifxjdbcx.jar`

Requires `DataStoreHelper` class:

`com.ibm.websphere.rsadapter.InformixDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the Informix instance on the server. Example: *ol_myserver*.
- **portNumber** The port on which the instances listen. Example: *1526*.
- **ifxIFXHOST** Either the IP address or the host name of the machine that is running the Informix database to which you want to connect. Example: *myserver.mydomain.com*.
- **databaseName** The name of the database from which the data source obtains connections. Example: *Sample*.
- **informixLockModeWait** Although not required, this property enables you to set the number of seconds that Informix software waits for a lock. By default, Informix code throws an exception if it cannot immediately acquire a lock. Example: *2*.

For more information on Informix, visit the Informix Web site at: <http://www.ibm.com/software/data/informix/>

Sybase

1. Sybase JDBC Driver

The Sybase JDBC Driver is a Type 4 JDBC driver that provides JDBC access to the Sybase database.

Sybase JDBC Driver supports one phase data source:

`com.sybase.jdbc2.jdbc.SybConnectionPoolDataSource`

Requires JDBC driver files: `jconn2.jar`.

Requires `DataStoreHelper` class:

`com.ibm.websphere.rsadapter.SybaseDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the database server. Example: *myserver.mydomain.com*.
- **databaseName** The name of the database from which the data source obtains connections. Example: *Sample*.
- **portNumber** The TCP/IP port number through which all communications to the server take place. Example: *4100*.
- **connectionProperties** A custom property required for applications containing EJB 2.0 enterprise beans. Value: `SELECT_OPEN_CURSOR=true`(Type: `java.lang.String`)

2. Sybase JDBC Driver (XA)

The Sybase JDBC Driver (XA) is a Type 4 JDBC driver that provides XA-compliant JDBC access to the Sybase database.

Sybase JDBC Driver (XA) supports two phase data source:

`com.sybase.jdbc2.jdbc.SybXADataSource`

Requires JDBC driver files: **jconn2.jar**.

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.SybaseDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the database server. Example: *myserver.mydomain.com*
- **databaseName** The name of the database from which the data source obtains connections. Example: *Sample*.
- **portNumber** The TCP/IP port number through which all communications to the server take place. Example: *4100*.
- **connectionProperties** A custom property required for applications containing EJB 2.0 enterprise beans. Value: `SELECT_OPEN_CURSOR=true`(Type: `java.lang.String`)

For more information on Sybase, visit the Sybase Web site at: <http://www.sybase.com/>

Oracle

1. Oracle JDBC Driver

The Oracle JDBC Driver provides JDBC access to the Oracle database. This JDBC driver supports both Type 2 JDBC access and Type 4 JDBC access.

Oracle JDBC Driver supports one phase data source:

`oracle.jdbc.pool.OracleConnectionPoolDataSource`

Requires JDBC driver files: **ojdbc14.jar**. (Note: If you require Oracle trace, use **ojdbc14_g.jar**.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.OracleDataStoreHelper`

(Note: If you are running Oracle10g, use `com.ibm.websphere.rsadapter.Oracle10gDataStoreHelper`.)

Requires a valid authentication alias.

Requires properties:

- **URL** The URL that indicates the database from which the data source obtains connections. Example: `jdbc:oracle:thin:@myServer:1521:myDatabase`, where *myServer* is the server name, *1521* is the port it is using for communication, and *myDatabase* is the database name.

2. Oracle JDBC Driver (XA)

The Oracle JDBC Driver (XA) provides XA-compliant JDBC access to the Oracle database. This JDBC driver supports both Type 2 JDBC access and Type 4 JDBC access.

Oracle JDBC Driver (XA) supports two phase data source:

`oracle.jdbc.xa.client.OracleXADataSource`

Requires JDBC driver files: **ojdbc14.jar**. (Note: If you require Oracle trace, use **ojdbc14_g.jar**.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.OracleDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **URL** The URL that indicates the database from which the data source obtains connections. Example: `jdbc:oracle:thin:@myServer:1521:myDatabase`, where *myServer* is the server name, *1521* is the port it is using for communication, and *myDatabase* is the database name.

For more information on Oracle, visit the Oracle Web site at: <http://www.oracle.com/>

MS SQL Server

1. DataDirect ConnectJDBC type 4 driver for MS SQL Server

DataDirect ConnectJDBC type 4 driver for MS SQL Server is a Type 4 JDBC driver that provides JDBC access to the MS SQL Server database. This provider is for use only with the Connect JDBC driver purchased from DataDirect Technologies.

This JDBC provider supports this data source:

`com.ddtek.jdbcx.sqlserver.SQLServerDataSource`

Requires JDBC driver files:

`sqlserver.jar`,
`base.jar` and `util.jar`

(The `spy.jar` file is optional. You need this file to enable spy logging. The `spy.jar` file is not in the same directory as the other three jar files. Instead, it is located in the `../spy/` directory.)

Requires `DataStoreHelper` class:

`com.ibm.websphere.rsadapter.ConnectJDBCDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which MS SQL Server resides. Example:
`myserver.mydomain.com`
- **portNumber** The TCP/IP port that MS SQL Server uses for communication. Port 1433 is the default.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

2. DataDirect ConnectJDBC type 4 driver for MS SQL Server (XA)

DataDirect ConnectJDBC type 4 driver for MS SQL Server (XA) is a Type 4 JDBC driver which provides XA-compliant JDBC access to the MS SQL Server database. This provider is for use only with the Connect JDBC driver purchased from DataDirect Technologies.

This JDBC provider supports this data source:

`com.ddtek.jdbcx.sqlserver.SQLServerDataSource`.

Requires JDBC driver files:

`sqlserver.jar`,
`base.jar` and `util.jar`.

(The `spy.jar` file is optional. You need this file to enable spy logging. The `spy.jar` file is not in the same directory as the other three jar files. Instead, it is located in the `../spy/` directory.)

Requires `DataStoreHelper` class:

`com.ibm.websphere.rsadapter.ConnectJDBCDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which MS SQL Server resides. Example:
`myserver.mydomain.com`
- **portNumber** The TCP/IP port that MS SQL Server uses for communication. Port 1433 is the default.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

For more information on the DataDirect ConnectJDBC driver, visit the DataDirect Web site at:

<http://www.datadirect-technologies.com/>

3. WebSphere embedded ConnectJDBC driver for MS SQL Server

WebSphere embedded ConnectJDBC driver for MS SQL Server is a Type 4 JDBC driver that provides JDBC access to the MS SQL Server database. This JDBC driver ships with WebSphere Application Server. Only use this provider with the Connect JDBC driver embedded in WebSphere; it cannot be used with a Connect JDBC driver purchased separately from DataDirect Technologies.

This JDBC provider supports this data source:

`com.ibm.websphere.jdbcx.sqlserver.SQLServerDataSource`.

Requires JDBC driver files:

sqlserver.jar
base.jar and
util.jar.

(The **spy.jar** file is optional. You need this file to enable spy logging. The **spy.jar** file for the WebSphere embedded Connect JDBC driver ships with WebSphere Application Server. All the files are located in the `WAS_HOME/lib/` directory.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.WSConnectJDBCDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which MS SQL Server resides. Example:
myserver.mydomain.com
- **portNumber** The TCP/IP port that MS SQL Server uses for communication. Port 1433 is the default.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

4. WebSphere embedded ConnectJDBC driver for MS SQL Server (XA)

WebSphere embedded ConnectJDBC driver for MS SQL Server (XA) is a Type 4 JDBC driver which provides XA-compliant JDBC access to the MS SQL Server database. This JDBC driver ships with WebSphere Application Server. Use this provider with the Connect JDBC driver embedded in WebSphere. Do not use it with a Connect JDBC driver purchased separately from DataDirect Technologies.

This JDBC provider supports this data source:

`com.ibm.websphere.jdbcx.sqlserver.SQLServerDataSource`.

Requires JDBC driver files:

sqlserver.jar
base.jar and
util.jar.

(The **spy.jar** file is optional. You need this file to enable spy logging. The **spy.jar** file for the WebSphere embedded Connect JDBC driver ships with WebSphere Application Server. All the files are located in the `WAS_HOME/lib/` directory.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.WSConnectJDBCDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which MS SQL Server resides. Example:
myserver.mydomain.com
- **portNumber** The TCP/IP port that MS SQL Server uses for communication. Port 1433 is the default.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

Whether you need to support one or two phase transactions with the WebSphere embedded Connect JDBC XA driver, you must install Stored Procedures for the Java Transaction API (JTA) on your machine that runs Microsoft SQL. The WebSphere Application Server installation disks provide a base level of Stored Procedures for JTA. (You can find the most current version of the software on the WebSphere Application Server-embedded DataDirect Technologies product update Web page.) Install Stored Procedures for JTA by performing the following steps:

- Determine whether you are running the 32-bit or 64-bit MS SQL Server and select the appropriate `sqljdbc.dll` and `instjdbc.sql` files.
- Stop your MS SQL Server service.
- Copy the `sqljdbc.dll` file into your `%SQL_SERVER_INSTALL%\Binn\` directory.
- Restart the MS SQL Server service.
- Run the `instjdbc.sql` script. (The script can be run by the MS SQL Server Query Analyzer or the ISQL utility).

You can download the latest patches and upgrades to the WebSphere embedded Connect JDBC driver from the following FTP site:

<ftp://ftp.software.ibm.com/software/websphere/info/tools/DataDirect/datadirect.htm>

5. **DataDirect SequeLink type 3 JDBC driver for MS SQL Server -- Deprecated**

Because this JDBC provider is deprecated in WebSphere Application Server Version 6.0, it is no longer an available option in the administrative console. In its place, use one of the Connect JDBC providers, which are described previously in this section.

DataDirect SequeLink type 3 JDBC driver for MS SQL Server is a type 3 JDBC driver that provides JDBC access to MS SQL Server via SequeLink server.

This JDBC provider supports this data source:

`com.ddtek.jdbcx.sequelink.SequeLinkDataSource`

Requires JDBC driver files:

`s1jc.jar` and
`spy-s1.jar`

(The JDBC driver shipped with WebSphere Application Server requires the `s1jc.jar` and the `spy-s1.jar` files. The JDBC driver purchased from DataDirect requires the `s1jc.jar` and the `spy.jar` files. The `spy.jar` and `spy-s1.jar` files are optional. You need these files to enable spy logging.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.SequeLinkDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which SequeLink Server resides. Example:
myserver.mydomain.com
- **portNumber** The TCP/IP port that SequeLink Server uses for communication. By default, SequeLink Server uses port 19996.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

6. **DataDirect SequeLink type 3 JDBC driver for MS SQL Server (XA) -- Deprecated**

Because this JDBC provider is deprecated in WebSphere Application Server Version 6.0, it is no longer an available option in the administrative console. In its place, use one of the Connect JDBC providers, which are described previously in this section.

DataDirect SequeLink type 3 JDBC driver for MS SQL Server (XA) is a type 3 JDBC driver that provides XA-compliant JDBC access to MS SQL Server via the SequeLink server.

This JDBC provider supports this data source:

`com.ddtek.jdbcx.sequelink.SequeLinkDataSource`

Requires JDBC driver files:

`s1jc.jar` and
`spy-s1.jar`

(The JDBC driver shipped with WebSphere Application Server requires the `s1jc.jar` and the `spy-s1.jar` files. The JDBC driver purchased from DataDirect requires the `s1jc.jar` and the `spy.jar` files. The `spy.jar` and `spy-s1.jar` files are optional. You need these files to enable spy logging.)

Requires **DataStoreHelper** class:

`com.ibm.websphere.rsadapter.SequeLinkDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which SequeLink Server resides. Example:
myserver.mydomain.com
- **portNumber** The TCP/IP port that SequeLink Server uses for communication. By default, SequeLink Server uses port 19996.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

Both of the WebSphere-embedded SequeLink JDBC drivers require installation of SequeLink Server on all machines running MS SQL Server. See the readme.html file found in the DataDirect folder on the WebSphere Application Server CD for instructions on how to install SequeLink Server. (Only install SequeLink Server from the WebSphere Application Server CD if you are using the SequeLink JDBC driver embedded in WebSphere. Otherwise, install a copy of SequeLink Server purchased from DataDirect Technologies.)

From the following FTP site, you can download the latest patches and upgrades for the version of SequeLink Server that is used with the WebSphere-embedded SequeLink JDBC driver:

<ftp://ftp.software.ibm.com/software/websphere/info/tools/DataDirect/datadirect.htm>

For more information on the DataDirect SequeLink type 3 JDBC driver, visit the DataDirect Web site at:

<http://www.datadirect-technologies.com/>

7. **Microsoft JDBC driver for MSSQLServer 2000** -- Deprecated

Because this JDBC provider is deprecated in WebSphere Application Server Version 6.0, it is no longer an available option in the administrative console. In its place, use one of the Connect JDBC providers, which are described previously in this section.

Microsoft JDBC driver for MSSQLServer 2000 is a type 4 JDBC driver that provides JDBC access to the MS SQL Server database.

This JDBC provider supports this data source:

`com.microsoft.jdbcx.sqlserver.SQLServerDataSource`

Requires JDBC driver files:

`mssqlserver.jar`,
`msbase.jar` and `msutil.jar`

(The `spy.jar` file is optional. You need it to enable spy logging. However, Microsoft does not ship the `spy.jar` file. Contact Microsoft about this issue.)

Requires `DataStoreHelper` class:

`com.ibm.websphere.rsadapter.ConnectJDBCDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which MS SQL Server resides. Example:
myserver.mydomain.com
- **portNumber** The TCP/IP port that MS SQL Server uses for communication. Port 1433 is the default.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

8. **Microsoft JDBC driver for MSSQLServer 2000 (XA)** -- Deprecated

Because this JDBC provider is deprecated in WebSphere Application Server Version 6.0, it is no longer an available option in the administrative console. In its place, use one of the Connect JDBC providers, which are described previously in this section.

Microsoft JDBC driver for MSSQLServer 2000 (XA) is a type 4 JDBC driver that provides XA-compliant JDBC access to the MS SQL Server database.

This JDBC provider supports this data source:

`com.microsoft.jdbcx.sqlserver.SQLServerDataSource`

Requires JDBC driver files:

`mssqlserver.jar`,
`msbase.jar` and `msutil.jar`

(The `spy.jar` file is optional. You need it to enable spy logging. However, Microsoft does not ship the `spy.jar` file. Contact Microsoft about this issue.)

Requires `DataStoreHelper` class:

`com.ibm.websphere.rsadapter.ConnectJDBCDataStoreHelper`

Requires a valid authentication alias.

Requires properties:

- **serverName** The name of the server in which MS SQL Server resides. Example:
myserver.mydomain.com
- **portNumber** The TCP/IP port that MS SQL Server uses for communication. Port 1433 is the default.
- **databaseName** The name of the database from which the data source obtains connections.
Example: *Sample*.

For more information on the Microsoft JDBC driver, visit the Microsoft Web site at:

<http://www.microsoft.com/sql>

Connector modules collection

Use this page to view established connector modules, which are resource adaptor (RAR) files that have been packaged into deployable components compliant with the J2EE Connector Architecture (JCA).

To view this administrative console page, click **Applications >Enterprise Applications > application > Connector Modules**.

You must generate a connector module for every resource adapter (RAR file) in the application. You do this through an assembly tool, which creates an instance of the connector module object for the RAR file. To learn more about the process, see the topic "Assembling resource adapter (connector) modules" in the Information Center.

Remove: Removes a module from the deployed application. The module is deleted from the application in the WebSphere Application Server configuration repository and also from all 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 then 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 then the application is restarted.

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 then the application is restarted.

URI:

Specifies the logical path to the resource that will be serviced by the product.

Name:

Specifies the display name of the connector module.

Connector module settings:

Use this page to view the settings of connector modules, which are resource adaptor (RAR) files that have been packaged into deployable components compliant with the J2EE Connector Architecture (JCA).

To view this administrative console page, click **Applications > Enterprise Applications > application > Connector Modules > connector_module**.

The following field descriptions refer to properties that are set when you create connector modules using an assembly tool. To learn more about the process, see the topic "Assembling resource adapter (connector) modules" in the Information Center.

Remove: Removes a module from the deployed application. The module is deleted from the application in the WebSphere Application Server configuration repository and also from all 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 then 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 then the application is restarted.

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 then the application is restarted.

Uri:

Specifies the logical path to the resource that is serviced by WebSphere Application Server.

Data type String

Name:

Specifies the display name of the connector module.

Data type String

altDD:

Specifies the alternate DD of the connector module.

The alternate DD URI for a given module.

Data type String

Starting weight:

Specifies the startup priority of the connector module over others.

When your application contains multiple modules, the starting weight you specify determines this module's startup priority over other modules during server startup. Modules with lower startup order are started first.

Data type String

Messaging resources

Learn about messaging resources

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Enable applications to use JMS resources and message-driven beans

- Develop applications that use asynchronous messaging (refer to "Using asynchronous messaging" in the information center)
- Install and configure a messaging provider
- Use JMS resources of WebSphere MQ
- Use JMS resources of a generic provider
- Set up JMS resources
- Set up JMS resources for service integration bus

Develop programs that use JMS and messaging directly

- Design an application that uses JMS
- Develop an application that uses JMS
- Develop a JMS client
- Assemble applications for deployment (same as any application type)

Develop programs that use message-driven beans

- Design an application that uses message-driven beans
- Develop an application that uses message-driven beans
- Deploy an application that uses message-driven beans as JCA 1.5-compliant resources

Deploy and administer applications that use JMS resources

- Deploy applications (same as any application type)
- Deploy applications (Education on Demand)
- Administer JMS providers and the messaging resources they provide
- Administer applications (same as any application type)
- Administer applications (Education on Demand)
- Adding security for JMS resources (Education on Demand)

Troubleshoot messaging

Refer to the *Troubleshooting and support* PDF.

Conceptual overviews

Documentation

Refer to the article *Introduction: Messaging resources* in the information center.

Presentations

Education on Demand offers:

- IBM service integration technologies overview
- IBM service integration technologies architecture
- Workload management and high availability of IBM service integration technologies
- Mediation
- Managing service integration bus resources
- JMS messaging support
- Managing JMS resources for service integration bus

See Chapter 11 of the IBM Redbook IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

developerWorks offers:

- Tutorial 3 - Message driven Timer

This tutorial makes full use of the MyBank sample codes. The sample consists of 2 entity beans, CustomerBean and AccountBean, whose abstract schema types are Customer and Account, respectively. Each entity bean has remote/local interfaces and remote/local home interfaces. The entity bean CustomerBean has a one-to-many relationship with AccountBean. The SenderBean.java session bean is responsible for sending the message to the destination, and the MDB MyBankListenerBean.java is the consumer for the message. The zip file comes with all sample code required to run this tutorial

Samples

The Samples Gallery offers:

- **WebSphere Bank**

Using the WebSphere Bank online bank, customers can open accounts, get account balances, and transfer funds between accounts. The WebSphere Bank application uses Web services, Java Message Service (JMS) API, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, Message-Driven Beans (MDB), JSP pages, and servlets

- **Greenhouse by WebSphere**

Using the Greenhouse by WebSphere online supplier, customers can open accounts, select items and amounts to order, and check their order status. The Greenhouse by WebSphere application uses Web services, the Java message service (JMS) API, scheduler, asynchronous beans, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, message-driven beans (MDB), Java server pages (JSP) files, and the struts framework.

- **MDB Sample**

The Message-Driven Bean (MDB) Sample consists of an Application client and two message driven beans. The application client sends a message to a queue or a topic, each of which has a message driven beans acting as a JMS message listener. The message includes a temporary reply-to queue. The message driven bean listening to the queue or topic receives the request message and sends a response to the reply-to destination taken from the request. The client then receives the response from the reply-to queue.

- **Asynchronous beans - WebSphere Trader**

This Sample illustrates how to implement a streaming stock ticker server and client using asynchronous beans and J2EE services such as:

- Servlets
- Java Message Service (JMS)
- Session enterprise beans
- Container-managed persistence (CMP) 2.0 enterprise beans
- Message-driven beans (MDB)

This Sample uses several parts to maximize the utilization of a server:

- Work - Runs J2EE context-aware code on a thread.
- Alarm - Runs J2EE context-aware code at a given time interval.
- EventSource - A method of broadcasting events to registered listeners.
- SubsystemMonitor - A thread that monitors the status of any asynchronous system and uses an EventSource method to inform registered listeners of the system status.
- WorkManager - Thread configuration and J2EE context policies that are used by various asynchronous beans parts.
- AsynchScope - A collection of alarms, subsystem monitors and other asynchronous scopes that support relationships. This collection utilizes a single WorkManager thread and is also an event source.
- Startup Bean - A specialized, stateful session enterprise bean that supports bootstrapping asynchronous work when the application starts.

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.

Besides using the programming interfaces directly to explicitly poll for messages, WebSphere Application Server also supports the use of message-driven beans 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 listen on, without an application having to explicitly poll the destination.

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 for that purpose. In the JCA 1.5 specification, such message-driven beans are commonly called message endpoints or simply endpoints.

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 JMS provider that has a JCA 1.5 resource adapter, such as the default messaging provider that is part of WebSphere Application Server version 6.

With a JCA 1.5 resource adapter, you deploy EJB 2.1 message-driven beans as JCA resources to use a J2C activation specification. If a 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).

You can use the WebSphere administrative console to administer the WebSphere Application Server support for asynchronous messaging. For example, you can configure JCA resource adapters, J2C activation specifications, JMS providers, and JMS resources, and can control the activity of messaging services.

To learn more about WebSphere messaging support, see the following topics:

- JMS providers
- Styles of messaging in applications
- Using JMS interfaces to explicitly poll for messages
- Using message-driven beans to automatically retrieve messages
- "Components of JMS support" in the information center.
- Components of message-driven bean support
- Security considerations for asynchronous messaging

JMS providers

This topic provides an overview of the support for JMS providers by 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 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.

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>).

Using JMS interfaces to explicitly poll 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.

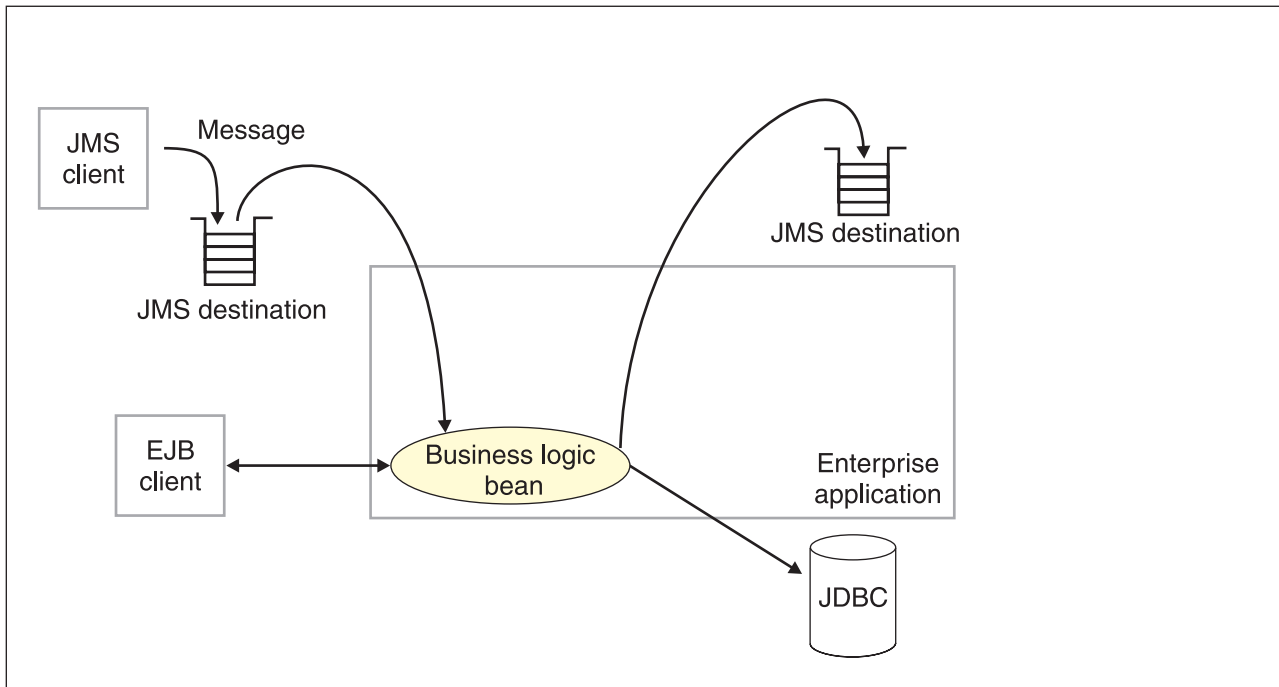


Figure 6. Asynchronous messaging using JMS. This figure shows an enterprise application polling a JMS destination to retrieve an incoming message, which it processes with a business logic bean. The business logic bean uses standard JMS calls to process the message; for example, to extract data or to 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.

Using message-driven beans to automatically retrieve messages

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.TimerObject 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).

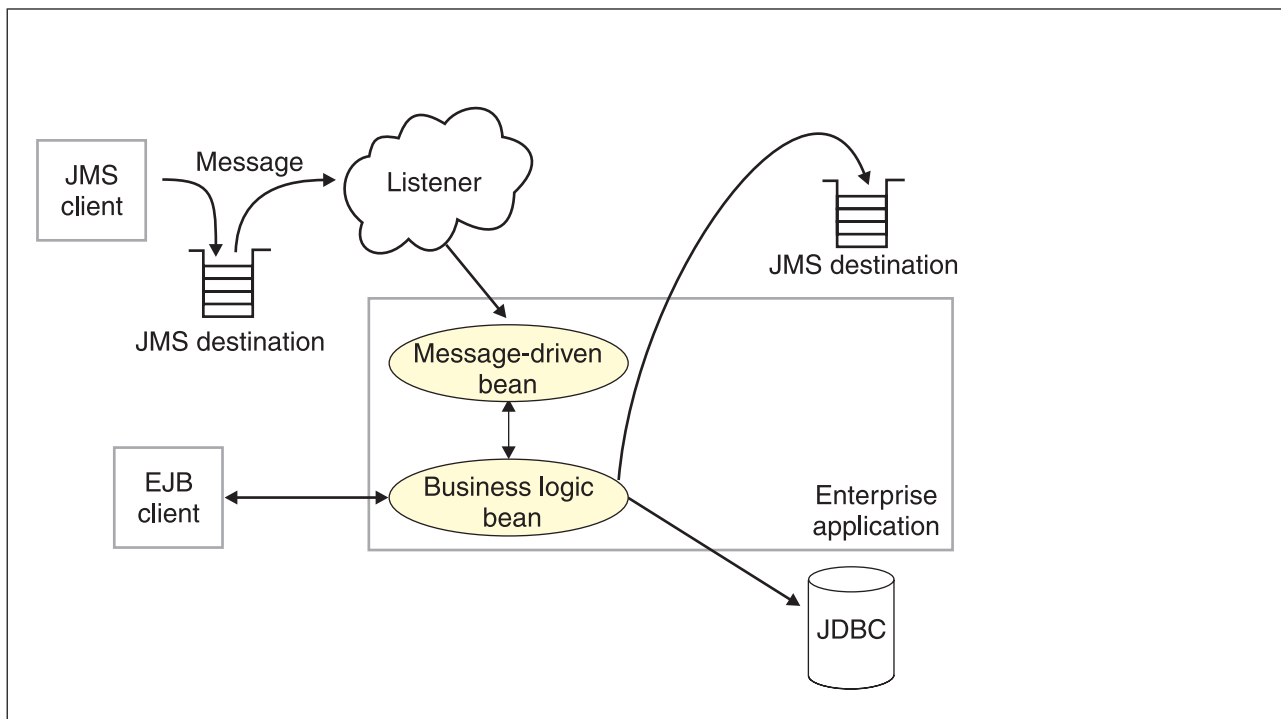


Figure 7. Messaging with message-driven beans. This figure shows an incoming message being passed automatically to the onMessage() method of a message-driven bean that is deployed as a listener for the destination. The message-driven bean processes the message, in this case passing the message on to 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 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 endpoints must specify the resource adapter that sends messages to the endpoint.

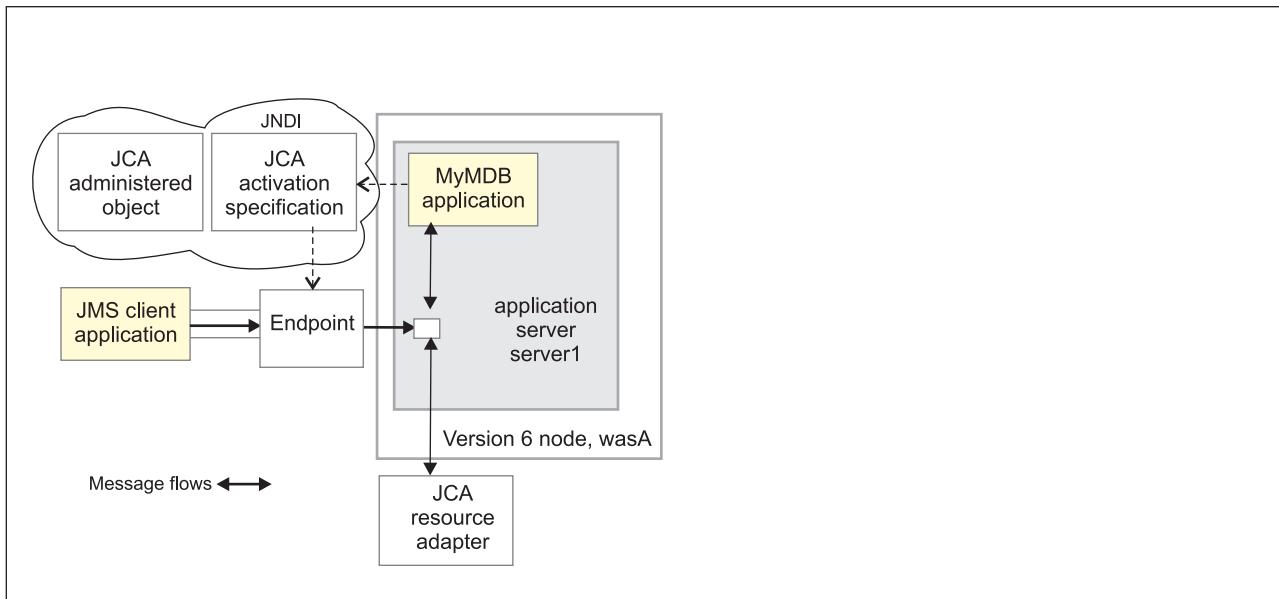


Figure 8. Message-driven bean components for a JCA resource adapter. This figure shows the main components of WebSphere support for message-driven beans for use with an external JCA resource adapter.

The administrator creates a J2C activation specification 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.

Components for 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 default messaging provider that is part of WebSphere Application Server and configure message-driven beans to use a JCA activation specification.

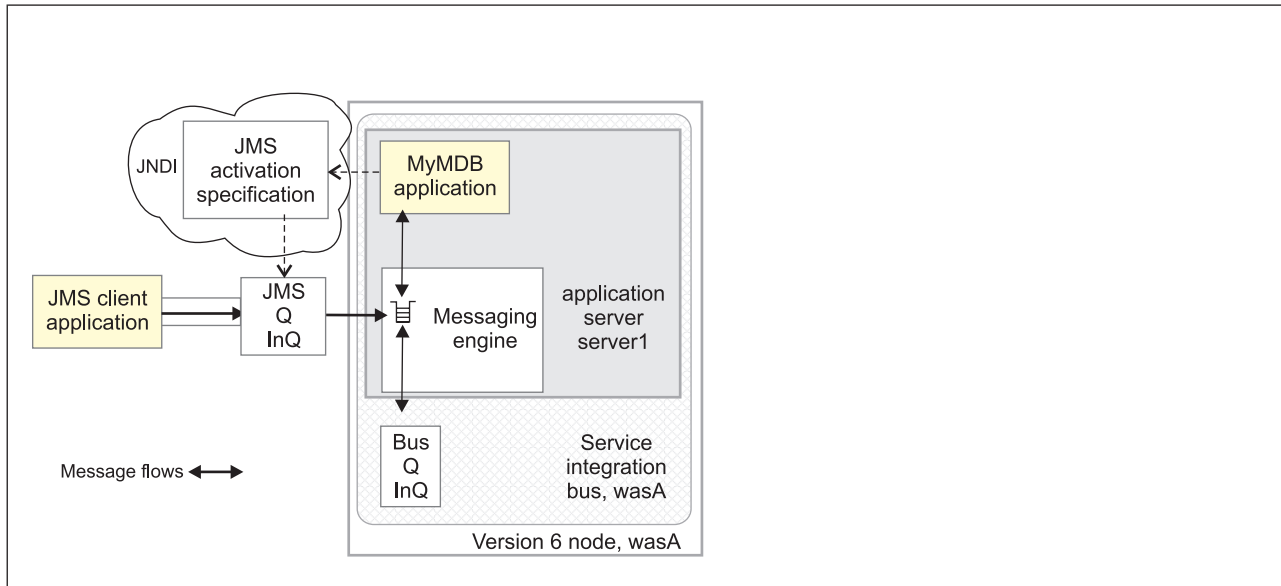


Figure 9. Message-driven bean components for the default messaging provider. This figure shows the main components of WebSphere support for message-driven beans for use with the default messaging provider.

With the default messaging provider, a message-driven bean acts as a listener on a specific JMS destination.

The administrator creates a JMS activation specification to provide information to the deployer about the configuration properties of a message-driven bean related to the processing of the inbound messages. For example, a JMS activation specification specifies the name of the service integration bus to connect to, 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" in the information center.

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 `MessageListener` types in its deployment descriptor (`ra.xml`). The `MessageListener` type 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 `MessageListener`-type interfaces provided by the resource adapter. By allowing multiple message listener types, 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 `MessageListener`-type 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 `MessageListener` type 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).

J2C activation specification configuration options and precedence: **Resource adapter scoped configuration**

A J2C activation specification configuration instance can be created and modified under an installed resource adapter at the cell, node, or server scope. This activation specification configuration is created based on a particular message listener type for the given resource adapter. Valid properties available for configuration are determined by introspection of the `ActivationSpec` class instance provided with the resource adapter. When created, an `ActivationSpec` class instance is referenced by its JNDI name. This activation specification configuration is needed during the deployment of a message-driven bean for the resource adapter.

Configuring a J2C activation specification instance at the cell, node, or server level offers two distinct advantages:

- The activation specification configuration information can be share among multiple message-driven beans across multiple applications.
- Updates to the configuration properties can be made without the need to redeploy the application.

Application-based configuration

Applications with message-driven beans have the option of specifying all, some, or none of the properties needed by the `ActivationSpec` class. 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.

To deploy and activate a message-driven bean with respect to application specification configuration properties would be as follows:

1. Use JNDI to look up a J2C activation specification configuration instance, which is based on its resource adapter-scoped definition.
2. Set the properties needed by the `ActivationSpec` class to the values defined by the cell, node, or server definition. If any of the properties are also defined as `activation-config` properties of the application, use the value defined by the `activation-config` property.
3. During application startup, the server activates the MDB endpoint by calling the resource adapter and passing a configured instance of the `ActivationSpec`.
4. Note that a resource adapter can specify in its deployment descriptor if a given `ActivationSpec` property is required. If it is required, and it is not supplied either by the cell, node or server definition, or an `activation-config` property from the applications deployment descriptor, then an exception is raised as part of the sequence to activate the message-driven bean.

WebSphere activation specification optional binding properties:

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 read from destinations (or endpoints) within the scope of a transaction. 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.

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 operates as a part of the WebSphere Application Server global security, and is enabled only when global security is enabled.

When global 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" in the information center.

Messaging: Resources for learning

- 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 605.

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. Define 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 define JMS resources for the default messaging provider.

For more information about the default messaging provider, see "Using the default messaging provider" in the information center.

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.

For more information about programming to use asynchronous messaging in WebSphere enterprise applications, see the following topics:

- "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 630 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 implementing WebSphere enterprise applications that use JMS, see the following topics:

- “Designing an enterprise application to use JMS”
- “Developing a J2EE application to use JMS” on page 624
- “Developing a JMS client” on page 627
- “Deploying a J2EE application to use JMS” on page 630

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
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" in the information center.

- 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 “Using JMS report messages” on page 621.
- 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 JMSEException 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 `createxxxSession` calls:
 - The `transacted` flag passed on `createxxxSession` 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` 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 `createxxxSession` call, then messages are automatically acknowledged by the application server and `Message.acknowledge()` is not used.

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.

Using 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 <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • MQRO_EXPIRATION • MQRO_EXPIRATION_WITH_DATA • MQRO_EXPIRATION_WITH_FULL_DATA
Confirm on arrival (COA)	<p>Send a report message when the request message has been put to the target queue.</p> <p>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.</p> <p>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.</p> <p>If a forward-routing path is used, the COA message are generated when the message arrives at the final destination in the path.</p>	JMS_IBM_Report_COA <ul style="list-style-type: none"> • MQRO_COA • MQRO_COA_WITH_DATA • MQRO_COA_WITH_FULL_DATA

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Confirm on delivery (COD)	<p>Send a report message when the request message has been destructively got by a message consumer.</p> <p>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.</p> <p>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.</p>	<p>JMS_IBM_Report_COD</p> <ul style="list-style-type: none"> • 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.	<p>JMS_IBM_Report_PAN</p> <ul style="list-style-type: none"> • MQRO_PAN
Negative action notification (NAN)	Ask the consumer application to send a report message if it has not successfully processed the request message.	<p>JMS_IBM_Report_NAN</p> <ul style="list-style-type: none"> • 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

This the default. the correlation Id of the report message is set to the correlation Id of the request message.

The following constants, in the SIB.mfp.api component, are available through the JMS API:

REPORT_NO_DATA
REPORT_WITH_DATA
REPORT_WITH_FULL_DATA
REPORT_EXPIRY
REPORT_EXCEPTION
REPORT_COA
REPORT_COD

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 createQueueSession 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 JMSEExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

Unlike normal Java exceptions, a JMSEException can contain another exception embedded in it. The implementation of JMSEException 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 (JMSEException 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, based on a sample client provided with WebSphere Application Server. 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 JMSEException, 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 `JMSEExceptions` to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

Unlike normal Java exceptions, a `JMSEException` can contain another exception embedded in it. The implementation of `JMSEException` 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 (JMSEException je)
{
    System.out.println("JMS failed with "+je);
    Exception le = je.getLinkedException();
    if (le != null)
    {
        System.out.println("linked exception "+le);
    }
}
```

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.

For more information about implementing WebSphere enterprise applications that use message-driven 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 ActivationSpec must be the same in both the ActivationSpec 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](#).

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 *nameBean*, 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.TimerObject 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`:

The result of this step is a message-driven bean that can be assembled into an EAR file for

```
public void onMessage(javax.jms.Message msg)
{
    String text      = null;
    String messageID = null;

    try
    {
        text = ((TextMessage)msg).getText();

        System.out.println("senderBean.onMessage(), msg text2: "+text);

        //
        // store the message id to use as the Correlator value
        //
        messageID = msg.getJMSMessageID();

        // Call a private method to put the message onto another queue
        putMessage(messageID, text);
    }
    catch (Exception err)
    {
        err.printStackTrace();
    }
    return;
}
```

Figure 10. Code example: The `onMessage()` method of a message bean. This figure shows a code extract for a basic `onMessage()` method of a sample message-driven bean. The method unpacks the incoming text message to extract the text and message identifier and calls a private `putMessage` method (defined within the same message bean class) to put the message onto another queue.

deployment.

3. **Optional:** Use the EJB deployment descriptor editor to review and, if needed, change the deployment properties. You can use the EJB deployment descriptor editor to review deployment properties that you specified on the EJB Creation Wizard (like Transaction type and Message selector) and other default deployment properties.

If needed, you can override the values of these properties later, after the enterprise application has been exported into an EAR file for deployment.

- a. In the property pane, select the Beans tab.
- b. Specify general deployment properties.

Transaction type

Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean. 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 createxxxSession 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.

Non-durable

Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

Message selector

The JMS message selector to be used to determine which messages the message bean receives; for example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

For more details about these properties, see “Message-driven bean deployment descriptor properties.”

- d. Specify bindings deployment properties.

Under WebSphere Bindings, select the JCA Adapter option then specify the bindings deployment properties:

ActivationSpec JNDI name

Type the JNDI name of the J2C activation specification that is to be used to deploy this message-driven bean. This name must match the name of a J2C activation specification that you define to WebSphere Application Server.

ActivationSpec Authorization Alias

The name of a J2C authentication alias used for authentication of connections to the JCA resource adapter. A J2C authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.

Destination JNDI name

Type the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI name space.

4. Assemble and package the application for deployment.

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:

The following deployment descriptor properties 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 `createxxxSession` 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" in the information center.
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 641.

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 the 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. 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

createxxxSession 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 635.

- 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.
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 EJB 2.1 message-driven beans.

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 one 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.

Message-driven beans samples: The following examples are provided, as part of the WebSphere Samples Gallery, to illustrate use of the message-driven beans support. When the Samples are installed on your local machine, they are available to try out. Locate them at <http://localhost:9080/WSsamples/> . (The default port is 9080.) For more information about where to find the Samples Gallery, see Samples Gallery.

- Point-to-point samples:

- "Tutorial: Creating JMS message sample"

This tutorial is designed to help you develop and deploy a JMS message sample application that tests the WebSphere Application Server message-driven beans support in a point-to-point scenario. This sample illustrates how to develop and deploy an application that comprises the following components:

- A Java/JMS program that writes a message to a queue.
- A message-driven bean that is invoked by a JMS listener when a message arrives on a defined queue.

For more information about this sample, see the samples article "Tutorial: Creating JMS message sample" that is installed with the Samples option.

- "Sample: Message Listener (point-to-point)"

This sample is designed to demonstrate the use and behavior of message-driven beans for a simple point-to-point scenario. This sample uses the JMS message sample deployed in the sample above.

For more information about this sample, see the samples article "Sample: Message Listener (Point-to-Point)" that is installed with the Samples option.

- Publish/subscribe samples

- "Tutorial: Creating JMS message publish/subscribe sample"

This tutorial is designed to help you develop and deploy a JMS message sample application that tests the WebSphere Application Server message-driven beans support in a publish/subscribe scenario. This sample illustrates how to develop and deploy an application that comprises the following components:

- A client program that starts the message sequence by publishing a message to a selected topic.
- A message-driven bean that is invoked by a JMS listener when the broker passes a message to the listener from a topic to which it has subscribed.

For more information about this sample, see the samples article "Tutorial: Creating JMS message publish/subscribe sample" that is installed with the Samples option.

- "Sample: Message Listener (publish/subscribe)"

This sample is designed to demonstrate the use and behavior of message-driven beans for a simple publish/subscribe scenario. This sample uses the JMS message sample deployed in the publish/subscribe sample above.

For more information about this sample, see the samples article "Sample: Message Listener (publish/subscribe)" that is installed with the Samples option.

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" in the information center.

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 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, 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 the 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 createxxxSession 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 635.

6. Specify bindings deployment properties.
 - a. Under WebSphere Bindings, specify the following property:
Listener port name
Type the name of the listener port for this message-driven bean.
7. Save your changes to the deployment descriptor.
 - a. Close the deployment descriptor editor.
 - b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.
8. Verify the archive files.
9. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
10. **Optional:** Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard. Note 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.

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 version 5.

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
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>.

Using WebSphere MQ functions from JMS applications

This topic provides a brief description about mapping the JMS message structure onto a WebSphere MQ message.

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

Learn about mail, URLs, and other J2EE resources

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Add mail capabilities to an application, and troubleshoot mail

- Look up a JavaMail session with JNDI
- Declare and create mail session references for an application
- Configure mail providers and sessions
- Enable debugging for a mail session

Use URL resources and resource environment entries

- Use URL resources within an application
- Establish a resource environment provider for an application
- Declare resource environment entries
- Configure resource environment entries

Conceptual overviews

Documentation

Refer to the article *Introduction: Mail, URLs, and other J2EE resources* in the information center.

Presentations

Education on Demand offers:

- Resource management

See Chapter 15 of the IBM Redbook IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

Samples are not available at this time.

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 Providers** in the console navigation tree. Then, click **Mail Provider** > *mail_provider* > **Mail Session**.
 - c. Click the *mail_provider* or *mail_session* that you want to modify. To remove a mail provider or mail session, click **Remove** after making your selection.
 - d. Click **Apply** or **OK**.
 - e. 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 1.2, 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 using the following Sun licensed packages:

- `mail.jar` - Contains the JavaMail APIs, and the SMTP, IMAP, and POP3 service providers.
- `activation.jar` - Contains the JavaBeans Activation Framework.

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. See the article, *Mail: Resources for learning*, for a link to this documentation.

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.

Mail migration tip

Parts of the JavaServer Page (JSP) 1.2 specification change the way the `EmailBean` class works with `Email.jsp`.

The specifications state that the JSP container creates a JSP page implementation class for each JSP page. The name of the JSP page implementation class is implementation-dependent. The JSP page implementation object belongs to an implementation-dependent named package which can vary between one JSP and another; therefore minimal assumptions should be made. The unnamed package should not be used without explicit import of the class.

Following these specifications, you should place `EmailBean.class` in a package referred to it by the fully qualified `packageName` in `Email.jsp`. Otherwise, `Email.jsp` is unable to find `EmailBean.class`.

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.jar` and `activation.jar` files. WebSphere Application Server uses the general-purpose JavaMail configuration files contained in the `mail.jar` and `activation.jar` files and does not put any mail configuration files in `<user.home>` and `<java.home>/lib`. File read permission for both the `mail.jar` and `activation.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}${java}${jre}${lib}${ext}${mail.jar}", "read";
  permission java.io.FilePermission "${was.install.root}${java}${jre}${lib}${ext}${activation.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}${java}${jre}${lib}${ext}${mail.jar}", "read";
  permission java.io.FilePermission "${was.install.root}${java}${jre}${lib}${ext}${activation.jar}", "read";
  permission java.io.FilePermission "${user.home}${mailcap}", "read";
  permission java.io.FilePermission "${java.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 Version 6.0 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.

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/ur1` 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 Providers** in the console navigation tree.
 - c. Click *URL_provider>URLs*.
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 Providers** in the console navigation tree.
 - c. Click **URL Provider > URLs**.
 - 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 create new URL providers to handle URL protocols that are not supported by the IBM Developer Kit For the Java™ Platform. You also have the option of selecting the default URL provider, which uses the URL support provided by the kit. Any URL resource with protocols based on Java 2 Standard Edition 1.3.1, such as HyperText Transfer Protocol (HTTP) or File Transfer Protocol (FTP), can use the default URL provider.

To view this administrative console page, click **Resources > URL Providers**.

Name:

Specifies the administrative name for the URL provider.

Description:

Describes the URL provider for your administrative records.

URL provider settings

Use this page create new URL providers.

To view this administrative console page, click **Resources > URL Providers > URL_provider**.

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 configuration collection

Use this page to view existing Uniform Resource Locator (URL) configurations, as well as begin configuring new URLs that point to electronically accessible resources (such as directory files on a machine in a network, or a document stored in a database).

To view this administrative console page, click **Resources > URL Providers > URL_provider > URLs**.

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.

URL configuration settings

Use this page to configure Uniform Resource Locators (URLs) that point to electronically accessible resources, such as a directory file on a machine in a network, or a document stored in a database.

To view this administrative console page, click **Resources > URL Providers > URL_provider > URLs > URL**.

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.

Spec:

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 a environment resource factory. Begin by clicking **Resources >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 654 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 Providers > your_resource_environment_provider > Referenceables**. Click **New** to begin the configuration process. See the "Referenceables settings" on page 655 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 the resource environment providers.

To view this administrative console page, click **Resources >Resource Environment Providers**.

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 provider settings:

Use this page to create settings for a resource environment provider.

To view this administrative console page, click **Resources >Resource Environment Providers > resource environment 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 Providers > New**.

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 resource environment entries.

An environment resource can be of any arbitrary type. See the latest EJB specification for more information about resource environment references and environment resources.

To view this administrative console page, click **Resources >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

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 set resource environment entries, which define configuration for an environment resource that is the binding target for a resource-environment-reference in some application's deployment descriptor.

To view this administrative console page, click **Resources >Resource Environment Providers > resource_environment_provider > Resource Environment Entries > resource_environment_entry**.

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 that holds the factoryClass object name of the factory that converts information in the name space into a class instance for the type of resource desired, and for the class name of the type to be returned.

Data type Drop-down menu

Referenceables collection

Use this page to specify the class name of the factory that will convert information in the name space into a class instance for the type of resource desired.

To view this administrative console page, click **Resources >Resource Environment Providers > resource_environment_provider > Referenceables**.

Factory Classname:

Specifies a javax.naming.ObjectFactory implementation class name

Data type String

Classname:

Specifies the Java type to which a referenceable provides access, for binding validation and to create the reference.

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 for the type of resource desired

To view this administrative console page, click **Resources >Resource Environment Providers > resource_environment_provider > Referenceables > referenceable**.

Factory Classname:

Specifies a javax.naming.ObjectFactory implementation class name

Data type String

Classname:

Specifies the Java type to which a Referenceable provides access, for binding validation and to create the reference.

Data type String

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.

Security

Securing applications and their environments

WebSphere Application Server supports the J2EE model for creating, assembling, securing, and deploying applications. This article provides a high-level description of what is involved in securing resources in a J2EE environment. Applications are often created, assembled and deployed in different phases and by different teams.

Consult the J2EE specifications for complete details.

1. Plan to secure your applications and environment. For more information, see “Planning to secure your environment” on page 658. Complete this step before you install the WebSphere Application Server.
2. Consider pre-installation and post-installation requirements. For more information, see “Implementing security considerations at installation time” on page 668. For example, during this step, you learn how to protect security configurations after you install the product.
3. Migrate your existing security systems. For more information, see “Migrating security configurations from previous releases” in the information center.
4. Develop secured applications. For more information, see “Developing secured applications” on page 673.
5. Assemble secured applications. For more information, see “Assembling secured applications” on page 734. Development tools, such as the Chapter 6, “Assembling applications,” on page 1005 are used to assemble J2EE modules and to set the attributes in the deployment descriptors.

Most of the steps in assembling J2EE applications involve deployment descriptors; deployment descriptors play a central role in application security in a J2EE environment.

Application assemblers combine J2EE modules, resolve references between them, and create from them a single deployment unit, typically an Enterprise Archive (EAR) file. Component providers and application assemblers can be represented by the same person but do not have to be.

6. Deploy secured applications. For more information, see “Deploying secured applications” on page 744.

Deployer link entities referred to in an enterprise application are mapped to the runtime environment. The deployer:

- Maps actual users and groups to application roles
- Installs the enterprise application into the environment
- Makes the final adjustments needed to run the application

7. Test secured applications. For more information, see “Testing security” on page 756.
8. Manage security configurations. For more information, see “Administering security” in the information center.
9. Improve performance by tuning security configurations. For more information, see “Tuning security configurations” in the information center.
10. Troubleshoot security configurations. For more information, see “Troubleshooting security configurations” in the information center.

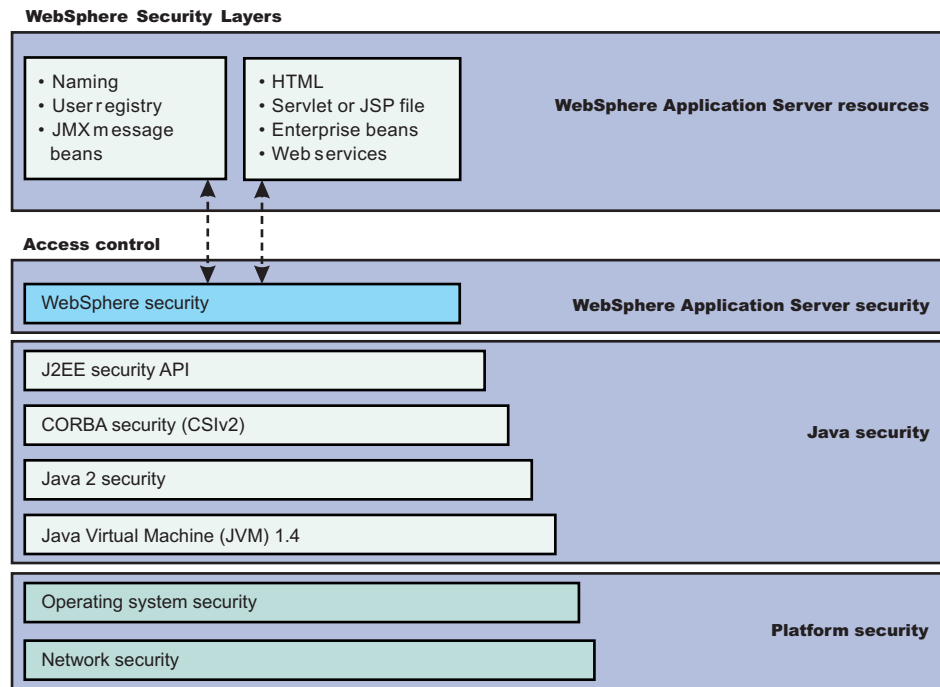
Your applications and production environment are secured.

See “Security: Resources for learning” in the information center for more information on the WebSphere Application Server security architecture.

Planning to secure your environment

There are several communication links from a browser on the Internet, through Web servers and product servers, to the enterprise data at the back-end. This section examines some typical configurations and common security practices. WebSphere Application Server security is built on a layered security architecture as showed in the following figure. This section also examines the security protection that is offered by each security layer and common security practice for good quality of protection in end-to-end security. The following figure illustrates the building blocks that comprise the operating environment for

security within WebSphere Application Server:



• **Operating System Security -**

The security infrastructure of the underlying operating system provides certain security services for WebSphere Application Server. These services include the file system security support that secure sensitive files in the product installation for WebSphere Application Server. The system administrator can configure the product to obtain authentication information directly from the operating system user registry.

- **Network Security** - The Network Security layers provide transport level authentication and message integrity and encryption. You can configure the communication between separate application servers to use Secure Sockets Layer (SSL) and HTTPS. Additionally, you can use IP Security and Virtual Private Network (VPN) for added message protection.
- **JVM 1.4** - The JVM security model provides a layer of security above the operating system layer.
- **Java 2 Security** - The Java 2 Security model offers fine-grained access control to system resources including file system, system property, socket connection, threading, class loading, and so on. Application code must explicitly grant the required permission to access a protected resource.
- **OMG CSIv2 Security** - Any calls made among secure Object Request Brokers (ORB) are invoked over the Common Security Interoperability Version 2 (CSIv2) security protocol that sets up the security context and the necessary quality of protection. After the session is established, the call is passed up to the enterprise bean layer. For backward compatibility, WebSphere Application Server supports the Secure Authentication Service (SAS) security protocol, which was used in prior releases of WebSphere Application Server and other IBM products.
- **J2EE Security** - The security collaborator enforces Java 2 Platform, Enterprise Edition (J2EE)-based security policies and supports J2EE security APIs.
- **WebSphere Security** - WebSphere Application Server security enforces security policies and services in a unified manner on access to Web resources, enterprise beans, and JMX administrative resources. It consists of WebSphere Application Server security technologies and features to support the needs of a secure enterprise environment.

WebSphere Application Server Network Deployment installation: The following figure shows a typical multiple-tier business computing environment for a WebSphere Application Server Network Deployment installation.

Important: There is a node agent instance on every computer node.

Each product application server consists of a Web container, an EJB container, and the administrative subsystem. The WebSphere Application Server deployment manager contains only WebSphere administrative code and the administrative console. The administrative console is a special J2EE Web application that provides the interface for performing administrative functions. WebSphere Application Server configuration data is stored in XML descriptor files, which must be protected by operating system security. Passwords and other sensitive configuration data can be modified using the administrative console. However, you must protect these passwords and sensitive data. For more information, see “Protecting plain text passwords” on page 671.

The administrative console Web application has a setup data constraint that requires the administrative console servlets and JSP files to be accessed only through an SSL connection when global security is enabled.

After installation, the administrative console HTTPS port is configured to use **DummyServerKeyFile.jks** and **DummyServerTrustFile.jks** with the default self-signed certificate. Using the dummy key and trust file certificate is not safe and you need to generate your own certificate to replace dummy ones immediately. It is more secure if you first enable global security and complete other configuration tasks after global security is enforced.

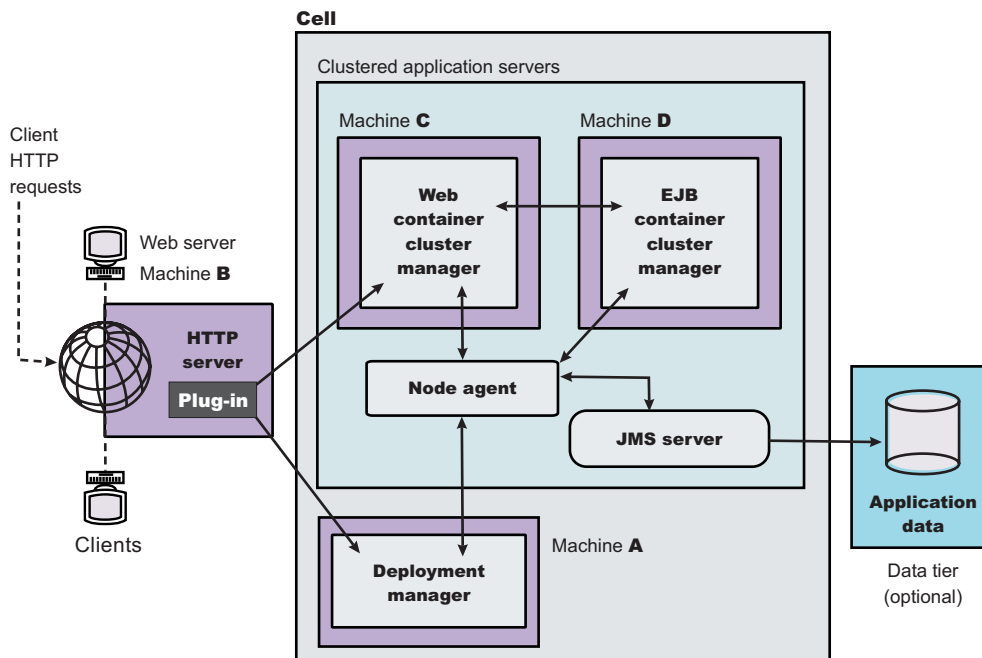


Figure 11. Multiple-tier business computing environment.

Global and administrative security:

WebSphere Application Servers interact with each other through CSv2 and Secure Authentication Services (SAS) security protocols as well as HTTP and HTTPS protocols.

You can configure these protocols to use Secure Sockets Layer (SSL) when you enable WebSphere Application Server global security. The WebSphere Application Server administrative subsystem in every server uses Simple Object Access Protocol (SOAP) Java Management Extensions (JMX) connectors and Remote Method Invocation over the Internet Inter-ORB Protocol (RMI/IIOP) JMX connectors to pass administrative commands and configuration data. When global security is disabled, the SOAP JMX connector uses HTTP protocol and the RMI/IIOP connector uses the TCP/IP protocol. When global

security is enabled, the SOAP JMX connector always uses HTTPS protocol. When global security is enabled, you can configure the RMI/IOP JMX connector to either use SSL or to use TCP/IP. It is recommended that you enable global security and enable SSL to protect the sensitive configuration data.

Global security and administrative security configuration is at the cell level.

When global security is enabled, you can disable application security at each individual application server by clearing the **Enable global security** option on the global security panel. The Global security panel is accessed through the administrative console by clicking **Security > Global security**. Disabling application server security does not affect the administrative subsystem in that application server, which is controlled by the global security configuration only. Both administrative subsystem and application code in an application server share the optional per server security protocol configuration. For more information, see "Configuring server security" in the information center.

Security for J2EE resources: Security for J2EE resources is provided by the Web container and the EJB container. Each container provides two kinds of security: declarative security and programmatic security.

In declarative security, an application security structure includes data integrity and confidentiality, authentication requirements, security roles, and access control. Access control is expressed in a form that is external to the application. In particular, the deployment descriptor is the primary vehicle for declarative security in the J2EE platform. WebSphere Application Server maintains J2EE security policy, including information derived from the deployment descriptor and specified by deployers and administrators in a set of XML descriptor files. At run time, the container uses the security policy that is defined in the XML descriptor files to enforce data constraints and access control.

When declarative security alone is not sufficient to express the security model of an application, you might use "Programmatic login" on page 689 to make access decisions. When global security is enabled and application server security is not disabled at the server level, J2EE applications security is enforced. When the security policy is specified for a Web resource, the Web container performs access control when the resource is requested by a Web client. The Web container challenges the Web client for authentication data if none is present according to the specified authentication method, ensures the data constraints are met, and determines whether the authenticated user has the required security role. The Web security collaborator enforces role-based access control by using an access manager implementation. An access manager makes authorization decisions that are based on security policy derived from the deployment descriptor. An authenticated user principal can access the requested servlet or JavaServer Pages (JSP) file if it has one of the required security roles. Servlets and JSP pages can use the `HttpServletRequest` methods `isUserInRole` and `getUserPrincipal`.

When global security is enabled and application server security is not disabled, the EJB container enforces access control on EJB method invocation.

The authentication takes place regardless of whether method permission is defined for the specific EJB method. The EJB security collaborator enforces role-based access control by using an access manager implementation. An access manager makes authorization decisions that are based on security policy derived from the deployment descriptor. An authenticated user principal can access the requested EJB method if it has one of the required security roles. EJB code can use the `EJBContext` methods `isCallerInRole` and `getCallerPrincipal`. Use the J2EE role-based access control to protect valuable business data from access by unauthorized users from both the Internet and the intranet. Refer to "Securing Web applications using an assembly tool" on page 737 and "Securing enterprise bean applications" on page 735.

Role-based security: WebSphere Application Server extends the security, role-based access control to administrative resources including the JMX system management subsystem, user registries, and JNDI name space. WebSphere administrative subsystem defines four administrative security roles:

Monitor role

A monitor can view the configuration information and status, but cannot make any changes.

Operator role

An operator can trigger run-time state changes, such as start an application server or stop an application, but cannot make configuration changes.

Configurator role

A configurator can modify the configuration information, but cannot change the state of the run time.

Administrator role

An operator as well as a configurator, which additionally can modify sensitive security configuration and security policy such as setting server ID and password, enable or disable global security and Java 2 security, and map users and groups to the administrator role.

A user with the configurator role can perform most administrative work including installing new applications and application servers. There are certain configuration tasks a configurator does not have sufficient authority to do when global security is enabled, including modifying a WebSphere Application Server identity and password, LTPA password and keys, and assigning users to administrative security roles.

Those sensitive configuration tasks require the administrative role because the server ID is mapped to the administrator role.

WebSphere Application Server administrative security is enforced when global security is enabled. It is recommended that WebSphere Application Server global security be enabled to protect administrative subsystem integrity. Application server security can be selectively disabled if there is no sensitive information to protect. For securing administrative security, refer to "Assigning users to administrator roles" and "Assigning users and groups to roles" on page 745.

Java 2 security permissions: WebSphere Application Server uses the Java 2 security model to create a secure environment to run application code. Java 2 security provides a fine-grained and policy-based access control to protect system resources such as files, system properties, opening socket connections, loading libraries, and so on. The J2EE Version 1.4 specification defines a typical set of Java 2 security permissions that Web and EJB components expect to have. These permissions are shown in the following table.

Table 2. J2EE security permissions set for Web components

Security Permission	Target	Action
java.lang.RuntimePermission	loadLibrary	
java.lang.RuntimePermission	queuePrintJob	
java.net.SocketPermission	*	connect
java.io.FilePermission	*	read, write
java.util.PropertyPermission	*	read

Table 3. J2EE security permissions set for EJB components

Security Permission	Target	Action
java.lang.RuntimePermission	queuePrintJob	
java.net.SocketPermission	*	connect
java.util.PropertyPermission	*	read

The WebSphere Application Server Java 2 security implementation is based on the J2EE Version 1.4 specification. The specification granted Web components read and write file access permission to any file

in the file system, which might be too broad. The WebSphere Application Server default policy gives Web components read and write permission to the subdirectory and the subtree where the Web module is installed. The default Java 2 security policy for all Java virtual machines and WebSphere Application Server processes are contained in the following policy files:

`${java.home}/jre/lib/security/java.policy`

Used as the default policy for the Java virtual machine (JVM).

`${USER_INSTALL_ROOT}/properties/server.policy`

Used as the default policy for all product server processes

To simplify policy management, WebSphere Application Server policy is based on resource type rather than code base (location). The following files are the default policy files for WebSphere Application Server subsystem. These policy files, which are an extension of WebSphere Application Server run time and are referred to as *Service Provider Programming Interfaces (SPI)*, are shared by multiple J2EE applications:

`${WAS_INSTALL_ROOT}/profiles/profile_name/config/cells/cell_name/nodes/node_name/spi.policy`

Used for embedded resources defined in the `resources.xml` file, such as the Java Message Service (JMS), JavaMail, and JDBC drivers.

`${WAS_INSTALL_ROOT}/profiles/profile_name/config/cells/cell_name/nodes/node_name/library.policy`

Used by the shared library that is defined by the WebSphere Application Server administrative console.

`${WAS_INSTALL_ROOT}/profiles/profile_name/config/cells/cell_name/nodes/node_name/app.policy`

Used as the default policy for J2EE applications.

In general, applications should not require more permissions to run than those recommended by the J2EE specification to be portable among various application servers. However, some applications might require more permissions. WebSphere Application Server supports a per application policy file, `was.policy`, to be packaged together with each application from granting extra permissions to that application.

Attention: Grant extra permissions to an application after careful consideration because of the potential of compromising the system integrity.

WebSphere Application Server uses a permission filtering policy file to alert users when an application requires permissions that are on the filter list during application installation and causes the offended application installation to fail. For example, it is recommended that you not give the `java.lang.RuntimePermission exitVM` permission to an application so that application code cannot terminate WebSphere Application Server. The filtering policy is defined by the `filterMask` in `${WAS_INSTALL_ROOT}/profiles/profile_name/config/cells/cell_name/filter.policy`. Moreover, WebSphere Application Server also performs run-time permission filtering that is based on the run-time filtering policy to ensure that application code is not granted a permission that is considered harmful to system integrity.

WebSphere Application Server Version 4 supported Java 2 Security, but enforced only three permissions checking against `exitVM`, `create` and `set the security manager`. Other permission checking is disabled by default.

Therefore, many applications developed for prior releases of WebSphere Application Server might not be Java 2 Security ready. To migrate those applications to WebSphere Application Server Version 6 quickly, you might temporarily give those applications `java.security.AllPermission` in the `was.policy` file. It is recommended to test or make those applications Java 2 Security ready; for example, identify what extra permissions, if any, are required and to grant only those permissions to a particular application. Not granting applications `AllPermission` can certainly reduce the risk of compromising system integrity. For more information on migrating applications to WebSphere Application Server Version 6, refer to "Migrating Java 2 security policy" in the information center.

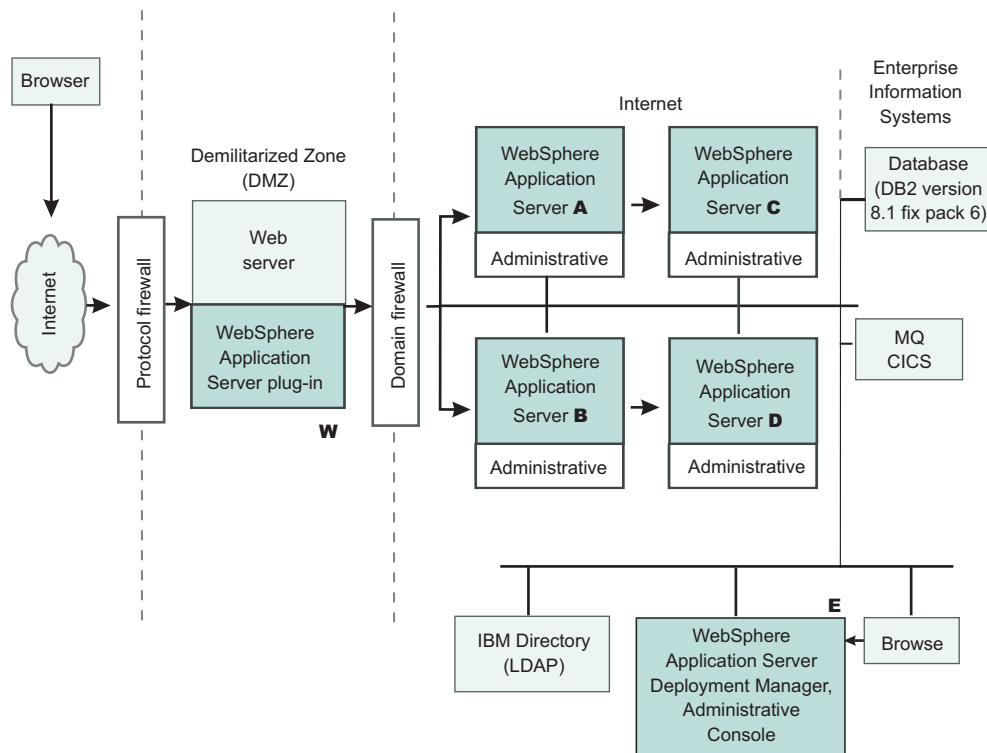
The WebSphere Application Server run time uses Java 2 Security to protect sensitive run-time functions; therefore, it is recommended that you enforce Java 2 security. Applications that are granted with `AllPermission` not only have access to sensitive system resources, but also WebSphere Application Server run-time resources and can potentially cause damage to both. In cases where an application can be trusted to be safe, WebSphere Application Server allows Java 2 Security to be disabled on a per application server basis. You can enforce Java 2 security by default in the security center and disable the per application server Java 2 Security flag to disable it at the particular application server.

When you specify the **Enable global security** and **Enable Java 2 Security** options on the Global security panel of the administrative console, the information, along with other sensitive configuration data, are stored in a set of XML configuration files. Both role-based access control and Java 2 Security permission-based access control are employed to protect the integrity of the configuration data. The example uses configuration data protection to illustrate how system integrity is maintained.

- When Java 2 security is enforced, the application code cannot access the WebSphere Application Server run-time classes that manage the configuration data unless it is granted the required WebSphere Application Server run-time permissions.
- When Java 2 security is enforced, application code cannot access the WebSphere Application Server configuration XML files unless it has been granted the required file read and write permission.
- The JMX administrative subsystem provides SOAP over HTTP or HTTPS and RMI/IIOP remote interface to enable application programs to extract and to modify configuration files and data. When global security is enabled, an application program can modify the WebSphere Application Server configuration if the application program has presented valid authentication data and the security identity has the required security roles.
- If a user can disable Java 2 security, then that user can modify the WebSphere Application Server configuration including the WebSphere Application Server security identity and authentication data along with other sensitive data. Only users with the administrator security role can disable Java 2 security.
- Because WebSphere Application Server security identity is given to the administrator role, only users with the administrator role can disable global security, to change server ID and password, and to map users and groups to administrative roles, and so on.

Other Runtime resources: Other WebSphere Application Server run time resources are protected by a similar mechanism as described previously. It is very important to enable WebSphere Application Server global security and to enforce Java 2 Security. J2EE Specification defines several authentication methods for Web components: HTTP Basic Authentication, Form-Based Authentication, and HTTPS Client Certificate Authentication. When you use client certificate login, it is more convenient for the browser client if the Web resources have integral or confidential data constraint. If a browser uses HTTP to access the Web resource, the Web container automatically redirects the browser to the HTTPS port. The CSv2 security protocol also supports client certificate authentication. You can also use SSL client authentication to setup secure communication among a selected set of servers based on a trust relationship.

If you start from the WebSphere Application Server plug-in at the Web server, you can configure SSL mutual authentication between it and the WebSphere Application Server HTTPS server. When using a self-signed certificate, you can restrict the WebSphere Application Server plug-in to communicate with only the selected two WebSphere Application Server servers as shown in the following figure. Suppose you want to restrict the HTTPS server in WebSphere Application Server **A** and in WebSphere Application Server **B** to accept secure socket connections only from the WebSphere Application Server plug-in **W**. You can generate three self-signed certificates using the IKEYMAN and the certificate management utilities. For example, use certificate **W** and trust certificate **A** and **B**. The HTTPS server of WebSphere Application Server **A** is configured to use certificate **A** and to trust certificate **W**. The HTTPS server of WebSphere Application Server **B** is configured to use certificate **B** and to trust certificate **W**. For more information on IKEYMAN, refer to "Starting the key management utility (iKeyman)" in the information center.



The trust relationship depicted in the previous picture is shown in the following table.

Server	Key	Trust
WebSphere Application Server plug-in	W	A, B
WebSphere Application Server A	A	W
WebSphere Application Server B	B	W

When WebSphere Application Server is configured to use an Lightweight Directory Access Protocol (LDAP) user registry, you also can configure SSL with mutual authentication between every application server and the LDAP server with self-signed certificate so that a password is not passed in clear text from WebSphere Application Server to the LDAP server. In this example, the node agent processes are not discussed. Each node agent must communicate with application servers on the same node and with the Deployment Manager. Node agents also must communicate with LDAP servers when they are configured to use an LDAP user registry. It is reasonable to let the deployment manager and the node agents use the same certificate. Suppose application server **A** and **C** are on the same computer node. The Node agent on that node needs to have certificates **A** and **C** in its trust file. WebSphere Application Server does not provide a user registry configuration or management utility. In addition, it does not dictate the user registry password policy. It is recommended that you use the password policy recommended by your user registry, including the password length and expiration period.

1. Determine which versions of WebSphere Application Server you are using.
2. Review the WebSphere Application Server security architecture.
3. Review each of the following topics in the information center.
 - Authentication protocol for EJB security
 - Supported authentication protocols
 - Common Secure Interoperability Version 2 features
 - Identity assertion
 - Authentication mechanisms
 - Lightweight Third Party Authentication settings

- Trust associations
- Single signon (SSO)
- User registries
 - Local operating system user registries
 - Lightweight Directory Access Protocol
- Custom user registries
- Java 2 security
 - Java 2 security policy files
- Java Authentication and Authorization Service (JAAS)
 - “Programmatic login” on page 689
- J2EE Connector security
- Access control exception
 - “Role-based authorization” on page 739
 - Administrative console and naming service authorization
- Secure Socket Layer (SSL)
 - Authenticity
 - Confidentiality
 - Integrity

Security considerations when adding a Base Application Server node to Network Deployment

At some point, you might decide to centralize the configuration of your stand-alone base application servers by adding them into a Network Deployment cell. If your base application server is currently configured with security, there are some issues to consider. The major issue when adding a node to the cell is whether the user registries between the base application server and the Deployment Manager are the same.

When adding a node to the cell, you automatically inherit both the user registry and the authentication mechanism of the cell.

For distributed security, all servers in the cell must use the same user registry and authentication mechanism. To recover from a user registry change, you must modify your applications so that the user and group to role mappings are correct for the new user registry. To do this, see the article on “Assigning users and groups to roles” on page 745.

Another major issue is the SSL public-key infrastructure. Prior to performing `addNode` with the Deployment Manager, verify that `addNode` can communicate as an SSL client with the Deployment Manager. This requires that the `addNode` truststore (configured in `sas.client.props`) contains the signer certificate of the Deployment Manager personal certificate as found in the keystore (specified in the administrative console).

See the article, “Managing digital certificates” in the information center.

The following are other issues to consider when running the `addNode` command with security:

1. When attempting to run system management commands such as `addNode`, you need to explicitly specify administrative credentials to perform the operation. The `addNode` command accepts `-username` and `-password` parameters to specify the `userid` and `password`, respectively. The user ID and password that are specified must be an administrative user; for example, a user that is a member of the console users with **Operator** or **Administrator** privileges or the administrative user ID configured in the User Registry. An example for `addNode`, `addNode CELL_HOST 8879 -includeapps -username user -password pass. -includeapps` is optional, but this option attempts to include the server applications into the Deployment Manager. The `addNode` command might fail if the user registries used by the WebSphere Application Server and the Deployment Manager are not the same. To correct this problem, either make the user registries the same or turn off security. If you change the user registries, remember to verify that the users to roles and groups to roles mappings are correct. See the `addNode` command for more information on the `addNode` syntax.

2. Adding a secured remote node through the administrative console is not supported. You can either disable security on the remote node before performing the operation or perform the operation from the command line using the addNode script.
3. Before running the addNode command, you must verify that the truststore files on the nodes can communicate with the keystore files from the Deployment Manager and vice versa. When using the default DummyServerKeyFile and DummyServerTrustFile, you should not see this problem as these are already able to communicate. However, never use these dummy files in a production environment or anytime sensitive data is being transmitted.
4. After running addNode, the application server is in a new SSL domain. It might contain SSL configurations that point to keystore and truststore files that are not prepared to interoperate with other servers in the same domain. Consider which servers will be intercommunicating and ensure that the servers are trusted within your truststore files.

Proper understanding of the security interactions between distributed servers greatly reduces problems encountered with secure communications. Security adds complexity because additional function needs to be managed. For security to function, it needs thorough consideration during the planning of your infrastructure. This document helps to reduce the problems that could occur due to inherent security interactions.

When you have security problems related to the WebSphere Application Server Network Deployment environment, check the "Troubleshooting security configurations" section in the information center to see if you can get information about the problem. When trace is needed to solve a problem, because servers are distributed, quite often it is required to gather trace on all servers simultaneously while recreating the problem. This trace can be enabled dynamically or statically, depending on the type problem occurring.

Creating login key files

1. Create a login key file. The authenticating user IDs, passwords, and target realms for each different target server are specified in the login key file, which is an ASCII file. When the security authentication service processes the login key file, the passwords in the file are encoded.
2. Add information to the login key file in the following format:

```
Realm_name    User_ID    Password
```

3. Make sure that the data conforms to the following rules:
 - One realm name
 - One user ID, and one password defined in each entry
 - One entry per line
 - No blank lines between entries
 - Comments on separate lines only
 - Begin any comment with a pound sign (#):

Example:

```
# Sample key file
#
# First target realm
#
TargetRealm serverID serverPassword
#
# Second target realm
#
TargetRealm2 serverID2 serverPassword2
#
# End of key file
```

A sample file named `wsserver.key` also contains these instructions. After installation, you can locate this sample file in the `install_root/properties` directory. You can use or modify the sample file as needed for testing.

Note: You can place the login key file anywhere on a host machine running the application server. However, it is recommended that you place the login key file under a securable file system .

After creating the login key files, read the article entitled, “Preparing truststore files.”

Preparing truststore files

Secure Sockets Layer (SSL) protocol protects the communication between WebSphere Application Servers. To complete the SSL connection, establish a valid truststore file for the WebSphere Application Server. A truststore file is a key database file that contains the public keys (See “Creating login key files” on page 667 for information about how to create a new keystore file.)

1. Extract the public key of the server by using the key management tool from WebSphere Application Server. For details, see “Configuring the server for request decryption: choosing the decryption method” in the information center.
2. Add the public key from the WebSphere Application Server as a signer certificate into the requesting WebSphere Application Server truststore file. For details, see the related information about how to import signer certificates.

The WebSphere Application Server truststore file is now ready to use for SSL connections with the WebSphere Application Server.

See “Configuring the application server for interoperability” for interoperability.

Configuring the application server for interoperability

After the truststore file is ready, complete the following steps to configure the WebSphere Application Server.

1. Configure the enterprise beans that access WebSphere Application Server. Before deploying the enterprise beans, configure the RunAs Identity.
2. Enable security.
3. Enable outbound SAS authentication protocol.
4. Specify the truststore file in an Secure Sockets Layer (SSL) configuration alias and configure the WebSphere Application Server with that alias.
5. Set the **Request timeout** and **Locate request timeout** values to zero for the Object Request Broker (ORB) service.
6. Specify a security property named `com.ibm.CORBA.keyFileName` for the absolute path of the login key file created earlier.
7. Restart the WebSphere Application Server.

Implementing security considerations at installation time

Complete the following tasks to implement security before, during, and after installing WebSphere Application Server.

1. “Securing your environment before installation” on page 669. This step describes how to install WebSphere Application Server with the proper authority.
2. Install the WebSphere Application Server. This step describes how to install WebSphere Application Server as the root user on a UNIX platform or as an administrator on a Windows platform.

During installation you are prompted to migrate security configurations from previous releases.

3. “Securing your environment after installation.” This step provides information on how to protect password information after you install WebSphere Application Server.

Securing your environment before installation

The following instructions explain how to perform a product installation with proper authority on UNIX platforms, Linux platforms, Solaris operating environments, and Windows platforms.

UNIX platforms

On UNIX platforms, log on as **root** and verify that the umask value is **022**.

To verify that the umask value is **022**, execute the **umask** command.

To set up the umask value as **022**, execute the **umask 022** command.

Linux platforms and Solaris operating environments

On Linux platforms or Solaris operating environments, make sure that the `/etc` directory contains a shadow password file. The shadow password file is named `shadow` and is in the `/etc` directory. If the shadow password file does not exist, an error occurs after enabling global security and configuring the user registry as local operating system.

To create the shadow file, run the `pwconv` command (with no parameters). This command creates an `/etc/shadow` file from the `/etc/passwd` file. After creating the shadow file, you can configure local operating system security.

Windows platforms

On Windows platforms, the logon user must be a member of the administrator group with the rights of **Act as part of the operating system** and **Log on as a service**.

To add the rights to a user on a Windows 2000 platform:

1. Click **Start > Programs > Administrative Tools > Local Security Policy** (for domain configuration, select **Domain Security Policies**, instead).
2. From the Local Security Settings Panel, click **Local Policies > User Rights Assignment** and add the following rights to the user ID:
 - Act as part of the operating system
 - Log on as a service

Securing your environment after installation

WebSphere Application Server depends on several configuration files created during installation. These files contain password information and need protection. Although the files are protected to a limited degree during installation, this basic level of protection is probably not sufficient for your site. Verify that these files are protected in compliance with the policies of your site.

The files in the `install_root\profiles\profile_name\config` and `install_root\profiles\profile_name\properties`, except for those in the following list, need protection. For example, give permission to the user who logs onto the system for WebSphere Application Server primary administrative tasks. Other users or groups, such as WebSphere Application Server console users and console groups, who perform partial WebSphere Application Server administrative tasks, like configuring, starting servers and stopping servers, need permissions as well.

The files in the `install_root\profiles\profile_name\properties` directory that should not be protected are:

- TraceSettings.properties
 - client.policy
 - client_types.xml
 - implfactory.properties
 - sas.client.props
 - sas.stdclient.properties
 - sas.tools.properties
 - soap.client.props
 - wsadmin.properties
 - wsjaas_client.conf
1. Secure files on a Windows system:
 - a. Open the browser for a view of the files and directories on the machine.
 - b. Locate and right-click the file or the directory that you want to protect.
 - c. Click **Properties**.
 - d. Click the **Security** tab.
 - e. Remove the Everyone entry and any other user or group that you do not want to have access to the file.
 - f. Add the users who can access the files with the proper permission.
 2. Secure files on UNIX systems. This procedure applies only to the ordinary UNIX file system. If your site uses access-control lists, secure the files by using that mechanism. Any site-specific requirements can affect the owner, group, and corresponding privileges. For example, on AIX,
 - a. Go to the *install_root* directory and change the ownership of the directory configuration and properties to the user who logs onto the system for WebSphere Application Server primary administrative tasks. Run the following command: `chown -R logon_name directory_name`
Where:
 - *login_name* is a specified user or group.
 - *directory_name* is the name of the directory that contains the files.

It is recommended that you assign ownership of the files that contain password information to the user who runs the application server. If more than one user runs the application server, provide permission to the group in which the users are assigned in the user registry.
 - b. Set up the permission by running the following command: `chmod -R 770 directory_name`.
 - c. Go to the *install_root*\profiles*profile_name*\properties directory and set the following file permission to **everybody** by running the following command: `chmod 777 file_names`. where *file_names* are the following files:
 - TraceSettings.properties
 - client.policy
 - client_types.xml
 - implfactory.properties
 - sas.client.props
 - sas.stdclient.properties
 - sas.tools.properties
 - soap.client.props
 - wsadmin.properties
 - wsjaas_client.conf
 - d. Create a group for WebSphere Application Server and put the users who perform full or partial WebSphere Application Server administrative tasks in that group.
 - e. If you want to use WebSphere MQ as a JMS provider, restrict access to the /var/mqm directories and log files used. Give write access to the user ID mqm or members of the mqm user group only.

After securing your environment, only the users given permission can access the files. Failure to adequately secure these files can lead to a breach of security in your WebSphere Application Server applications.

If failures occur that are caused by file accessing permissions, check the permission settings.

Protecting plain text passwords

The WebSphere Application Server has several plain text passwords. These passwords are not encrypted, but are encoded. The following is a list of files with encoded passwords:

Important: *WAS_INSTALL_ROOT* is a WebSphere Application Server Environment variable that you can configure through the administrative console by clicking **Environment > WebSphere variables**.

File name	Additional information
<i>WAS_INSTALL_ROOT</i> \profiles \profile_name\config\cells \cell_name\security.xml	The following fields contain encoded passwords: <ul style="list-style-type: none"> • LTPA password • JAAS authentication data • User registry server password • LDAP user registry bind password • Key file password • Trust file password • Cryptographic token device password
<i>WAS_INSTALL_ROOT</i> \profiles \profile_name\properties \sas.client.props	Specifies passwords for: <ul style="list-style-type: none"> • com.ibm.ssl.keyStorePassword • com.ibm.ssl.trustStorePassword • com.ibm.CORBA.loginPassword
war/WEB-INF/ibm_web_bnd.xml	Specify passwords for the default basic authentication for the "resource-ref" bindings within all descriptors (except in the Java cryptography architecture)
ejb jar/META-INF/ibm_ejbjar_bnd.xml	Specify passwords for the default basic authentication for the "resource-ref" bindings within all descriptors (except in the Java cryptography architecture)
client jar/META-INF/ibm_appclient_bnd.xml	Specify passwords for the default basic authentication for the "resource-ref" bindings within all descriptors (except in the Java cryptography architecture)
ear/META-INF/ibm_application_bnd.xml	Specify passwords for the default basic authentication for the "run as" bindings within all descriptors
<i>WAS_INSTALL_ROOT</i> \profiles\profile_name\config\cells \cell_name\nodes\node_name\servers \server.xml	The following fields contain encoded passwords: <ul style="list-style-type: none"> • Key file password • Trust file password • Cryptographic token device password • Authentication target password • Session persistence password • DRS Client data replication password

File name	Additional information
<code>WAS_INSTALL_ROOT\profiles\profile_name\config\cells\cell_name\nodes\node_name\servers\server1\resources.xml</code>	The following fields contain encoded passwords: <ul style="list-style-type: none"> • WAS40Datasource password • mailTransport password • mailStore password • MQQueue queue mgr password
For WebSphere Application Server and WebSphere Application Server Express: <ul style="list-style-type: none"> • <code>WAS_INSTALL_ROOT\profiles\profile_name\config\cells\cell_name\ws-security.xml</code> • <code>WAS_INSTALL_ROOT\profiles\profile_name\config\cells\cell_name\nodes\node_name\servers\server1\ws-security</code> For Network Deployment: <code>WAS_INSTALL_ROOT\profiles\profile_name\config\cells\cell_name\ws-security.xml</code>	
<code>ibm-webservices-bnd.xmi</code>	
<code>ibm-webservicesclient-bnd.xmi</code>	
<code>WAS_INSTALL_ROOT \profiles\profile_name\properties\soap.client.props</code> <code>com.ibm.ssl.trustStorePassword</code>	Specifies passwords for: <ul style="list-style-type: none"> • <code>com.ibm.ssl.keyStorePassword</code> • <code>com.ibm.ssl.trustStorePassword</code> • <code>com.ibm.SOAP.loginPassword</code>
<code>WAS_INSTALL_ROOT \profiles\profile_name\properties\sas.tools.properties</code>	Specifies passwords for: <ul style="list-style-type: none"> • <code>com.ibm.ssl.keyStorePassword</code> • <code>com.ibm.ssl.trustStorePassword</code> • <code>com.ibm.CORBA.loginPassword</code>
<code>WAS_INSTALL_ROOT \profiles\profile_name\properties\sas.stdclient.properties</code> <code>com.ibm.ssl.keyStorePassword</code>	Specifies passwords for: <ul style="list-style-type: none"> • <code>com.ibm.ssl.keyStorePassword</code> • <code>com.ibm.ssl.trustStorePassword</code> • <code>com.ibm.CORBA.loginPassword</code>
<code>WAS_INSTALL_ROOT \profiles\profile_name \properties\wssserver.key</code>	

To re-encode a password in one of the previous files, complete the following steps:

1. Access the file using a text editor and type over the encoded password in plain text. The new password is shown in plain text and must be encoded.
2. Use the `PropFilePasswordEncoder.bat` or `PropFilePasswordEncode.sh` file in the `WAS_INSTALL_ROOT\profiles\profile_name\bin\` directory to re-encode the password.

If you are re-encoding SAS properties files, type `PropFilePasswordEncoder file_name -sas` and the `PropFilePasswordEncoder` file encodes the known SAS properties.

If you are encoding files that are not SAS properties files, type `PropFilePasswordEncoder file_name password_properties_list`

`file_name` is the name of the z/SAS properties file. `password_properties_list` is the name of the properties to encode within the file.

Use the `PropFilePasswordEncoder` utility to encode WebSphere Application Server password files only. The utility cannot encode passwords contained in XML files or other files that contain open and close tags.

If you reopen the affected file or files, the passwords do not display in plain text. Instead, the passwords appear encoded. WebSphere Application Server does not provide a utility for decoding the passwords.

PropFilePasswordEncoder command reference

Purpose

The **PropFilePasswordEncoder** command encodes passwords located in plain text property files. This command encodes both Secure Authentication Server (SAS) property files and non-SAS property files. After you have encoded the passwords, note that a decoding command does not exist. To encode passwords, you must run this command from the *install_dir/bin* directory of a WebSphere Application Server installation.

Syntax

The command syntax is as follows:

```
PropFilePasswordEncoder file_name
```

Parameters

The following option is available for the PropFilePasswordEncoder command:

-sas

Encodes SAS property files.

The following examples demonstrate the correct syntax.

```
PropFilePasswordEncoder file_name password_properties_list  
PropFilePasswordEncoder file_name -SAS
```

Developing secured applications

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 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 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 run time. 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.
2. Develop secure Web applications. For more information, see “Developing with programmatic security APIs for Web applications” on page 674.
3. Develop servlet filters for form login processing. For more information, see “Developing servlet filters for form login processing” on page 676.
4. Develop form login pages. For more information, see “Developing form login pages” on page 681.
5. Develop enterprise bean component applications. For more information, see “Developing with programmatic APIs for EJB applications” on page 685.
6. 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 696.

7. Develop your own Java 2 security mapping module. For more information, see "Configuring application logins for Java Authentication and Authorization Service" in the information center.
8. Develop custom user registries. For more information, see "Developing custom user registries" on page 724.
9. Develop a custom interceptor for trust associations. For more information, see "Trust association interceptor support for Subject creation" on page 732

Developing with programmatic security 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 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 containing the remote user name. If no user is authenticated, it returns **null**.

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 passed to this method. Since 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 development tools such as Rational Web Developer 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 is when a Web application wants 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 `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(), 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 737.

Example: Web applications code: The following example depicts a Web application or servlet using the programmatic security model. The following example is one usage and not necessarily the only usage of the programmatic security model. The application can use the information returned by the getUserPrincipal(), isUserInRole() and 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 as shown in the following example:

```

<security-role-ref>
<description> </description>
<role-name>Mgr</role-name>
</security-role-ref>

```

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 use the information contained in the requests or responses. One or more servlet filters can attach to a servlet or a group of servlets. Servlet filters also can attach to JSP files and HTML pages. All 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 compliant 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 containing the user name and the `j_password` parameter containing the password. A servlet filter can use the user name parameter and password information to perform more authentication or other special needs.

A servlet filter implements the `javax.servlet.Filter` class. There are three methods in the filter class that need implementing:

- **init(javax.servlet.FilterConfig cfg)**. This method is called by the container exactly 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. `FilterChain` passed to this method can be used to invoke the next filter in the chain of filters. The original requested servlet executes when the last filter in the chain calls the `chain.doFilter()` method. Therefore, all filters should call the `chain.doFilter()` method for the original servlet to execute after filtering. If an additional authentication check is implemented in the filter code and results in failure, the original servlet does not be execute. The `chain.doFilter()` method is not called and can be redirected to some other error page.

If a servlet maps to many servlet filters, servlet filters are called in the order that is listed in the deployment descriptor of the application (`web.xml`).

An example of a servlet filter follows: This login filter can map to `/j_security_check` 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.
    }
}
```

```
}  
}
```

Place the servlet filter class file in the WEB-INF/classes directory of the application.

Configuring servlet filters:

IBM Rational Application Developer or an assembly tool can configure the servlet filters. There are two steps 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 application deployment descriptor, `web.xml`, 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 URL or servlet.

After mapping the servlet filter to a servlet or a URL, the application deployment descriptor (`web.xml`) 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`, 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 \(Samples Gallery\)](#).

Example: Servlet filters: This example illustrates one way 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` was broken into two lines due to the width of the page. The `public void doFilter(ServletRequest request` line and the line after it are one continuous line.

Portion of the `web.xml` file showing the `LoginFilter` configured and mapped to `j_security_check`:

```
<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
```

Developing form login pages

A Web client or 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 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 is a strong authentication mechanism 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 (not the HTTP POST) request to get the page specified in the URL. 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 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 if required. This step is required when a 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 \(Samples Gallery\)](#).

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: For the authentication to proceed appropriately, the action of the login form must always be `j_security_check`. 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, it 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 showing login and error pages for the previous login configuration:

```
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 the form logout mechanism, access to a protected Web resource requires reauthentication. This feature is not required by J2EE specifications, but is provided as an additional feature in WebSphere security.

Suppose that it is desirable 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. It is an HTML or 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, it has a special post action. This post action is recognized by the Web container, which dispatches it to a special internal WebSphere 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 JSP file within the same Web application to which that the user is redirected after logging out. 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. For more information on how to access the form login sample, see [Accessing the Samples \(Samples Gallery\)](#).

Developing with programmatic APIs for 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` interface 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 specified security role (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 containing the bean caller name. If the caller is not authenticated, it returns a principal containing `UNAUTHENTICATED` name.

You can enable a login module to indicate which principal class is returned by these calls.

Refer to for more information.

When the `isCallerInRole()` method is used, declare a `security-role-ref` element in the deployment descriptor with a `role-name` subelement containing the role name passed to this method. Since 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 actual role. During assembly, assembler creates a `role-link` sub element to link the `role-name` to the actual role. Creation of a `security-role-ref` element is possible if development tools such as Rational Web Developer 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.

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. An example where this implementation is useful is 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, a security-role-ref element can be created:

```
<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 (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” on page 735.

Example: Enterprise bean application code: The following EJB component example illustrates the use of `isCallerInRole()` and `getCallerPrincipal()` methods in an EJB module. Using that declarative security is recommended. The following example is one way of using the `isCallerInRole()` and `getCallerPrincipal()` methods. The application can use this result in any way that is suitable.

A remote interface

File : Hello.java

```
package tests;
import java.rmi.RemoteException;
/**
 * Remote interface for Enterprise Bean: Hello
 */
public interface Hello extends javax.ejb.EJBObject {
    public abstract String getMessage()throws RemoteException;
    public abstract void setMessage(String s)throws RemoteException;
}
```

A home interface

File : HelloHome.java

```
package tests;
/**
 * Home interface for Enterprise Bean: Hello
 */
public interface HelloHome extends javax.ejb.EJBHome {
    /**
     * Creates a default instance of Session Bean: Hello
     */
    public tests.Hello create() throws javax.ejb.CreateException,
        java.rmi.RemoteException;
}
```

A bean implementation

File : HelloBean.java

```
package tests;
/**
 * Bean implementation class for Enterprise Bean: Hello
 */
public class HelloBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext mySessionCtx;
    /**
     * getSessionContext
     */
    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }
    /**
     * setSessionContext
     */
}
```

```

    */
    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }
    /**
     * ejbActivate
     */
    public void ejbActivate() {
    }
    /**
     * ejbCreate
     */
    public void ejbCreate() throws javax.ejb.CreateException {
    }
    /**
     * ejbPassivate
     */
    public void ejbPassivate() {
    }
    /**
     * ejbRemove
     */
    public void ejbRemove() {
    }

    public java.lang.String message;

    //business methods

    // all users can call getMessage()
    public String getMessage() throws java.rmi.RemoteException {
        return message;
    }

    // all users can call setMessage() but only few users can set new message.
    public void setMessage(String s) throws java.rmi.RemoteException {

        // 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 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” on page 735. Use the information under `Map security-role-ref` and `role-name` to `role-link` to create the element.

Programmatic login

Programmatic login is a type of form login that supports application presentation site-specific login forms for the purpose of authentication.

When enterprise bean client applications require the user to provide identifying information, the writer of the application must collect that information and authenticate the user. You can broadly classify the work of the programmer in terms of where the actual user authentication is performed:

- In a client program
- In a server program

Users of Web applications can receive prompts for authentication data in many ways. The `<login-config>` element in the Web application deployment descriptor file defines the mechanism used to collect this information. Programmers who want to customize login procedures, rather than relying on general purpose devices like a 401 dialog window in a browser, can use a form-based login to provide an application-specific HTML form for collecting login information.

No authentication occurs unless global security is enabled. If you want to use form-based login for Web applications, you must specify `FORM` in the `auth-method` tag of the `<login-config>` element in the deployment descriptor of each Web application.

Applications can present site-specific login forms by using the WebSphere Application Server `form-login` type. The Java 2 Platform, Enterprise Edition (J2EE) specification defines form login as one of the authentication methods for Web applications. WebSphere Application Server provides a `form-logout` mechanism.

Java Authentication and Authorization Service programmatic login

Java Authentication and Authorization Service (JAAS) is a new feature in WebSphere Application Server. It is also mandated by the J2EE 1.3 Specification. JAAS is a collection of strategic authentication APIs that replace the CORBA programmatic login APIs. WebSphere Application Server provides some extensions to JAAS:

Before you begin developing with programmatic login APIs, consider the following points :

- For the pure Java client application or client container application, initialize the client Object Request Broker (ORB) security prior to performing a JAAS login. Do this by executing the following code prior to the JAAS login:

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
// Perform an InitialContext and default lookup prior to logging
// in to initialize ORB security and for the bootstrap host/port
// to be determined for SecurityServer lookup. If you do not want
// to validate the userid/password during the JAAS login, disable
// the com.ibm.CORBA.validateBasicAuth property in the
```

```
// sas.client.props file.

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("");
```

For more information, see “Example: Programmatic logins” on page 699.

- For the pure Java client application or client container application, make sure that the host name and the port number of the target JNDI bootstrap properties are specified properly. See the Developing applications that use CosNaming (CORBA Naming interface) section for details.
- If the application uses custom JAAS login configuration, make sure that the custom JAAS login configuration is properly defined. See the “Configuring application logins for Java Authentication and Authorization Service” section in the information center for details.
- Some of the JAAS APIs are protected by Java 2 security permissions. If these APIs are used by application code, make sure that these permissions are added to the application was.policy file. See “Adding the was.policy file to applications,” “Using PolicyTool to edit policy files” and “Configuring the was.policy file” sections in the information center for details. For more details of which APIs are protected by Java 2 Security permissions, check the IBM Developer Kit, Java edition; JAAS and the WebSphere Application Server public APIs Javadoc for more details. The following list indicates the APIs used in the samples code provided in this documentation.
 - javax.security.auth.login.LoginContext constructors are protected by javax.security.auth.AuthPermission “createLoginContext”.
 - javax.security.auth.Subject.doAs() and com.ibm.websphere.security.auth.WSSubject.doAs() are protected by javax.security.auth.AuthPermission “doAs”.
 - javax.security.auth.Subject.doAsPrivileged() and com.ibm.websphere.security.auth.WSSubject.doAsPrivileged() are protected by javax.security.auth.AuthPermission “doAsPrivileged”.
- com.ibm.websphere.security.auth.WSSubject: Due to a design oversight in the JAAS 1.0, javax.security.auth.Subject.getSubject() does not return the Subject associated with the thread of execution inside a java.security.AccessController.doPrivileged() code block. This can present an inconsistent behavior that is problematic and causes undesirable effort. The com.ibm.websphere.security.auth.WSSubject API provides a work around to associate Subject to thread of execution. The com.ibm.websphere.security.auth.WSSubject API extends the JAAS model to J2EE resources for authorization checks. The Subject associated with the thread of execution within com.ibm.websphere.security.auth.WSSubject.doAs() or com.ibm.websphere.security.auth.WSSubject.doAsPrivileged() code block is used for J2EE resources authorization checks.
- UI support for defining new JAAS login configuration: You can configure JAAS login configuration in the administrative console and store it in the WebSphere Configuration API. Applications can define new JAAS login configuration in the administrative console and the data is persisted in the configuration repository (stored in the WebSphere Configuration API). However, WebSphere Application Server still supports the default JAAS login configuration format (plain text file) provided by the JAAS default implementation. But if there are duplication login configurations defined in both the WebSphere Configuration API and the plain text file format, the one in the WebSphere Configuration API takes precedence. There are advantages to defining the login configuration in the WebSphere Configuration API:
 - UI support in defining JAAS login configuration.
 - You can manage the JAAS configuration login configuration centrally.
 - The JAAS configuration login configuration is distributed in a Network Deployment installation.
- JAAS login configurations For WebSphere Application Server: WebSphere Application Server provides JAAS login configurations for application to perform programmatic authentication to the WebSphere Application Server security run time. These JAAS login configurations for WebSphere Application Server

perform authentication to the configured authentication mechanism (SWAM or LTPA) and user registry (Local OS, LDAP, or Custom) based on the authentication data supplied. The authenticated Subject from these JAAS login configurations contain the required Principal and Credentials that can be used by WebSphere Application Server security run time to perform authorization checks on J2EE role-based protected resources. Here is the JAAS login configurations provided by WebSphere Application Server:

- *WSLogin JAAS login configuration*: A generic JAAS login configuration that a Java Client, client container application, servlet, JSP file, enterprise bean, and so on, can use to perform authentication based on a user ID and password, or a token to the WebSphere Application Server security run time. However, this does not honor the CallbackHandler specified in the Client Container deployment descriptor.
- *ClientContainer JAAS login configuration*: This JAAS login configuration honors the CallbackHandler specified in the client container deployment descriptor. The login module of this login configuration uses the CallbackHandler in the client container deployment descriptor if one is specified, even if the application code specified one CallbackHandler in the LoginContext. This is for client container application.
- Subject authenticated with the previously mentioned JAAS login configurations contain a `com.ibm.websphere.security.auth.WSPPrincipal` and a `com.ibm.websphere.security.auth.WSCredential`. If the authenticated Subject is passed the `in com.ibm.websphere.security.auth.WSSubject.doAs()` (or the other `doAs()` methods), the WebSphere Application Server security run time can perform authorization checks on J2EE resources, based on the Subject `com.ibm.websphere.security.auth.WSCredential`.
- **Customer-defined JAAS login configurations**: You can define other JAAS login configurations. See "Configuring application logins for Java Authentication and Authorization Service" section for details. Use these login configurations to perform programmatic authentication to the customer authentication mechanism. However, the subjects from these customer-defined JAAS login configurations might not be used by WebSphere Application Server security run time to perform authorization checks if the subject does not contain the required principal and credentials.

Finding the root cause login exception from a JAAS login

If you get a `LoginException` after issuing the `LoginContext.login()` API, you can find the root cause exception from the configured user registry. In the login modules, the registry exceptions are wrapped by a `com.ibm.websphere.security.auth.WSLoginFailedException`. This exception has a `getCause()` method that allows you to pull out the exception that was wrapped after issuing the above command.

Note: You are not always guaranteed to get an exception of type `WSLoginFailedException`, but you should note that most of the exceptions generated from the user registry show up here.

The following is a `LoginContext.login()` API example with associated catch block. `WSLoginFailedException` has to be casted to `com.ibm.websphere.security.auth.WSLoginFailedException` if you want to issue the `getCause()` API.

Note: The `determineCause()` example below can be used for processing `CustomUserRegistry` exception types.

```
try
{
    lc.login();
}
catch (LoginException le)
{
    // drill down through the exceptions as they might cascade through the runtime
    Throwable root_exception = determineCause(le);

    // now you can use "root_exception" to compare to a particular exception type
    // for example, if you have implemented a CustomUserRegistry type, you would
```

```

// know what to look for here.
}

/* Method used to drill down into the WSLoginFailedException to find the
"root cause" exception */

public Throwable determineCause(Throwable e)
{
    Throwable root_exception = e, temp_exception = null;

    // keep looping until there are no more embedded WSLoginFailedException or
    // WSSecurityException exceptions
    while (true)
    {
        if (e instanceof com.ibm.websphere.security.auth.WSLoginFailedException)
        {
            temp_exception = ((com.ibm.websphere.security.auth.WSLoginFailedException)
                e).getCause();
        }
        else if (e instanceof com.ibm.websphere.security.WSSecurityException)
        {
            temp_exception = ((com.ibm.websphere.security.WSSecurityException)
                e).getCause();
        }
        else if (e instanceof javax.naming.NamingException)
            // check for Ldap embedded exception
            {
                temp_exception = ((javax.naming.NamingException)e).getRootCause();
            }
        else if (e instanceof your_custom_exception_here)
        {
            // your custom processing here, if necessary
        }
        else
        {
            // this exception is not one of the types we are looking for,
            // lets return now, this is the root from the WebSphere
            // Application Server perspective
            return root_exception;
        }
        if (temp_exception != null)
        {
            // we have an exception, let's go back and see if this has another
            // one embedded within it.
            root_exception = temp_exception;
            e = temp_exception;
            continue;
        }
        else
        {
            // we finally have the root exception from this call path, this
            // has to occur at some point
            return root_exception;
        }
    }
}

```

```

    }
  }
}

```

Finding the root cause login exception from a Servlet filter

You can also receive the root cause exception from a servlet filter when addressing post-Form Login processing. This is suitable because it shows the user what happened. The following API can be issued to obtain the root cause exception:

```

Throwable t = com.ibm.websphere.security.auth.WSSubject.getRootLoginException();
if (t != null)
    t = determineCause(t);

```

Note: Once you have the exception you can run it through the `determineCause()` example above to get the native registry root cause.

Enabling root cause login exception propagation to pure Java clients

Currently, the root cause does not get propagated to a pure client for security reasons. However, you might want to propagate the root cause to a pure client in a trusted environment. If you want to enable root cause login exception propagation to a pure client, click **Security > Global Security > Custom Properties** on the WebSphere Application Server administrative console and set the following property:

```
com.ibm.websphere.security.registry.propagateExceptionsToClient=true
```

Non-prompt programmatic login

WebSphere Application Server provides a non-prompt implementation of the `javax.security.auth.callback.CallbackHandler` interface, which is called `com.ibm.websphere.security.auth.callback.WSCallbackHandlerImpl`. Using this interface, an application can push authentication data to the WebSphere LoginModule instance to perform authentication. This capability proves useful for server-side application code to authenticate an identity and to use that identity to invoke downstream J2EE resources.

```

javax.security.auth.login.LoginContext lc = null;

try {
    lc = new javax.security.auth.login.LoginContext("WSLogin",
        new com.ibm.websphere.security.auth.callback.WSCallbackHandlerImpl("user",
            "securityrealm", "securedpassword"));

    // create a LoginContext and specify a CallbackHandler implementation
    // CallbackHandler implementation determine how authentication data is collected
    // in this case, the authentication data is "push" to the authentication mechanism
    // implemented by the LoginModule.
} catch (javax.security.auth.login.LoginException e) {
    System.err.println("ERROR: failed to instantiate a LoginContext and the exception: "
        + e.getMessage());
    e.printStackTrace();

    // may be javax.security.auth.AuthPermission "createLoginContext" is not granted
    // to the application, or the JAAS login configuration is not defined.
}

if (lc != null)

```

```

try {
lc.login(); // perform login
javax.security.auth.Subject s = lc.getSubject();
// get the authenticated subject

// Invoke a J2EE resource using the authenticated subject
com.ibm.websphere.security.auth.WSSubject.doAs(s,
new java.security.PrivilegedAction() {
public Object run() {
try {
bankAccount.deposit(100.00); // where bankAccount is a protected EJB
} catch (Exception e) {
System.out.println("ERROR: error while accessing EJB resource, exception: "
+ e.getMessage());
e.printStackTrace();
}
return null;
}
});
} catch (javax.security.auth.login.LoginException e) {
System.err.println("ERROR: login failed with exception: " + e.getMessage());
e.printStackTrace();

// login failed, might want to provide relogin logic
}

```

You can use the `com.ibm.websphere.security.auth.callback.WSCallbackHandlerImpl` callback handler with a pure Java client, a client application container, enterprise bean, JavaServer page (JSP) files, servlet, or other Java 2 Platform, Enterprise Edition (J2EE) resources. See “Example: Programmatic logins” on page 699 for more information about object request broker (ORB) security initialization requirements in a Java pure client.

User interface prompt programmatic login

WebSphere Application Server also provides a user interface implementation of the `javax.security.auth.callback.CallbackHandler` implementation to collect authentication data from user through user interface login prompts. This callback handler, `com.ibm.websphere.security.auth.callback.WSGUICallbackHandlerImpl`, presents a user interface login panel to prompt users for authentication data.

```

javax.security.auth.login.LoginContext lc = null;

try {
lc = new javax.security.auth.login.LoginContext("WSLogin",
new com.ibm.websphere.security.auth.callback.WSGUICallbackHandlerImpl());

// create a LoginContext and specify a CallbackHandler implementation
// CallbackHandler implementation determine how authentication data is collected
// in this case, the authentication date is collected by GUI login prompt
// and pass to the authentication mechanism implemented by the LoginModule.
} catch (javax.security.auth.login.LoginException e) {
System.err.println("ERROR: failed to instantiate a LoginContext and the exception: "
+ e.getMessage());
e.printStackTrace();
}

```

```

// may be javax.security.auth.AuthPermission "createLoginContext" is not granted
// to the application, or the JAAS login configuration is not defined.
}

if (lc != null)
try {
lc.login(); // perform login
javax.security.auth.Subject s = lc.getSubject();
// get the authenticated subject

// Invoke a J2EE resources using the authenticated subject
com.ibm.websphere.security.auth.WSSubject.doAs(s,
new java.security.PrivilegedAction() {
public Object run() {
try {
bankAccount.deposit(100.00); // where bankAccount is a protected enterprise bean
} catch (Exception e) {
System.out.println("ERROR: error while accessing EJB resource, exception: "
+ e.getMessage());
e.printStackTrace();
}
return null;
}
});
} catch (javax.security.auth.login.LoginException e) {
System.err.println("ERROR: login failed with exception: " + e.getMessage());
e.printStackTrace();

// login failed, might want to provide relogin logic
}

```

Attention: Do not use the `com.ibm.websphere.security.auth.callback.WSGUICallbackHandlerImpl` callback handler for server-side resources (like enterprise bean, servlet, JSP file, or any other server side resources). The user interface login prompt blocks the server for user input. This behavior is not desirable for a server process.

Stdin prompt programmatic login

WebSphere Application Server also provides a stdin implementation of the `javax.security.auth.callback.CallbackHandler` interface to collect authentication data from a user through stdin, which is called `com.ibm.websphere.security.auth.callback.WSStdinCallbackHandlerImpl`. This callback handler prompts a user for authentication data.

```

javax.security.auth.login.LoginContext lc = null;

try {
lc = new javax.security.auth.login.LoginContext("WSLogin",
new com.ibm.websphere.security.auth.callback.WSStdinCallbackHandlerImpl());

// create a LoginContext and specify a CallbackHandler implementation
// CallbackHandler implementation determine how authentication data is collected
// in this case, the authentication date is collected by stdin prompt
// and pass to the authentication mechanism implemented by the LoginModule.
} catch (javax.security.auth.login.LoginException e) {
System.err.println("ERROR: failed to instantiate a LoginContext and the exception:

```

```

        " + e.getMessage());
e.printStackTrace();

// may be javax.security.auth.AuthPermission "createLoginContext" is not granted
// to the application, or the JAAS login configuration is not defined.
}

if (lc != null)
try {
lc.login(); // perform login
javax.security.auth.Subject s = lc.getSubject();
// get the authenticated subject

// Invoke a J2EE resource using the authenticated subject
com.ibm.websphere.security.auth.WSSubject.doAs(s,
new java.security.PrivilegedAction() {
public Object run() {
try {
bankAccount.deposit(100.00);
// where bankAccount is a protected enterprise bean
} catch (Exception e) {
System.out.println("ERROR: error while accessing EJB resource, exception: "
+ e.getMessage());
e.printStackTrace();
}
return null;
}
});
} catch (javax.security.auth.login.LoginException e) {
System.err.println("ERROR: login failed with exception: " + e.getMessage());
e.printStackTrace();

// login failed, might want to provide relogin logic
}

```

Do not use the `com.ibm.websphere.security.auth.callback.WSStdinCallbackHandlerImpl` callback handler for server side resources (like enterprise beans, servlets, JSP files, and so on). The input from the stdin prompt is not sent to the server environment. Most servers run in the background and do not have a console. However, if the server does have a console, the stdin prompt blocks the server for user input. This behavior is not desirable for a server process.

Developing programmatic logins with the Java Authentication and Authorization Service

Java Authentication and Authorization Service represents the strategic application programming interfaces (API) for authentication.

JAAS replaces the CORBA programmatic login APIs

WebSphere Application Server provides some extension to JAAS:

- Refer to the “Developing applications that use CosNaming (CORBA Naming interface)” on page 784 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 custom JAAS login configuration, verify that it is properly defined. See the "Configuring application logins for Java Authentication and Authorization Service" article in the information center 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. See "Adding the `was.policy` file to applications," "Using PolicyTool to edit policy files" and "Configuring the `was.policy` file" articles in the information center for details. For more details on which APIs are protected by Java 2 Security permissions, check the IBM Application Developer Kit, Java Technology Edition; JAAS and WebSphere Application Server public APIs Javadoc in "Security: Resources for learning" in the information center. Some of the APIs used in the sample code in this documentation and the Java 2 Security permissions required by these APIs follow:
 - `javax.security.auth.login.LoginContext` constructors are protected by `javax.security.auth.AuthPermission "createLoginContext"`
 - `javax.security.auth.Subject.doAs()` and `com.ibm.websphere.security.auth.WSSubject.doAs()` are protected by `javax.security.auth.AuthPermission "doAs"`
 - `javax.security.auth.Subject.doAsPrivileged()` and `com.ibm.websphere.security.auth.WSSubject.doAsPrivileged()` are protected by `javax.security.auth.AuthPermission "doAsPrivileged"`
- **Enhanced model to 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 associated with the thread of execution inside a `java.security.AccessController.doPrivileged()` code block. This can present an inconsistent behavior, which might have undesirable effects. The `com.ibm.websphere.security.auth.WSSubject` provides a workaround to associate a Subject to a thread of execution. The `com.ibm.websphere.security.auth.WSSubject` extends the JAAS model to J2EE resources for authorization checks. If the Subject associates with the thread of execution 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 resources authorization checks.
- **User Interface support for defining new JAAS login configuration.** You can configure JAAS login configuration in the administrative console and store it in the WebSphere Common Configuration Model. Applications can define a new JAAS login configuration in the administrative console and the data is persisted in the configuration repository (stored in the WebSphere Common Configuration Model). However, WebSphere Application Server still supports the default JAAS login configuration format (plain text file) provided by the JAAS default implementation. If there are duplication login configurations defined in both the WebSphere Common Configuration and the plain text file format, the one in the WebSphere Common Configuration takes precedence. There are advantages to defining the login configuration in the WebSphere Common Configuration:
 - UI support in defining JAAS login configuration
 - JAAS configuration login configuration can be managed centrally
 - JAAS configuration login configuration is distributed in a Network Deployment installation
- **Application support for programmatic authentication.** WebSphere Application Server provides JAAS login configurations for applications to perform programmatic authentication to the WebSphere security run time. These configurations perform authentication to the WebSphere-configured authentication mechanism (Simple WebSphere Authentication Mechanism (SWAM) or Lightweight Third Party Authentication (LTPA)) and user registry (Local OS, Lightweight Directory Access Protocol (LDAP) or Custom) based on the authentication data supplied. The authenticated Subject from these JAAS login configurations contains the required Principal and Credentials that the WebSphere security run time can use to perform authorization checks on J2EE role-based protected resources. Here are the JAAS login configurations provided by the WebSphere Application Server:
 - **WSLogin JAAS login configuration.** A generic JAAS login configuration can use Java clients, client container applications, servlets, JSP files, and EJB components to perform authentication based on a user ID and password, or a token to the WebSphere security run time. However, this does not honor the `CallbackHandler` specified in the client container deployment descriptor.
 - **ClientContainer JAAS login configuration.** This JAAS login configuration honors the `CallbackHandler` specified in the client container deployment descriptor. The login module of this

login configuration uses the CallbackHandler in the client container deployment descriptor if one is specified, even if the application code specified one CallbackHandler in the LoginContext. This is for a client container application.

A Subject authenticated with the previously mentioned JAAS login configurations contains a `com.ibm.websphere.security.auth.WSPincipal` principal and a `com.ibm.websphere.security.cred.WSCredential` credential. If the authenticated Subject is passed in `com.ibm.websphere.security.auth.WSSubject.doAs()` or the other `doAs()` methods, the product security run time can perform authorization checks on J2EE resources based on the Subject `com.ibm.websphere.security.cred.WSCredential`.

- **Customer-defined JAAS login configurations.** You can define other JAAS login configurations to perform programmatic authentication to your authentication mechanism. See the "Configuring application logins for Java Authentication and Authorization Service" article for details. For the product security run time 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 executing 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 that you are authenticating to. This 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 770.

See the article, "Example: Programmatic logins."

Example: Programmatic logins: The following 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", "realm", "password"));
} catch (LoginException le) {
    System.out.println("Cannot create LoginContext. " + le.getMessage());
    // insert error processing code
} catch (SecurityException se) {
    System.out.println("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
}
```

As shown in the example, the new `LoginContext` is initialized with the `WSLogin` login configuration and the `WSCallbackHandlerImpl` `CallbackHandler`. Use the `WSCallbackHandlerImpl` instance on a server-side application where prompting is not desirable. A `WSCallbackHandlerImpl` instance is initialized by the specified user ID, password, and realm information. The present `WSLoginModuleImpl` class implementation that is specified by `WSLogin` can only retrieve authentication information from the specified `CallbackHandler`. You can construct a `LoginContext` with a `Subject` object, but the `Subject` is disregarded by the present `WSLoginModuleImpl` implementation. For product client container applications, replace `WSLogin` by `ClientContainer` login configuration, which specifies the `WSCClientLoginModuleImpl` implementation that is tailored for client container requirements.

For a pure Java application client, the product provides two other `CallbackHandler` 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 `CallbackHandler` implementations depending on the particular application environment. You can develop a new `CallbackHandler` if neither of these implementations fit your particular application requirement.

You also can develop your own `LoginModule` if the default `WSLoginModuleImpl` implementation fails to meet all your requirements. This product provides utility functions that the custom `LoginModule` can use, which are described in the next section.

In cases where there is no `java.naming.provider.url` set as a system property or in the `jndi.properties` file, a default `InitialContext` does not function if the product server is not at the `localhost:2809` location. In this situation, perform a new `InitialContext` programmatically ahead of the JAAS login. JAAS needs to know where the `SecurityServer` resides to verify that the user ID or password entered is correct, prior to doing a `commit()`. By performing a new `InitialContext` in the way specified below, the security code has the information needed to find the `SecurityServer` location and the target realm.

Attention: The first line starting with `env.put` was split into two lines because it extends beyond the width of the printed page.

```
...
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.println("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
}
```

Custom login module development for a system login configuration

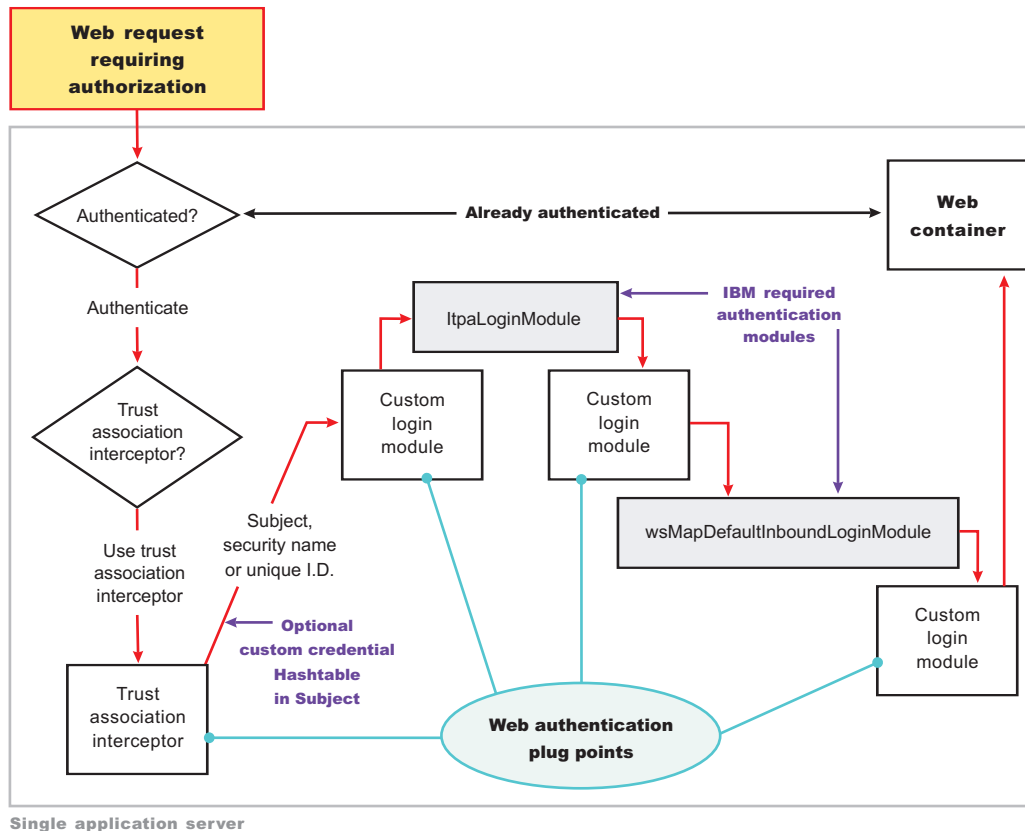
For WebSphere Application Server, there are multiple Java Authentication and Authorization Service (JAAS) plug in points for configuring system logins. WebSphere Application Server uses system login configurations to authenticate incoming requests, outgoing requests, and internal server logins. Application login configurations are called by Java 2 Platform, Enterprise Edition (J2EE) applications for obtaining a Subject 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 are 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 of the login modules combined in Figure 1. Figure 1 shows where you can plug in custom login modules and where the ItpaLoginModule and wsMapDefaultInboundLoginModule are required.

Figure 1

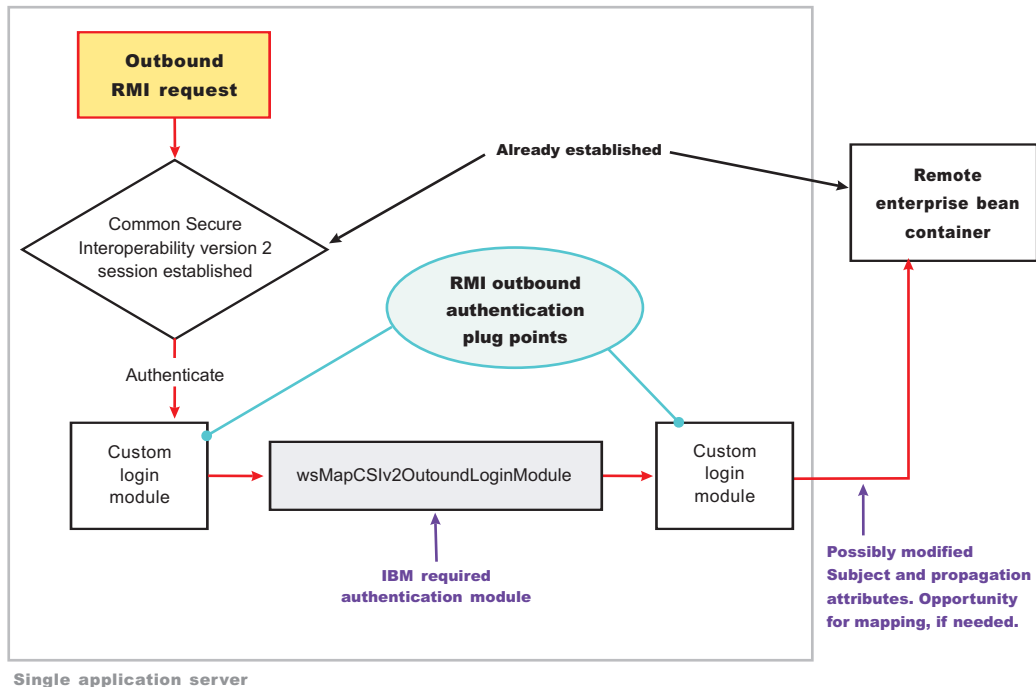


For more detailed information on the WEB_INBOUND configuration including its associated callbacks, see "RMI_INBOUND, WEB_INBOUND, DEFAULT" in "System login configuration entry settings for Java Authentication and Authorization Service" in the information center.

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 Subject passed in (the invocation Subject) and other security context information such as PropagationTokens. 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 wsMapCSlv2OutboundLoginModule is required.

Figure 2

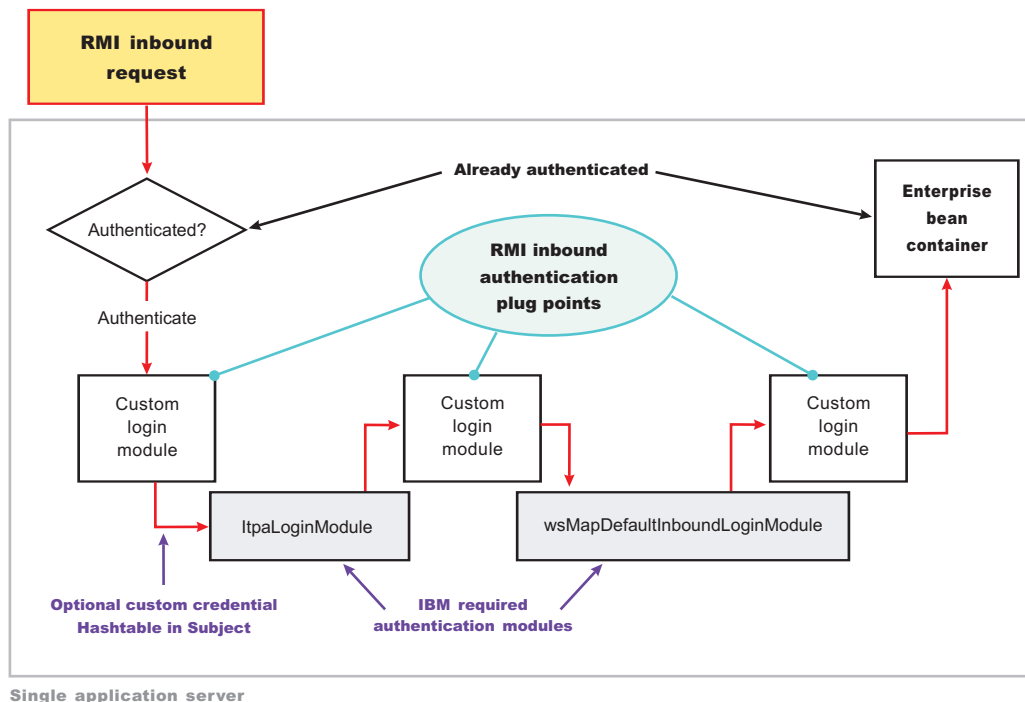


For more information on the RMI_OUTBOUND login configuration including its associated callbacks, see "RMI_OUTBOUND" in "System login configuration entry settings for Java Authentication and Authorization Service" in the information center.

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 de-serialize the information 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, 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 ItpaLoginModule and wsMapDefaultInboundLoginModule are required.

Figure 3



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" in the information center.

DEFAULT login configuration

The DEFAULT login configuration is a plug point that handles all of the other types of authentication requests, including administrative Simple Object Access Protocol (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" in the information center.

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 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. Thus, 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 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 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 might not have sufficient attributes for a full login. Thus, check the requiresLogin() method on the WSTokenHolderCallback 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 is a list of the callbacks that might be present in the system login configurations. The list includes a description of their responsibility.

Callback	Description
<pre>callbacks[0] = new javax.security. auth.callback.NameCallback ("Username: ");</pre>	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.
<pre>callbacks[1] = new javax.security. auth.callback.PasswordCallback ("Password: ", false);</pre>	This callback handler collects the password for the login.
<pre>callbacks[2] = new com.ibm. websphere.security.auth.callback. WSCredTokenCallbackImpl ("Credential Token: ");</pre>	This callback handler collects the Lightweight Third Party Authentication (LTPA) token, or other token type, for the login. It is typically present when a user name and password is not present.
<pre>callbacks[3] = new com.ibm. wsspi.security.auth.callback. WSTokenHolderCallback ("Authz Token List: ");</pre>	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.
<pre>callbacks[4] = new com.ibm. websphere.security.auth.callback. WSServletRequestCallback ("HttpServletRequest: ");</pre>	This callback handler collects the HTTP servlet request object, if present. It enables login modules to get information from the HTTP request for use in the login. This callback handler is presented from the WEB_INBOUND login configuration only.

Callback	Description
<pre>callbacks[5] = new com.ibm. websphere.security.auth.callback. WSServletResponseCallback ("HttpServletResponse: ");</pre>	<p>This callback handler collects the HTTP servlet response object, if present. It 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 to the response. This callback handler is presented from the WEB_INBOUND login configuration only.</p>
<pre>callbacks[6] = new com.ibm. websphere.security.auth.callback. WSAppContextCallback ("ApplicationContextCallback: ");</pre>	<p>This callback handler collects the Web application context used during the login. It consists of a HashMap, which contains the application name and the redirect URL, if present. This callback handler is presented from the WEB_INBOUND login configuration only.</p>

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 an abort 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. You should associate a `java.util.Hashtable` with this property. This hashtable contains properties 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.

Login module in which variables are set:

`ItpaLoginModule`, `swamLoginModule`, and `wsMapDefaultInboundLoginModule`

Shared state variable

`com.ibm.wsspi.security.auth.callback.Constants.WSPRINCIPAL_KEY`

Purpose

Specifies the `com.ibm.websphere.security.auth.WSPPrincipal` object. See the WebSphere Application Server Javadoc 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.

Login module in which variables are set:

ItpaLoginModule, 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 Javadoc 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 wsMapDefaultInboundLoginModule. The information set here is propagated downstream and available to the Application. See the WebSphere Application Server Javadoc 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 is present and contains sufficient information from a sending server to recreate all the required objects needed by WebSphere Application Server run time. In cases where there is sufficient information for WebSphere Application Server run time, the information you might add to the Subject is likely exists from the previous login. To verify if your object is present, you can get access to the ArrayList present in the WSTokenHolderCallback, 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 run time might add additional information at the end of the name to know which Subject set to add the custom object after de-serialization.

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."

Sample code

```
// 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.

```

public customLoginModule()
{
    // Defines your login module variables
    com.ibm.wsspi.security.token.AuthenticationToken customAuthzToken = null;
    com.ibm.wsspi.security.token.AuthenticationToken defaultAuthzToken = null;
    com.ibm.wsspi.security.cred.WSCredential credential = null;
    com.ibm.websphere.security.auth.WSPPrincipal 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 > JAAS Configuration > System logins**. For more information on using the wsadmin utility for system login configuration, see “Example: Customizing a server-side Java Authentication and Authorization Service authentication and login configuration.” Also refer to the “Example: Customizing a server-side Java Authentication and Authorization Service authentication and login configuration” article for information on system login modules and to determine whether to add additional login modules.

Example: Customizing 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 WSCredential and WSPincipal 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 located in the “System login configuration entry settings for Java Authentication and Authorization Service” article in the information center.

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**. Under Authentication, click **JAAS Configuration > System logins**.

Refer to the following code sample to configure a system login configuration using the wsadmin tool. The following sample JAAS 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 were split onto two lines because of the width of the printed page.

```

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
      $entry {{moduleName
        "com.ibm.ws.security.common.auth.module.proxy.WSLoginModuleProxy"}}]
33.   set newPropertyId [$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 LoginModule before the AuthenLoginModule, delete the AuthenLoginModule and then recreate it after inserting the custom LoginModule. Save the sample script into a file, sample.jacl, executing 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 LoginModules:

```

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 were split onto two or more lines because of the width of the printed page. The two lines of code for line 122 are normally one continuous line. The three lines of code for line 124 are normally one continuous line. Also, the three lines of code for line 126 are normally one continuous line.

```

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 ItpaLoginModule and AuthenLoginModule use the shared state to save state information so that custom LoginModules can modify the information. The ItpaLoginModule initializes the callback array in the login() method using the following code. The callback array is created by ItpaLoginModule only if an array is not defined in the shared state area. In the following code sample, the error handling code was removed to make the sample concise. If you insert a custom LoginModule before the ItpaLoginModule, custom LoginModule might follow the same style to save the callback into the shared state.

Attention: In the following code sample, several lines of code have been split onto two lines because of the width of the printed page. Each of these split lines are one continuous line.

```

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.     }

```

ItpaLoginModule and AuthenLoginModule generate both a WSPincipal and a WSCredential object to represent the authenticated user identity and security credentials. The WSPincipal 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 LoginModule, which is inserted after the ItpaLoginModule and the AuthenLoginModule, can modify the WSPincipal and WSCredential objects before they are committed. The WSCredential and WSPincipal 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 when it is used to make any security decisions.

AuthenLoginModule uses the following code to initialize the callback array:

Attention: In the following code sample, several lines of code have been split onto two lines because of the width of the printed page. Each of these split lines are one continuous line.

```

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: a java.util.Map, a HttpServletRequest, and a 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 URL. 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 have been split onto two lines because of the width of the printed page. Each of these split lines are one continuous line.

```
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 `LoginModule` to perform mapping functions. The error page of the form-based login might be modified by a custom `LoginModule`. 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 `LoginModule`. 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 thread of execution during the request processing. When a Web or EJB module is configured to use the caller identity, the user identity is propagated to the downstream service in an EJB request. `WSCredential` 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 thus cannot be modified. For more information on the `WSSubject`, see “Example: Getting the Caller Subject from the Thread.”

Example: Getting the Caller Subject from the Thread

The Caller subject (or “received subject”) contains the user authentication information used in the call for this request. This subject is returned after issuing the `WSSubject.getCallerSubject()` API to prevent replacing existing objects. The subject is marked read-only. This API can be used to get access to the `WSCredential` (documented in the Javadoc information) 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` is propagated downstream (and a new caller subject 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 execute this 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` (documented in the Javadoc information) so that you can put or set data in the hashmap within the credential.

Note: Most data within the Subject is not propagated downstream to another server. Only the credential token within the `WSCredential` 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 following 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 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 execution thread.

Gives flexibility for associating the Subject with all remote calls on this thread whether using a `WSSubject.doAs()` to associate the subject with the remote action or not. 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. The following Java Command Language (JACL) revokes a user when given the realm and user ID, and cycles through all security administration MBean instances returned for the entire cell when run from the Deployment Manager WSADMIN. The command also purges the user from the cache during each process.

Note: This procedure can be called from another JACL script.

Attention: In some of the following lines of code, the lines have been split onto two or more lines.

```

proc revokeUser {realm userid} {
global AdminControl AdminConfig

    if {[catch {$AdminControl queryNames WebSphere:type=SecurityAdmin,*}
result]} {
puts stdout "\$AdminControl queryNames WebSphere:type=SecurityAdmin,*
caught an exception $result\n"
return
} else {
if {$result != {}} {
foreach secBean $result {
if {$secBean != {} || $secBean != "null"} {
if {[catch {$AdminControl invoke $secBean
purgeUserFromAuthCache "$realm $userid"} result]} {
puts stdout "\$AdminControl invoke $secBean

```

```

        purgeUserFromAuthCache $realm $userid caught an
        exception $result\n"
            return
        } else {
            puts stdout "\nUser $userid has been purged from the
            cache of process $secBean\n"
        }
        } else {
            puts stdout "unable to get securityAdmin Mbean, user
            $userid not revoked"
        }
    }
} else {
    puts stdout "Security Mbean was not found\n"
    return
}
    }
    return true
}

```

Developing your own J2C principal mapping module

You can develop your own J2C mapping module if your application requires more sophisticated mapping functions. The mapping LoginModule that you might have developed on WebSphere Application Server Version 5 is still supported in WebSphere Application Server Version 6. The Version 5 LoginModules can be used in the connection factory mapping configuration (that is, they can be defined on the resource). They also can also be used in the resource manager connection factory reference mapping configuration. A Release 5 mapping LoginModule is not able to take advantage of the custom mapping properties.

If you want to develop a new mapping LoginModule in Version 6, use the programming interface described in the following sections.

Migrate your Version 5 mapping LoginModule to use the new programming model to take advantage of the new custom properties as well as the mapping configuration isolation at application scope. Note that mapping LoginModules developed using the WebSphere Application Server Release 6 cannot be used at the deprecated resource connection factory mapping configuration.

Resource Reference Mapping LoginModule invocation

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.

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,
        UnsupportedOperationException;
}

```

WSMappingCallbackHandler can handle two new callback types defined in Release 6:

```

com.ibm.wsspi.security.auth.callback.WSManagedConnectionFactoryCallback
com.ibm.wsspi.security.auth.callback.WSMappingPropertiesCallback

```

The two Callback types should be used by new LoginModules that are used at the resource manager connection factory reference mapping configuration. The WSManagedConnectionFactoryCallback provides a ManagedConnectionFactory instance that should be set in the PasswordCredential. It allows a ManagedConnectionFactory instance to determine whether a PasswordCredential instance is used for sign-on to the target EIS instance. The WSMappingPropertiesCallback provides a HashMap that contains custom mapping properties. The property name "com.ibm.mapping.authDataAlias" is reserved for setting the authentication data alias.

The WebSphere Application Server Release 6 WSMappingCallbackHandle continues to support the two WebSphere Application Server Release 5 Callback types that can be used by older mapping LoginModules. The two Callbacks defined below can only be used by LoginModules that are used by login configuration at the connection factory. For backward compatibility, WebSphere Application Server Release 6 passes the authentication data alias, if defined in the list of custom properties under the "com.ibm.mapping.authDataAlias" property name, via the WSAuthDataAliasCallback to Release 5 LoginModules:

```

com.ibm.ws.security.auth.j2c.WSManagedConnectionFactoryCallback
com.ibm.ws.security.auth.j2c.WSAuthDataAliasCallback

```

Connection Factory Mapping LoginModule Invocation

The WSPrincipalMappingCallbackHandler class handles two Callback types: WSManagedConnectionFactoryCallback and WSMappingPropertiesCallback:

```

com.ibm.wsspi.security.auth.callback.WSManagedConnectionFactoryCallback
com.ibm.wsspi.security.auth.callback.WSMappingPropertiesCallback

```

The WSPrincipalMappingCallbackHandler and the two Callbacks are deprecated in WebSphere Application Server Release 6 and should not be used by new development work.

Mapping LoginModule Resource Reference Mapping Properties

You can pass arbitrary custom properties to your mapping LoginModule. The following example shows how the WebSphere Application Server default mapping LoginModule 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 WebSphere Application Server Version 6 default mapping LoginModule requires one mapping property to define the authentication data alias. The property name, `MAPPING_ALIAS`, is defined in the `Constants.class` in the `com.ibm.wsspi.security.auth.callback` package.

```
MAPPING_ALIAS = "com.ibm.mapping.authDataAlias"
```

When you specify the **Use default method > Select authentication data entry authentication** method 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 choose to create your own custom login configuration and then use the default mapping LoginModule, you'll have to 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` in the principal set and a `WSCredential` 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` or an `org.ietf.jgss.GSSCredential` instance in the set of private credentials.

The `GenericCredential` interface that was defined in Java Cryptography Architecture (JCA) Spec Version 1.0 has been removed in the JCA Version 1.5 spec. The `GenericCredential` interface is supported by WebSphere Application Server Version 6 to support older resource adapters that might have been programmed to the `GenericCredential` interface.

Developing custom user registries

WebSphere Application Server security supports the use of custom registries in addition to Local OS and Lightweight Directory Access Protocol (LDAP) registries for authentication and authorization purposes. A custom user registry is a customer implemented user registry which implements the `UserRegistry` Java interface as provided by WebSphere Application Server. A custom implemented user registry can support virtually any type or notion of an accounts repository from a relational database, flat file, and so on. The custom user registry provides considerable flexibility in adapting WebSphere Application Server security to various environments where some notion of a user registry, other than LDAP or LocalOS, already exist in the operational environment.

Implementing a custom user registry is a software development effort. Use the methods defined in the `UserRegistry` interface to make calls to the desired registry to obtain user and group information. The interface defines a very general set of methods, for encapsulating a wide variety of registries. You can configure a custom user registry as the active user registry when configuring WebSphere Application Server global security.

Make sure that your implementation of the custom registry does not depend on any WebSphere Application Server components such as data sources, enterprise beans, and so on. Do 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 will eliminate the dependency. For example, if your previous implementation used data sources to connect to a database, use Java DataBase Connectivity (JDBC) to connect to the database.

For backward compatibility, the WebSphere Application Server Version 4 custom registry is also supported. Refer to "Migrating custom user registries" in the information center for more information on migrating. If

your previous implementation uses data sources to connect to a database, change the implementation to use JDBC connections. However, it is recommended that you use the new interface to implement your custom registry.

1. If not familiar with the custom user registry concept, refer to the article, "Custom user registries," in the information center. This section explains each of the methods in the interface in detail and the changes for these methods from the version 4 release.
2. 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.
3. Build your implementation. You need the `%install_root%/lib/sas.jar` and `%install_root%/lib/wssec.jar` files in your class path. For example: `%install_root%\java\bin\javac -classpath %install_root%\lib\wssec.jar;%install_root%\lib\sas.jar yourImplementationFile.java`.
4. Copy the class files generated in the previous step to the product class path. The preferred location is the `%install_root%/lib/ext` directory. This should be copied to all the product processes (cell, all NodeAgents) class path.
5. Follow the steps in "Configuring custom user registries" in the information center to configure your implementation using the administrative console. This step is required to implement custom user registries in Version 5.x or later.

If you enabling security, make sure you complete the remaining steps. Once this is done, make sure you save and synchronize the configuration and restart all the servers. Try accessing some J2EE resources to verify that the custom registry implementation is successful.

Example: Custom user registries: A custom user registry is a customer-implemented user registry that implements the `UserRegistry` Java interface as provided by WebSphere Application Server. A custom-implemented user registry can support virtually any type or form of an accounts repository from a relational database, flat file, and so on. The custom user registry provides considerable flexibility in adapting WebSphere Application Server security to various environments where some form of a user registry, other than Lightweight Directory Access Protocol (LDAP) or Local OS, already exist in the operational environment.

Implementing a custom user registry is a software development effort. You must use the methods defined in the `UserRegistry` interface to make calls to the desired registry for obtaining user and group information. The interface defines a very general set of methods, so it can encapsulate a wide variety of registries. You can configure a custom user registry as the active user registry when configuring the product global security.

If you are using the WebSphere Application Server Version 4.x `CustomRegistry` interface, you can plug in your registry without any changes. However, using the new interface to implement your custom registry is recommended.

To view a sample custom registry, refer to the following files:

- `FileRegistrySample.java` file
- `users.props` file
- `groups.props` file

UserRegistry interface methods: Implementing this interface enables WebSphere Application Server security to use custom registries. This capability should extend the `java.rmi` file. With a remote registry, you can complete this process remotely.

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 void initialize(java.util.Properties props)
    throws CustomRegistryException,
           RemoteException;
```

This method is called to initialize the UserRegistry method. All the properties defined in the Custom User Registry panel propagate to this method.

For the sample, the initialize method retrieves the names of the registry files containing the user and group information.

This method is called during server bring up to initialize the registry. This method is also called when validation is performed by the administrative console, when security is on. This method remains the same as in version 4.x.

```
public String checkPassword(String userSecurityName, String password)
    throws PasswordCheckFailedException,
           CustomRegistryException,
           RemoteException;
```

The checkPassword method is called to authenticate users when they log in using a name (or user ID) and a password. This method returns a string which, in most cases, is the user being authenticated. Then, a credential is created for the user for authorization purposes. This user name is also returned for the enterprise bean call, `getCallerPrincipal()`, and the servlet calls, `getUserPrincipal()` and `getRemoteUser()`. See the `getUserDisplayName` method for more information if you have display names in your registry. In some situations, if you return a user other than the one who is logged in, verify that the user is valid in the registry.

For the sample, the `mapCertificate` method gets the distinguished name (DN) from the certificate chain and makes sure it is a valid user in the registry before returning the user. For the sample, the `checkPassword` method checks the name and password combination in the registry and (if they match) returns the user being authenticated.

This method is called for various scenarios. It is called by the administrative console to validate the user information once the registry is initialized. It is also called when you access protected resources in the product for authenticating the user and before proceeding with the authorization. This method is the same as in version 4.x.

```
public String mapCertificate(X509Certificate[] cert)
    throws CertificateMapNotSupportedException,
```

```
CertificateMapFailedException,  
CustomRegistryException,  
RemoteException;
```

The `mapCertificate` method is called to obtain a user name from an X.509 certificate chain supplied by the browser. The complete certificate chain is passed to this method and the implementation can validate the chain if needed and get the user information. A credential is created for this user for authorization purposes. If browser certificates are not supported in your configuration, you can throw the exception, `CertificateMapNotSupportedException`. The consequence of not supporting certificates is authentication failure if the challenge type is certificates, even if valid certificates are in the browser.

This method is called when certificates are provided for authentication. For Web applications, when the authentication constraints are set to CLIENT-CERT in the `web.xml` file of the application, this method is called to map a certificate to a valid user in the registry. For Java clients, this method is called to map the client certificates in the transport layer, when using the transport layer authentication. Also, when the Identity Assertion Token (when using the CSIV2 authentication protocol) is set to contain certificates, this method is called to map the certificates to a valid user.

In WebSphere Application Server Version 4.x, the input parameter was the `X509Certificate`. In WebSphere Application Server Version 5.x and later, this parameter changes to accept an array of `X509Certificate` certificates (such as a certificate chain). In version 4.x, this parameter was called only for Web applications, but in version 5.x and later you can call this method for both Web and Java clients.

```
public String getRealm()  
    throws CustomRegistryException,  
           RemoteException;
```

The `getRealm` method is called to get the name of the security realm. The name of the realm identifies the security domain for which the registry authenticates users. If this method returns a null value, a default name of `customRealm` is used.

For the sample, the `getRealm` method returns the string, `customRealm`. One of the calls to this method is when the registry information is validated. This method is the same as in version 4.x.

```
public Result getUsers(String pattern, int limit)  
    throws CustomRegistryException,  
           RemoteException;
```

The `getUsers` method returns the list of users from the registry. The names of users depend on the pattern parameter. The number of users are limited by the limit parameter. In a registry that has many users, getting all the users is not practical. So the limit parameter is introduced to limit the number of users retrieved from the registry. A limit of 0 indicates to return all the users that match the pattern and might cause problems for large registries. Use this limit with care.

The custom registry implementations are expected to support at least the wildcard search (*). For example, a pattern of (*) returns all the users and a pattern of (b*) returns the users starting with *b*.

The return parameter is an object of type `com.ibm.websphere.security.Result`. This object contains two attributes, a `java.util.List` and a `java.lang.Boolean`. The list contains the users returned and the Boolean flag indicates if there are more users available in the registry for the search pattern. This Boolean flag is used to indicate to the client whether more users are available in the registry.

In the sample, the `getUsers` retrieves the required number of users from the registry and sets them as a list in the result object. To find out if there are more users than requested, the sample gets one more user

than requested and if it finds the additional user, it sets the Boolean flag to true. For pattern matching, the match method in the RegExpSample class is used, which supports wildcard characters such as the asterisk (*) and question mark (?).

This method is called by the administrative console to add users to roles in the various map users to roles panels. The administrative console uses the Boolean set in the result object to indicate that more entries matching the pattern are available in the registry.

In WebSphere Application Server Version 4.x, this method specifies to take only the pattern parameter. The return is a list. In WebSphere Application Server Version 5.x or later, this method is changed to take one additional parameter, the limit. Ideally, your implementation should change to take the limit value and limit the users returned. The return is changed to return a result object, which consists of the list (as in version 4) and a flag indicating if more entries exist. So, when the list returns, use the Result.setList(List) to set the List in the result object. If there are more entries than requested in the limit parameter, set the Boolean attribute to true in the result object, using Result.setHasMore() method. The default for the Boolean attribute in the result object is false.

```
public String getUserDisplayName(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

The getUserDisplayName method returns a display name for a user, if one exists. The display name is an optional string that describes the user that you can set in some registries. This is a descriptive name for the user and does not have to be unique in the registry.

For example in Windows systems, you can display the full name of the user.

If you do not need display names in your registry, return null or an empty string for this method.

Note: In WebSphere Application Server Version 4.x, if display names existed for any user these names were useful for the EJB method call getCallerPrincipal() and the servlet calls getUserPrincipal() and getRemoteUser(). If the display names were not the same as the security name for any user, the display names are returned for the previously mentioned enterprise beans and servlet methods. Returning display names for these methods might become problematic in some situations because the display names might not be unique in the registry. Avoid this problem by changing the default behavior to return the user's security name instead of the user's display name in this version of the product. However, if you want to have the same behavior as in Version 4, set the property WAS_UseDisplayName to true in the **Custom Registry Properties** panel in the administrative console. For more information on how to set properties for the custom registry, see the section on *Setting Properties for Custom Registries*.

In the sample, this method returns the display name of the user whose name matches the user name provided. If the display name does not exist this returns an empty string.

This method can be called by the product to present the display names in the administrative console, or using the command line using the wsadmin tool. Use this method only for displaying. This method is the same as in Version 4.0.

```
public String getUniqueUserId(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method returns the unique ID of the user given the security name.

In the sample, this method returns the uniqueId of the user whose name matches the supplied name. This method is called when forming a credential for a user and also when creating the authorization table for the application.

```
public String getUserSecurityName(String uniqueUserId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method returns the security name of a user given the unique ID. In the sample, this method returns the security name of the user whose unique ID matches the supplied ID.

This method is called to make sure a valid user exists for a given uniqueUserId. This method is called to get the security name of the user when the uniqueUserId is obtained from a token.

```
public boolean isValidUser(String userSecurityName)
    throws CustomRegistryException,
           RemoteException;
```

This method indicates whether the given user is a valid user in the registry.

In the Sample, this method returns true if the user is found in the registry, otherwise this method returns false. This method is primarily called in situations where knowing if the user exists in the directory prevents problems later. For example, in the mapCertificate call, once the name is obtained from the certificate if the user is found to be an invalid user in the registry, you can avoid trying to create the credential for the user.

```
public Result getGroups(String pattern, int limit)
    throws CustomRegistryException,
           RemoteException;
```

The getGroups method returns the list of groups from the registry. The names of groups depend on the pattern parameter. The number of groups is limited by the limit parameter. In a registry that has many groups, getting all the groups is not practical. So, the limit parameter is introduced to limit the number of groups retrieved from the registry. A limit of 0 implies to return all the groups that match the pattern and can cause problems for large registries. Use this limit with care. The custom registry implementations are expected to support at least the wildcard search (*). For example, a pattern of (*) returns all the users and a pattern of (b*) returns the users starting with *b*.

The return parameter is an object of type `com.ibm.websphere.security.Result`. This object contains two attributes, a `java.util.List` and a `java.lang.Boolean`. The list contains the groups returned and the Boolean flag indicates whether there are more groups available in the registry for the pattern searched. This Boolean flag is used to indicate to the client if more groups are available in the registry.

In the sample, the getUsers retrieves the required number of groups from the registry and sets them as a list in the result object. To find out if there are more groups than requested, the sample gets one more user than requested and if it finds the additional user, it sets the Boolean flag to true. For pattern matching, the match method in the RegExpSample class is used. It supports wildcards like *, ?.

This method is called by the administrative console to add groups to roles in the various map groups to roles panels. The administrative console will use the boolean set in the Result object to indicate that more entries matching the pattern are available in the registry.

In WebSphere Application Server Version 4, this method is used to take the pattern parameter only and returns a list. In WebSphere Application Server Version 5.x or later, this method is changed to take one additional parameter, the limit. Change to take the limit value and limit the users returned. The return is

changed to return a result object, which consists of the list (as in version 4) and a flag indicating whether more entries exist. Use the `Result.setList(List)` to set the list in the result object. If there are more entries than requested in the limit parameter, set the Boolean attribute to `true` in the result object using `Result.setHasMore()`. The default for the Boolean attribute in the result object is `false`.

```
public String getGroupDisplayName(String groupSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

The `getGroupDisplayName` method returns a display name for a group if one exists. The display name is an optional string describing the group that you can set in some registries. This name is a descriptive name for the group and does not have to be unique in the registry. If you do not need to have display names for groups in your registry, return `null` or an empty string for this method.

In the sample, this method returns the display name of the group whose name matches the group name provided. If the display name does not exist, this method returns an empty string.

The product can call this method to present the display names in the administrative console or through command line using the `wsadmin` tool. This method is only used for displaying.

```
public String getUniqueGroupId(String groupSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method returns the unique ID of the group given the security name.

In the sample, this method returns the unique ID of the group whose name matches the supplied name. This method is called when creating the authorization table for the application.

```
public List getUniqueGroupIds(String uniqueUserId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method returns the unique IDs of all the groups to which a user belongs.

In the sample, this method returns the unique ID of all the groups that contain this `uniqueUserID`. This method is called when creating the credential for the user. As part of creating the credential, all the `groupUniqueIds` in which the user belongs are collected and put in the credential for authorization purposes when groups are given access to a resource.

```
public String getGroupSecurityName(String uniqueGroupId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method returns the security name of a group given its unique ID.

In the sample, this method returns the security name of the group whose unique ID matches the supplied ID. This method verifies that a valid group exists for a given `uniqueGroupId`.

```
public boolean isValidGroup(String groupSecurityName)
    throws CustomRegistryException,
           RemoteException;
```

This method indicates if the given group is a valid group in the registry.

In the sample, this method returns true if the group is found in the registry, otherwise the method returns false. This method can be used in situations where knowing whether the group exists in the directory might prevent problems later.

```
public List getGroupsForUser(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method returns all the groups to which a user belongs whose name matches the supplied name. This method is similar to the `getUniqueGroupIds` method with the exception that the security names are used instead of the unique IDs.

In the sample, this method returns all the group security names that contain the `userSecurityName`.

This method is called by the administrative console or the scripting tool to verify that the users entered for the RunAs roles are already part of that role in the users and groups to role mapping. This check is required to ensure that a user cannot be added to a RunAs role unless that user is assigned to the role in the users and groups to role mapping either directly or indirectly (through a group that contains this user). Since a group in which the user belongs can be part of the role in the users and groups to role mapping, this method is called to check if any of the groups that this user belongs to mapped to that role.

```
public Result getUsersForGroup(String groupSecurityName, int limit)
    throws NotImplementedException,
           EntryNotFoundException,
           CustomRegistryException,
           RemoteException;
```

This method retrieves users from the specified group. The number of users returned is limited by the `limit` parameter. A limit of **0** indicates to return all the users in that group. This method is not directly called by the WebSphere Application Server security component. However, this can be called by other components. For example, this method issued by the process choreographer when staff assignments are modeled using groups. In rare situations, if you are working with a registry where getting all the users from any of your groups is not practical (for example, if there are a large number of users), you can throw the `NotImplementedException` exception for the particular groups. In this case, verify that if the process choreographer is installed (or if it is installed later) the staff assignments are not modeled using these particular groups. If there is no concern about returning the users from groups in the registry, it is recommended that you do not throw the `NotImplemented` exception when implementing this method.

The return parameter is an object of type `com.ibm.websphere.security.Result`. This object contains two attributes, `java.util.List` and `java.lang.boolean`. The list contains the users returned and the Boolean flag, which indicates whether there are more users available in the registry for the search pattern. This Boolean flag indicates to the client whether users are available in the registry.

In the example, this method gets one user more than the requested number of users for a group if the `limit` parameter is not set to **0**. If it succeeds in getting one more user, it sets the Boolean flag to `true`.

In WebSphere Application Server Version 4, this `getUsers` method was mandatory for the product. For WebSphere Application Server Version 5.x or later, this method can throw the exception `NotImplementedException` exception in situations where it is not practical to get the requested set of users. However, this exception should be thrown in rare situations, as other components can be affected. In version 4, this method accepted only the pattern parameter and the returned a list. In version 5, this method accepts one additional parameter, the `limit`. Change your implementation to take the `limit` value and limit the users returned. The return changes to return a result object, which consists of the list (as in

version 4) and a flag indicating whether more entries exist. When the list is returned, use the `Result.setList(List)` method to set the list in the `Result` object. If there are more entries than requested in the `limit` parameter, set the `Boolean` attribute to `true` in the result object using `Result.setHasMore()`. The default for the `Boolean` attribute in the `Result` object is `false`.

Attention: The first two lines of the following code sample is one continuous line.

```
public com.ibm.websphere.security.cred.WSCredential
    createCredential(String userSecurityName)
    throws NotImplementedException,
        EntryNotFoundException,
        CustomRegistryException,
        RemoteException;
```

In this release of the WebSphere Application Server, this method is not called. You can return `null`. In the example, a `null` is returned.

Trust association interceptor support for Subject creation

The new 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. Although the existing interface is still supported, it is being deprecated in a future release.

The new TAI interface supports a multi-phase, negotiated authentication process. For example, some systems require a challenge response protocol back to the client. The two key methods in this new interface are:

Key method name

```
public boolean isTargetInterceptor (HttpServletRequest req)
```

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.

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.

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 used by WebSphere Application Server. If the credentials are invalid, the code throws the `WebTrustAssociationException`, which indicates that the proxy server is not trusted and the request is denied. If the credentials are valid, the code returns a `TAIResult`, which indicates the status of the request processing along with the client identity (Subject and principal name) to be used for authorizing the Web resource.

Key method name

```
public TAIResult negotiateValidateandEstablishTrust (HttpServletRequest req, HttpServletResponse res)
```

Method result

Returns a `TAIResult`, which indicates the status of the request processing. The request object can be queried and the response object can be modified.

The TAIResult class 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 in one of the following ways:

- 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.
- Other values tell WebSphere Application Server to return the TAI output, which is placed into the HttpServletResponse, to the Web client. Typically, the Web client provides additional information and then places another call to the TAI.

The created TAIResults have the following meanings:

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.

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 has determined the user identity. WebSphere Application Server receives the user ID only and then it 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 had 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 the information center. 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 there is an authentication failure. WebSphere Application Server fails the authentication request:

```
//log error message
// ....
throw new WebTrustAssociationFailedException("TAI failed for this reason");
```

There are a few additional methods on the `TrustAssociationInterceptor` interface that are discussed in the Java documentation. These methods are used for initialization, shut down, and for identifying the TAI to WebSphere Application Server.

Assembling secured applications

There are several assembly tools that are graphical user interfaces for assembling enterprise (J2EE) applications. You can use these tools to assemble an application and secure EJB and Web modules in that application. An EJB module consists of one or more beans. You can enforce security at the EJB method level. A Web module consists of one or more Web resources (an HTML page, a JSP file or a servlet). You can also enforce security for each Web resource. You can use an assembly tool to secure an EJB module (Java archive (JAR) file) or a Web module (Web archive (WAR) file) or an application (enterprise archive (EAR) file). You can create an application, an EJB module, or a Web Module and secure them using an assembly tool or development tools like the IBM Rational Application Developer.

1. Secure EJB applications using an assembly tool. For more information, see “Securing enterprise bean applications” on page 735.
2. Secure Web applications using an assembly tool. For more information, see “Securing Web applications using an assembly tool” on page 737.
3. Add users and groups to roles while assembling secured application using an assembly tool. For more information, see “Adding users and groups to roles using an assembly tool” on page 743.
4. Map users to RunAs roles using an assembly tool. For more information, see “Mapping users to RunAs roles using an assembly tool” on page 744.
5. Add the `was.policy` file to applications.
6. Assemble the application components that you just secured using an assembly tool. For more information, see *Assembling applications*.

After securing an application, the resulting `.ear` file contains security information in its deployment descriptor. The EJB module security information is stored in the `ejb-jar.xml` file and the Web module security information is stored in the `web.xml` file. The `application.xml` file of the application EAR file contains all the roles used in the application. The user and group to roles mapping is stored in the `ibm-application-bnd.xmi` file of the application EAR file.

The `was.policy` file of the application EAR contains the permissions granted for the application to access system resources.

This task is required to secure EJB modules and Web modules in an application. This task is also 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 be able to access system resources.

After securing an application, you can install an application using the administrative console. When you install a secured application, refer to the “Deploying secured applications” on page 744 article to complete this task.

Enterprise bean component security

An EJB module consists of one or more beans. You can use development tools such as Rational Web Developer to develop an EJB module. You can also enforce security at the EJB method level.

You can assign a set of EJB methods to a set of one or more roles. When an EJB method is secured by associating a set of roles, grant at least one role in that set so that you can access that method. To exclude a set of EJB methods from being accessed by anyone mark them **excluded**. You can give everyone access to a set of enterprise beans method by clearing those methods. You can run enterprise beans as a different identity (runAs identity) before invoking other enterprise beans.

Securing enterprise bean applications

You can protect enterprise bean methods by assigning security roles to them. Before you assign security roles, you need to know which EJB methods need protecting and how to protect them.

1. In an assembly tool, import your EJB JAR file or an application archive (EAR) file that contains one or more Web modules. For more information, see the Importing EJB files article or the Importing enterprise applications article.
2. In the Project Explorer, click the **EJB Projects** directory and click the name of your application.
3. Right-click the Deployment descriptor and select **Open with > Deployment Descriptor Editor**. If you selected an EJB .jar file, an EJB deployment descriptor editor opens. If you selected an application .ear file, an application deployment descriptor editor opens. To see online information about the editor, press **F1** and click the editor name.
4. Create security roles. You can create security roles at the application level or at the EJB module level. If you create a security role at the EJB module level, the role displays in the application level. If a security role is created at the application level, the role does not appear in all the EJB modules. You can copy and paste one or more EJB module security roles that you create at application level:
 - Create a role at an EJB module level. In an EJB deployment descriptor editor, select the **Assembly** tab. Under **Security Roles**, click **Add**. In the Add Security Role wizard, name and describe the security role; then click **Finish**.
 - Create a role at the application level. In an application deployment descriptor editor, select the **Security** tab. Under the list of security roles, click **Add**. In the Add Security Role wizard, name and describe the security role; then click **Finish**.
5. Create method permissions. Method permissions map one or more methods to a set of roles. An enterprise bean has four types of methods: Home methods, Remote methods, LocalHome methods and Local methods. You can add permissions to enterprise beans on the method level. You cannot add a method permission to an enterprise bean unless you already have one or more security roles defined. For Version 2.0 EJB projects, there is an unchecked option that specifies that the selected methods from the selected beans do not require authorization to execute. To add a method permission to an enterprise bean:
 - a. On the **Assembly** tab of an EJB deployment descriptor editor, under **Method Permissions**, click **Add**. The Add Method Permission wizard opens.
 - b. Select a security role from the list of roles found and click **Next**.
 - c. Select one or more enterprise beans from the list of beans found. You can click **Select All** or **Deselect All** to select or deselect all of the enterprise beans in the list. Click **Next**.
 - d. Select the methods that you want to bind to your security role. The Method Elements page lists all methods associated with the enterprise bean(s). You can click **Apply to All** or **Deselect All** to quickly select or clear multiple methods. It selects only the * method for each bean. Creating a method permission for the exact method signature overrides the default (*) method permission setting. The * method represents all methods within the bean. There are * for each interface as well. By not selecting all of the individual methods in the tree, you can set other permissions on the remaining methods.
 - e. Click **Finish**.After the method permission is created, you can see the new method permission in the tree. Expand the tree to see the bean and methods defined in the method permission.
6. Exclude user access to methods. Users cannot access excluded methods. Any method in the enterprise beans that is not assigned to a role or is not excluded, is deselected during the application installation by the deployer.
 - a. On the **Assembly** tab of an EJB deployment descriptor editor, under **Excludes List**, click **Add**. The Exclude List wizard opens.
 - b. Select one or more enterprise beans from the list of beans found and click **Next**.
 - c. Select one or more of the method elements for the security identity and click **Finish**.

7. Map the security-role-ref and role-name to the role-link. When developing enterprise beans, you can create the security-role-ref element. The security-role-ref element contains only the role-name field. The role-name field determines if the caller is in a specified role(isCallerInRole()) and contains the name of the role that is referenced in the code. Since you create security roles during the assembly stage, the developer uses a *logical rolename* in the **role-name** field and provides enough information in the **description** field for the assembler to map the actual role (role-link). The security-role-ref element is located at the EJB level. Enterprise beans can have zero or more security-role-ref elements.
 - a. On the **Reference** tab of an EJB deployment descriptor editor, under the list of references, click **Add**. The Add Reference wizard opens.
 - b. Select **Security role reference** and click **Next**.
 - c. Name the security role reference, select a security role to link the reference to, describe the security role reference, and click **Finish**.
 - d. Map every role-name used during development to the role (role-link) using the previous steps.
8. Specify the RunAs Identity for enterprise beans components. The RunAs Identity of the enterprise bean is used to invoke the next enterprise beans in the chain of EJB invocations. When the next enterprise beans are invoked, the RunAsIdentity passes to the next enterprise beans for performing an authorization check on the next enterprise bean. If the RunAs Identity is not specified, the client identity is propagated to the next enterprise bean. The RunAs Identity can represent each of the enterprise beans or can represent each method in the enterprise beans.
 - a. On the **Access** tab of an EJB deployment descriptor editor, next to the **Security Identity (Bean Level)** field, click **Add**. The Add Security Identity wizard opens.
 - b. Select the appropriate run as mode, describe the security identity, and click **Next**. Select the **Use identity of caller** mode to instruct the security service to not make changes to the credential settings for the principal. Select the **Use identity assigned to specific role (below)** mode to use a principal that has been assigned to the specified security role for running the bean methods. This association is part of the application binding in which the role is associated with the user ID and password of a user who is granted that role. If you select the **Use identity assigned to specific role (below)** mode , you must specify a role name and role description.
 - c. Select one or more enterprise beans from the list of beans found and click **Next**. If **Next** is unavailable, click **Finish**.
 - d. Optional: On the Method Elements page, select one or more of the method elements for the security identity and click **Finish**.
9. Close the deployment descriptor editor and, when prompted, click **Yes** to save the changes.

After securing an EJB application, the resulting .jar file contains security information in its deployment descriptor. The security information of the EJB modules is stored in the ejb-jar.xml file.

After securing an EJB application using an assembly tool, you can install the EJB application using the administrative console. During the installation of a secured EJB application, follow the steps in the “Deploying secured applications” on page 744 article to complete the task of securing the EJB application.

Web component security

A Web module consists of servlets, JSP files, server-side utility classes, static Web content (HTML, images, sound files, cascading style sheets (CSS)), and client-side classes (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 JSP file can run as different identities (RunAs identity) before invoking another enterprise bean component. All the

secured Web resources require the user to log in by using a configured login mechanism. There are three types of Web login authentication mechanisms: basic authentication, form-based authentication and client certificate-based authentication.

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

Securing Web applications using an assembly tool

There are three types of Web login authentication mechanisms that you can configure on 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.

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 the Importing WAR files article or the Importing enterprise applications.
2. In the Project Explorer, locate your Web application.
3. Right-click the deployment descriptor and select **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 selected Web archive (WAR) file, a Web deployment descriptor editor opens. If you selected an enterprise application (EAR) file, an application deployment descriptor editor opens.
4. Create security roles either at the application level or at 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 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, select 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, select the **Security** tab. Under the list of security roles, click **Add**. In the Add Security Role wizard, name and describe the security role; 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 that opens, you can do the following:
 - 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* means that the application requires no transport guarantees. *Integral* means that data cannot be changes in transit between client and server. And *Confidential* means that 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. Security run time uses the exact

match first to map the incoming URL with URL patterns. If the exact match is not present, the security run time 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 at this stage. The role-name field contains the name of the role that is referenced in the servlet or JSP code to determine if the caller is in a specified role (isUserInRole()). Since 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 (role-link). The Security-role-ref element is at the servlet level. A servlet or 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. There are 5 types of references you can define on this tab:
 - EJB reference
 - Service reference
 - Resource reference
 - Message destination reference
 - Security role reference
 - Resource environment reference
 - b. Under the list of 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 used during development to the role (role-link) using the previous steps. Every role name 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 JSPs**, click **Add**. The Add Servlet or JSP wizard opens.
 - b. Specify the servlet or JavaServer page (JSP) 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. Under **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 used by your Web application.
8. Configure the login mechanism for the Web module. This configured login mechanism applies to all the servlets, JavaServer page (JSP) files and HTML resources in the Web module.
- a. On the **Pages** tab of a Web deployment descriptor editor, under **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 URLs (for example: /login.jsp and /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 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 the “Deploying secured applications” on page 744 article to finish securing the Web application.

Role-based authorization

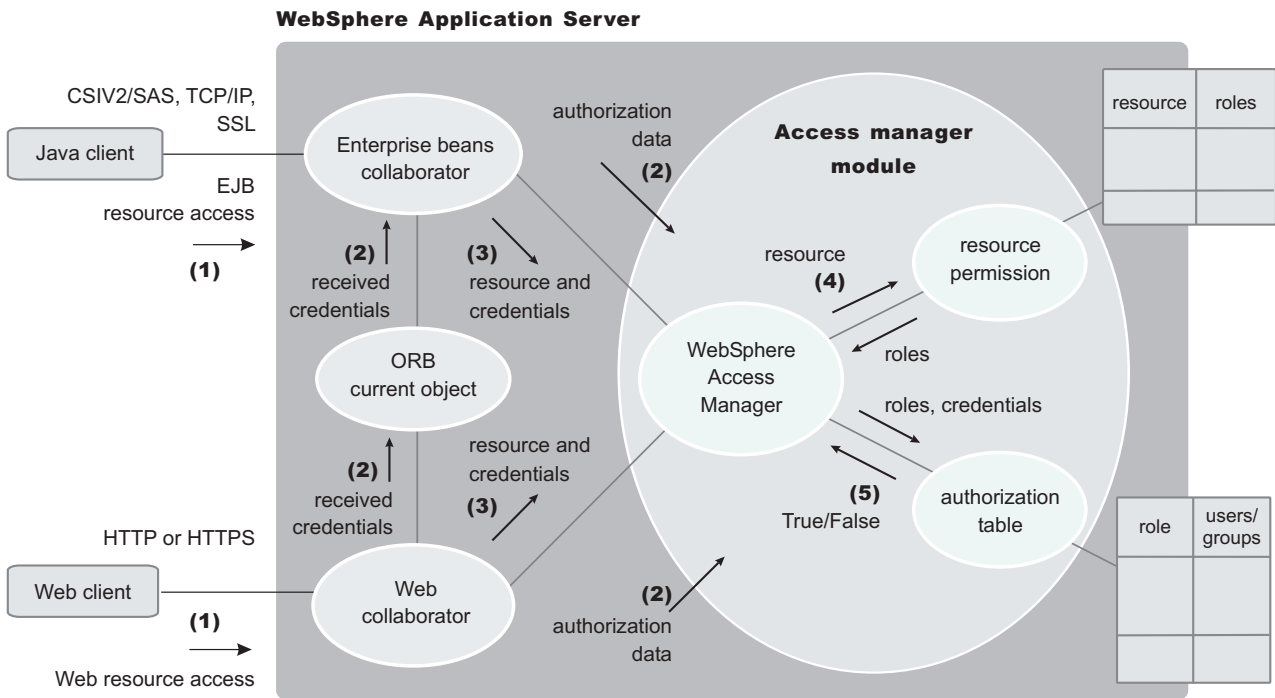
Use authorization information to determine whether a caller has the necessary privileges to request a service.

The following figure illustrates the process used during authorization. Web resource access from a Web client is handled by a Web collaborator. The EJB resource access from a Java client (can be enterprise beans or a servlet) is handled by an EJB Collaborator. The EJB collaborator and the Web collaborator extract the client credentials from the object request broker (ORB) current object. The client credentials are set during the authentication process as received credentials in the ORB Current. The resource and the received credentials are presented to WSAccessManager to check whether access is permitted to the client for accessing the requested resource.

The access manager module contains two main modules:

- Resource permission module helps determine the required roles for a given resource. It uses a resource to roles mapping table that is built by the security run time during application startup. To build the resource-to-role mapping table, the security run time reads the deployment descriptor of the enterprise beans or the Web module (ejb-jar.xml or web.xml)
- Authorization table module consults a role to user or group table to determine whether a client is granted one of the required roles. The role to user or group mapping table, also known as the *authorization table*, is created by the security run time during application startup.
 - To build the authorization table, the security run time reads the application binding file (ibm-application-bnd.xmi file).

Authentication



Use authorization information to determine whether a caller has the necessary privilege to request a service. You can store authorization information many ways. For example, with each resource, you can store an *access-control list*, which contains a list of users and user privileges. Another way to store the information is to associate a list of resources and the corresponding privileges with each user. This list is called a *capability list*.

WebSphere Application Server uses the Java 2 Enterprise Edition (J2EE) authorization model. In this model, authorization information is organized as follows:

- During the assembly of an application, permission to invoke methods is granted to one or more roles. A role is a set of permissions; for example, in a banking application, roles can include teller, supervisor, clerk, and other industry-related positions. The teller role is associated with permissions to run methods related to managing the money in an account, such as the withdraw and deposit methods. The teller role is not granted permission to close accounts; this permission is given to the supervisor role. The application assembler defines a list of method permissions for each role; this list is stored in the deployment descriptor for the application.

There are two *special subjects* that are not defined by the J2EE model, but are worth understanding: AllAuthenticatedUsers and Everyone. A special subject is a product-defined entity independent of the user registry. It is used to generically represent a class of users or groups in the registry.

- AllAuthenticatedUsers is a special subject that permits all authenticated users to access protected methods. As long as the user can authenticate successfully, the user is permitted access to the protected resource.
- Everyone is a special subject that permits unrestricted access to a protected resource. Users do not have to authenticate to get access; this special subject provides access to protected methods as if the resources are unprotected.

During the deployment of an application, real users or groups of users are assigned to the roles. When a user is assigned to a role, the user gets all the method permissions that are granted to that role.

The application deployer does not need to understand the individual methods. By assigning roles to methods, the application assembler simplifies the job of the application deployer. Instead of working with a set of methods, the deployer works with the roles, which represent semantic groupings of the methods.

Users can be assigned to more than one role; the permissions granted to the user are the union of the permissions granted to each role. Additionally, if the authentication mechanism supports the grouping of users, these groups can be assigned to roles. Assigning a group to a role has the same effect as assigning each individual user to the role.

A best practice during deployment is to assign groups, rather than individual users to roles for the following reasons:

- Improves performance during the authorization check. Typically far fewer groups exist than users.
- Provides greater flexibility, by using group membership to control resource access.
- Supports the addition and deletion of users from groups outside of the product environment. This action is preferred to adding and removing them to WebSphere Application Server roles. Stop and restart the enterprise application for these changes to take effect. This action can be very disruptive in a production environment.

At run time, WebSphere Application Server authorizes incoming requests based on the user's identification information and the mapping of the user to roles. If the user belongs to any role that has permission to execute a method, the request is authorized. If the user does not belong to any role that has permission, the request is denied.

The J2EE approach represents a declarative approach to authorization, but it also recognizes that you cannot deal with all situations declaratively. For these situations, methods are provided for determining user and role information programmatically. For Enterprise JavaBeans, the following two methods are supported by WebSphere Application Server:

- **getCallerPrincipal**: This method retrieves the user identification information.
- **isCallerInRole**: This method checks the user identification information against a specific role.

For servlets, the following methods are supported by WebSphere Application Server:

- getRemoteUser
- isUserInRole
- getUserPrincipal

These methods correspond in purpose to the enterprise bean methods.

For more information on the J2EE security authorization model see the following Web site:
<http://java.sun.com>

Admin roles:

The J2EE role-based authorization concept has been extended to protect the WebSphere Application Server administrative subsystem. A number of administrative roles have been defined to provide degrees of authority needed to perform certain WebSphere administrative functions from either the Web-based administrative console or the system management scripting interface. The authorization policy is only enforced when global security is enabled. The following table describes the admin roles:

Admin roles

Role	Description
monitor	Least privileged that basically allows a user to view the WebSphere Application Server configuration and current state.
configurator	Monitor privilege plus the ability to change the WebSphere Application Server configuration.

Admin roles

operator	Monitor privilege plus the ability to change runtime state, such as starting or stopping services for example.
administrator	Operator and configurator privilege, plus additional privileges granted solely to the administrator role. Examples include: <ul style="list-style-type: none">• Modifying the server user ID and password• Mapping users and groups to the administrator role

The identity specified when enabling global security is automatically mapped to the administrator role. Users, groups, can be added or removed from the admin roles from the WebSphere Application Server administrative console at anytime. However, a server restart is required for the changes to take effect. A best practice is to map a group or groups, rather than specific users, to admin roles because it is more flexible and easier to administer in the long run. By mapping a group to an admin 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.

In addition to mapping user or groups, a special-subject can also be mapped to the admin roles. A special-subject is a generalization of a particular class of users. The AllAuthenticated special subject means that the access check of the admin role ensures that the user making the request has at least been authenticated. The Everyone special subject means that anyone, authenticated or not, can perform the action, as if security was not enabled.

Naming roles:

The J2EE role-based authorization concept has been extended to protect the WebSphere 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 WebSphere Name Space. There are generally two ways in which client programs will result in CosNaming calls. The first is through the JNDI interfaces. The second is CORBA clients invoking CosNaming methods directly.

Four security roles are introduced: **CosNamingRead, CosNamingWrite, CosNamingCreate, and CosNamingDelete**. However, the roles now have authority level from low to high as follows:

- **CosNamingRead**. Users who have been assigned the CosNamingRead role will be allowed to do queries of the WebSphere Name Space, such as through the JNDI "lookup" method. The special-subject Everyone is the default policy for this role.
- **CosNamingWrite**. Users who have been assigned the CosNamingWrite role will be allowed to do write operations such as JNDI "bind", "rebind", or "unbind", plus CosNamingRead operations. The special-subject AllAuthenticated is the default policy for this role.
- **CosNamingCreate**. Users who have been assigned the CosNamingCreate role will be allowed to create new objects in the Name Space through such operations as JNDI "createSubcontext", plus CosNamingWrite operations. The special-subject AllAuthenticated is the default policy for this role.
- **CosNamingDelete**. And finally users who have been assigned CosNamingDelete role will be able to destroy objects in the Name Space, for example using the JNDI "destroySubcontext" method, as well as CosNamingCreate operations. The special-subject AllAuthenticated is the default policy for this role.

Users, groups, or the special subjects AllAuthenticated and Everyone can be added or removed to or from the naming roles from the WebSphere Application Server administrative console at anytime. 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 an 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 has been assigned a different naming role, the user will be granted the most permissive access between the role he was assigned and the role his group was assigned. For example, assume that user MyUser has been assigned the CosNamingRead role. Also, assume that group MyGroup has been assigned the CosNamingCreate role. If MyUser is a member of MyGroup, MyUser will be assigned the CosNamingCreate role because he is a member of MyGroup. If MyUser were not a member of MyGroup, he would be assigned the CosNamingRead role.

The CosNaming authorization policy is only enforced when global security is enabled. When global security is enabled, attempts to do CosNaming operations without the proper role assignment will result in a org.omg.CORBA.NO_PERMISSION exception from the CosNaming Server.

In WebSphere Application Server version 4.0.2, each CosNaming function is assigned to only one role. Therefore, users who have been assigned CosNamingCreate role will not be able to query the Name Space unless they have also been assigned CosNamingRead. In most cases a creator would need to be assigned three roles: **CosNamingRead**, **CosNamingWrite**, and **CosNamingCreate**. This has been changed in the release. The **CosNamingRead** and **CosNamingWrite** roles assignment for the creator example in above have been included in **CosNamingCreate** role. In most of the 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 previous one.

Although the ability exist to greatly restrict access to the Name space by changing the default policy, doing so may result in unexpected org.omg.CORBA.NO_PERMISSION exceptions at run time. Typically, J2EE applications access the Name space and the identity they use is that of the user that authenticated to WebSphere Application Server when they access the J2EE application. Unless the J2EE application provider clearly communicates the expected Naming roles, care should be taken when changing the default naming authorization policy.

Adding users and groups to roles using an assembly tool

Before you perform this task, you should have already completed the steps in the “Securing Web applications using an assembly tool” on page 737 and “Securing enterprise bean applications” on page 735 articles where you created new roles and assigned those roles to EJB and Web resources. Complete these steps during application installation. This is because the environment (user registry) under which the application is running is not known until deployment.

If you already know the environment in which the application is running and the user registry that is used, then you can use an assembly tool to assign users and groups to roles. It is recommended that you use the administrative console to assign users and groups to roles.

1. In the Project Explorer view of an assembly tool, right-click an enterprise application project (EAR file) and click **Open With > Deployment Descriptor Editor**. An application deployment descriptor editor opens on the EAR file. To access information about the editor, press F1 and click **Application deployment descriptor editor**.
2. Click the **Security** tab and, under the main pane, click **Add**.
3. In the Add Security Role wizard, name and describe the security role. Then click **Finish**.
4. Under **WebSphere Bindings**, select the user or group extension properties for the security role. Available values include: Everyone, All authenticated users, and Users/Groups.
5. If you selected Users/Groups, click **Add** beside the **Users** or **Groups** panes. In the wizard that opens, specify a user or group name and click **Finish**. Repeat this step until you have added all users and groups to which the security role applies.
6. Close the application deployment descriptor editor and, when prompted, click **Yes** to save the changes.

The `ibm-application-bnd.xml` file in the application contains the users and groups to roles mapping table (*authorization table*).

After securing an application, install the application using the administrative console.

Mapping users to RunAs roles using an assembly tool

RunAs roles are used for delegation. A servlet or enterprise bean component uses the RunAs role to invoke another enterprise bean by impersonating that role.

Before you perform this task:

- Secure the Web application and enterprise bean applications, including creating and assigning new roles to enterprise bean and Web resources. For more information, see “Securing Web applications using an assembly tool” on page 737 and “Securing enterprise bean applications” on page 735.
- Assign users and groups to roles. For more information, see “Adding users and groups to roles using an assembly tool” on page 743. Complete this step during the installation of the application. The environment or user registry under which the application is going to run is not known until deployment. If you already know the environment in which the application is going to run and you know the user registry, then you can use an assembly tool to assign users to RunAs roles.

You must define RunAs roles when a servlet or an enterprise bean in an application is configured with RunAs settings.

1. In the Project Explorer view of an assembly tool, right-click an enterprise application project (EAR file) and click **Open With > Deployment Descriptor Editor**. An application deployment descriptor editor opens on the EAR file. To access information about the editor, press F1 and click **Application deployment descriptor editor**.
2. On the **Security** tab, under **Security Role Run As Bindings**, click **Add**.
3. Click **Add** under **RunAs Bindings**.
4. In the Security Role wizard, select one or more roles and click **Finish**.
5. Repeat steps 3 through 5 for all the RunAs roles in the application.
6. Close the application deployment descriptor editor and, when prompted, click **Yes** to save the changes.

The `ibm-application-bnd.xml` file in the application contains the user to RunAs role mapping table.

After securing an application, you can install the application using the administrative console. You can change the RunAs role mappings of an installed application. For more information, see “RunAs roles to users mapping” on page 754.

Deploying secured applications

Before you perform this task, verify that you have already designed, developed and assembled an application with all the relevant security configurations. For more information on these tasks refer to the “Developing secured applications” on page 673 and “Assembling secured applications” on page 734 articles. In this context, deploying and installing an application are considered the same task.

Deploying applications that have security constraints (secured applications) is not much different than deploying applications any security constraints. The only difference is that you might need to assign users and groups to roles for a secured application, which requires that you have the correct active registry. To deploy a newly secured application click **Applications > Install New Application** in the navigation panel on the left and follow the prompts. If you are installing a secured application, roles would have been defined in the application. If delegation was required in the application, RunAs roles also are defined.

One of the steps required to deploy secured applications is to assign users and groups to roles defined in the application. This task is completed as part of the step titled *Map security roles to users and groups*.

This assignment might have already been done through an assembly tool. In that case you can confirm the mapping by going through this step. You can add new users and groups and modify existing information during this step.

If the applications support delegation, then a RunAs role is already defined in the application. If the delegation policy is set to **Specified Identity** (during assembly) the intermediary invokes a method using an identity setup during deployment. Use the RunAs role to specify the identity under which the downstream invocations are made. For example, if the RunAs role is assigned user "bob" and the client "alice" is invoking a servlet, with delegation set, which in turn calls the enterprise beans, then the method on the enterprise beans is invoked with "bob" as the identity. As part of the deployment process one of the steps is to assign or modify users to the RunAs roles. This step is titled "Map RunAs roles to users". Use this step to assign new users or modify existing users to RunAs roles when the delegation policy is set to Specified Identity.

These steps are common for both installing an application and modifying an existing application. If the application contains roles, you see the "Map security roles to users and groups" link during application installation and also during managing applications, as a link in the Additional properties section.

1. Click **Applications > Install New Application**. Complete the steps (non-security related) required prior to the step titled **Map security roles to users and groups**.
2. Assign users and groups to roles. For more information, see "Assigning users and groups to roles."
3. Map users to RunAs roles if RunAs roles exist in the application. For more information, see "Assigning users to RunAs roles" on page 752.
4. Click **Correct use of System Identity** to specify RunAs roles if needed. Complete this action if the application has delegation set to use System Identity (applicable to enterprise beans only). System Identity uses the WebSphere Application Server security server ID to invoke downstream methods and should be used with caution as this ID has more privileges than other identities in terms of accessing WebSphere Application Server internal methods. This task is provided to make sure that the deployer is aware that the methods listed in the panel have System Identity set up for delegation and to correct them if necessary. If no changes are necessary, skip this task.
5. Complete the remaining (non-security related) steps to finish installing and deploying the application.

Once a secured application is deployed, verify that you can access the resources in the application with the correct credentials. For example, if your application has a protected Web module, make sure only the users that you assigned to the roles are able to use the application.

Assigning users and groups to roles

Before you perform this task:

- Secure the Web applications and EJB applications where new roles were created and assigned to Web and EJB resources.
- Create all the roles in your application.
- Verify that you have properly configured the user registry that contains the users that you want to assign. It is preferable to have security turned on with the user registry of your choice before beginning this process.
- Make sure that if you change anything in the security configuration (for example, enable security or change the user registry) you save the configuration and restart the server before the changes become effective.

Since the default active registry is LocalOS, it is not necessary, although it is recommended, that you enable security if you want to use the LocalOS registry to assign users and groups to roles. You can enable security once the users and groups are assigned in this case. The advantage of enabling security with the appropriate registry before proceeding with this task is that you can validate the security setup (which includes checking the user registry configuration) and avoid any problems using the registry.

These steps are common for both installing an application and modifying an existing application. If the application contains roles, you see the Map security roles to users/groups link during application installation and also during application management, as a link in the Additional properties section at the bottom.

1. Access the administrative console by typing `http://localhost:9060/ibm/console` in a Web browser.
2. Click **Applications > Enterprise applications > *application_name***.
3. Under Additional properties, click **Map security roles to users/groups**. A list of all the roles that belong to this application displays. If the roles already had users or special subjects (All Authenticated, Everyone) assigned, they display here.
4. To assign the special subjects, select either the **Everyone** or the **All Authenticated** check box for the appropriate roles.
5. Click **Apply** to save any changes and then continue working with user or group roles.
6. To assign users or groups, select the role. You can select multiple roles at the same time, if the same users or groups are assigned to all the roles.
7. Click **Look up users** or **Look up groups**.
8. Get the appropriate users and groups from the registry by completing the **limit** (number of items) and the **Search String** fields and clicking **Search**. The **limit** field limits the number of users that are obtained and displayed from the registry. The pattern is a searchable pattern matching one or more users and groups. For example, `user*` lists users like `user1`, `user2`. A pattern of asterisk (*) indicates all users or groups.

Use the limit and the search strings cautiously so as not to overwhelm the registry. When using large registries (like Lightweight Directory Access Protocol (LDAP)) where information on thousands of users and groups resides, a search for a large number of users or groups can make the system very slow and can make it fail. When there are more entries than requests for entries, a message displays on top of the panel. You can refine your search until you have the required list.
9. Select the users and groups to include as members of these roles from the **Available** field and click **>>** to add them to the roles.
10. To remove existing users and groups, select them from the **Selected** field and click **<<**. When removing existing users and groups from roles use caution if those same roles are used as RunAs roles.

For example, if `user1` is assigned to RunAs role, `role1`, and you try to remove `user1` from `role1`, the administrative console validation does not delete the user since a user can only be a part of a RunAs role if the user is already in a role (`User1` should be in `role1` in this case) either directly or indirectly through a group. For more information on the validation checks that are performed between RunAs role mapping and user and group mapping to roles, see the “Assigning users to RunAs roles” on page 752 section.
11. Click **OK**. If there are any validation problems between the role assignments and the RunAs role assignments the changes are not committed and an error message indicating the problem displays at the top of the panel. If there is a problem, make sure that the user in the RunAs role is also a member of the regular role. If the regular role contains a group which contains the user in the RunAs role, make sure that the group is assigned to the role using the administrative console. Follow steps 4 and 5. Avoid using the Application Server Toolkit or any other manual process where the complete name of the group, host name, group name, or distinguished name (DN) is not used.

The user and group information is added to the binding file in the application. This information is used later for authorization purposes.

This task is required to assign users and groups to roles, which enables the correct users and groups to access a secured application. If you are installing an application, complete your installation. Once the application is installed and running you can access your resources according to the user and group mapping you did in this task. If you are managing applications and have modified the users and groups to role mapping, make sure you save, stop and restart the application so that the changes become effective. Try accessing the J2EE resources in the application to verify that the changes are effective.

Security role to user and group mappings:

Use this page to map security roles to users. You can map roles to specific users, to specific groups, or to different categories.

To view this administrative console page, click **Application > Install New Application**. While running the Application Installation Wizard, prompts appear to help you map security roles to users or groups. To change role to user or group mappings for deployed applications, click **Application > Enterprise Application > *deployed_application* > Map security roles to users/groups**.

Users:

Specifies the users for role mapping. Verify that the users are defined in your chosen user registry.

To change the roles to users mapping, click **Manage Application > *application* > Map security roles to users**.

Data type: String

Groups:

Specifies the groups for role mapping. Verify that the groups are defined in your chosen user registry.

To change the roles to users mapping, click **Manage Application > *application* > Map security roles to groups**.

Data type: String

Roles:

Specifies the roles to which you want to map users and groups. Role privileges give users and groups permission to run as specified.

Select the check boxes to choose a role or a set of roles. Click **Look-up Users** to map users to the roles that you have selected. Click **Look-up Groups** to map groups to the selected roles. Use the check boxes to map roles to **EVERYONE** or **ALL AUTHENTICATED** special subject.

Data type: String

Everyone:

Specifies to map roles to everyone. Mapping a role to everyone means that anyone can access resources protected by this role, and essentially, there is no security.

Data type: Boolean

All Authenticated:

Specifies to authenticate all users. Roles are mapped to all authenticated users, and all authenticated users in the selected user registry are granted access to the role.

Data type: Boolean

Security role to user and group selections:

Use this page to select users and groups for security roles.

To view this administrative console page, click **Application > Install New Application**.

While using the Install New Application Wizard, prompts appear to help you map security roles to users. You also can configure security roles to user mappings of deployed applications. Different roles can have different security authorizations. Mapping users or groups to a role authorizes those users or groups to access applications defined by the role. Users, groups and roles are defined when an application is installed or configured.

You also can select role to user and group mappings while you are deploying applications. After deployment in **Additional Properties**, click **Map Security roles to users** to change user and group mappings to a role.

Look up users:

Specifies whether the server looks up selected users.

Choose the role by selecting the check box beside the role and clicking **Lookup users**. Complete the **Limit** and the **Pattern** fields. The **Limit** field contains the number of entries that the search function returns. The **Pattern** field contains the search pattern used for searching entries. For example, bob* searches all users or groups starting with bob. A limit of zero returns all the entries that match the pattern. Use this value only when a small number of users or groups match this pattern in the registry. If the registry contains more entries that match the pattern than requested, a message appears in the console to indicate that there are more entries in the registry. You can either increase the limit or refine the search pattern to get all the entries.

Look up groups:

Specifies whether the server looks up selected groups.

Choose the role by selecting the check box beside the role and clicking **Lookup groups**. Complete the **Limit** and the **Pattern** fields. The **Limit** field contains the number of entries that the search function returns. The **Pattern** field contains the search pattern used for searching entries. For example, bob* searches all users or groups starting with bob. A limit of zero returns all the entries that match the pattern. Use this value only when a small number of users or groups match this pattern in the registry. If the registry contains more entries that match the pattern than requested, a message appears in the console to indicate that there are more entries in the registry. You can either increase the limit or refine the search pattern to get all the entries.

Role:

Specifies user roles.

A number of administrative roles are defined to provide degrees of authority needed to perform certain WebSphere administrative functions from either the Web-based administrative console or the system management scripting interface. The authorization policy is only enforced when global security is enabled. The following roles are valid:

- **Monitor**--least privileged that basically allows a user to view the server configuration and current state
- **Configurator**--monitor privilege plus the ability to change the server configuration
- **Operator**--monitor privilege plus the ability to change the run time state, such as starting or stopping services
- **Administrator**--operator plus configurator privilege

Range

Monitor, Configurator, Operator, Administrator

Everyone:

Specifies to authenticate everyone.

Range Monitor, Configurator, Operator, Administrator

All authenticated:

Range Monitor, Configurator, Operator, Administrator

Mapped users:

Mapped groups:

Look up users and groups settings:

Use this page to select users and groups for security roles.

To view this administrative console page, click **Applications > Enterprise Applications > application_name > Map security roles to users/groups > Look up users or groups** button.

Different roles can have different security authorizations. Mapping users or groups to a role authorizes those users or groups to access applications defined by the role. Users, groups and roles are defined when an application is installed or configured. Use the Search field to display users in the Available Users list. Click the arrows to add users from the Available Users list to the Selected Users list.

Limit:

Specifies the maximum number of users/groups that can be returned when assigning users/groups to roles.

A value of 0 implies a return of all users/groups that match the pattern. You can either increase the limit or refine the search pattern to get all the entries.

Data type	Integer
Units	User name
Default	20
Range	0 or more

Pattern:

Indicates the search pattern used to search for the entries.

The pattern field should contain the search pattern that should be used to search for the entries. For example, bob* will search all users or groups starting with bob. A limit of 0 gets all the entries that match the pattern and should be used only when a small number users/groups match that pattern in the registry. If the registry contains more entries that match the pattern than requested for, a message shows in the console to indicate that there are more entries in the registry.

Data type	String
Units	Number of users
Default	20
Range	A-Z with *

Delegations

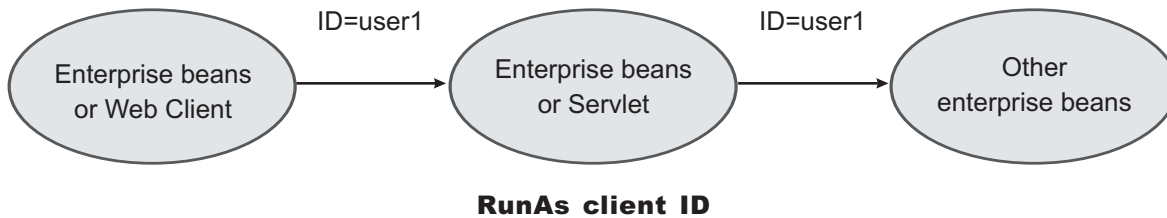
Delegation is a process security identity propagation from a caller to a called object. As per the J2EE specification, a servlet and enterprise beans can propagate either the client (remote user) identity when invoking enterprise beans or they can use another specified identity as indicated in the corresponding deployment descriptor.

The IBM extension supports Enterprise JavaBeans (EJB) to propagate to the server ID when invoking other entity beans. There are three types of delegations:

- Delegate (RunAs) Client Identity
- Delegate (RunAs) Specified Identity
- Delegate (RunAs) System Identity

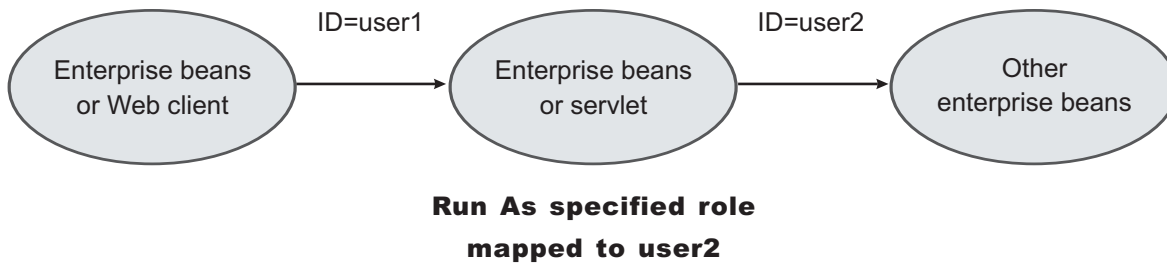
Delegate (RunAs) Client Identity

Delegate Client Identity



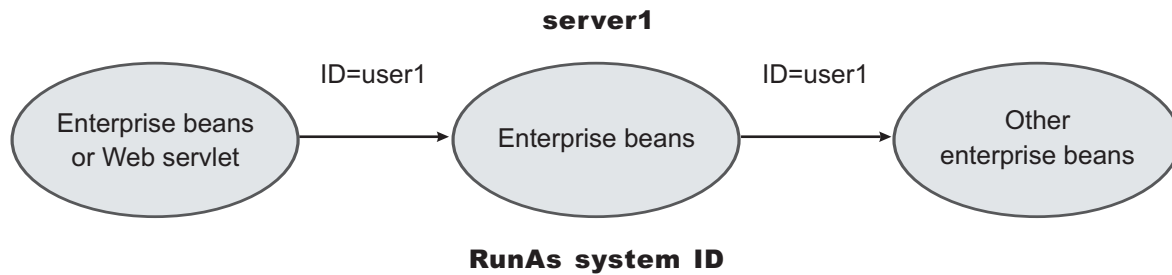
Delegate (RunAs) Specified Identity

Delegate Specified Identity



Delegate (RunAs) System Identity

Delegate System Identity



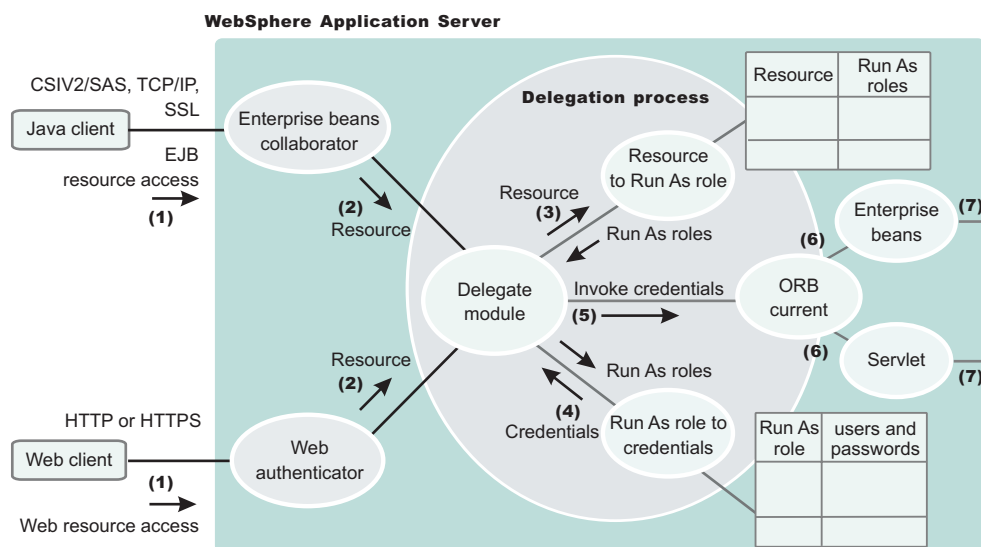
The EJB specification only supports delegation (RunAs) at the EJB level. But an IBM extension allows EJB method level RunAs specification. Method EJB method level runAs specification allows one to specify a different RunAs role for different methods within the same enterprise beans.

The RunAs specification is detailed in the deployment descriptor (the `ejb-jar.xml` file in the EJB module and the `web.xml` file in the Web module). The IBM extension to the RunAs specification is included in the `ibm-ejb-jar-ext.xmi` file.

There is also an IBM specific binding file for each application that contains a mapping from the RunAs role to the user. This file is specified in the `ibm-application-bnd.xmi` file.

These specifications are read by the run time during application startup. The following figure illustrates the delegation mechanism as implemented in the WebSphere Application Server security model.

Delegation



Delegation Process

There are two tables that help in the delegation process:

- Resource to RunAs role mapping table
- RunAs role to user ID and password mapping table

Use the Resource to RunAs role mapping table to get the role that is used by a servlet or by enterprise beans to propagate to the next enterprise beans call.

Use the RunAsRole to User ID and Password mapping table to get the user ID that belongs to the RunAs role and its password.

Delegation is performed after successful authentication and authorization. During this process, the delegation module consults the Resource to RunAs role mapping table to get the RunAs role (3). The delegation module consults the RunAs role to user ID and password mapping table to get the user that belongs to the RunAs role (4). The user ID and password is used to create a new credential using the authentication module, which is not shown in figure. The resulting credential is stored in the ORB Current as an invocation credential (5). Servlet and enterprise beans when invoking other enterprise beans pick up the invocation credential from the ORB Current (6) and call the next enterprise beans (7).

Assigning users to RunAs roles

Before you perform this task,

- Secure the Web applications and EJB applications where new RunAs roles were created and assigned to Web and EJB resources.
- Create all the RunAs roles in your application. The user in the RunAs role can only be entered if that user or a group to which that user belongs is already part of the regular role.
- Assign users and groups to security roles. Refer to “Assigning users and groups to roles” on page 745 for more information.
- Verify that the user registry requirements are met. These requirements are the same as those discussed in the same as in the case of “Assigning users and groups to roles” on page 745 task. For example, if role1 is a role that is also used as a RunAs role, then the user, user1, can be added to the RunAs role. role1, if user1 or a group that user1 belongs to, already is assigned to role1. The administrative console checks this logic when **Apply** or **OK** is clicked. If the check fails, the change is not made and an error message displays at the top of the panel.

If the special subjects “Everyone” or “All Authenticated” are assigned to a role, then no check takes place for that role.

The checking is done every time **Apply** in this panel is clicked or when **OK** is clicked in the **Map security roles to users/groups** panel. The check verifies that all the users in all the RunAs roles do exist directly or indirectly (through a group) in those roles in the **Map security roles to users/groups** panel. If a role is assigned both a user and a group to which that user belongs, then either the user or the group (not both) can be deleted from **Map security roles to users/groups** panel.

If the RunAs role user belongs to a group and if that group is assigned to that role, make sure that the assignment of this group to the role is done through administrative console and not through an assembly tool or any other method. When using the administrative console, the full name of the group is used (for example, hostname\groupName in windows systems, and distinguished names (DN) in Lightweight Directory Access Protocol (LDAP)). During the check, all the groups to which the RunAs role user belongs are obtained from the registry. Since the list of groups obtained from the registry are the full names of the groups, the check works correctly. If the short name of a group is entered using an assembly tool (for example, group1 instead of CN=group1, o=myCompany.com) then this check fails.

These steps are common to both installing an application and modifying an existing application. If the application contains RunAs roles, you see the **Map RunAs roles to users** link during application installation and also during managing applications as a link in the **Additional properties** section at the bottom.

1. Click **Applications > Enterprise Applications > application_name**.
2. Under Additional properties, click **Map RunAs roles to users**. A list of all the RunAs roles that belong to this application displays. If the roles already had users assigned, they display here.
3. To assign a user, select the role. You can select multiple roles at the same time if the same user is assigned to all the roles.

4. Enter the user's name and password in the designated fields. The user name entered can be either the short name (preferred) or the full name (as seen when getting users and groups from the registry).
5. Click **Apply**. The user is authenticated using the active user registry. If authentication is successful, a check is made to verify that this user or group is mapped to the role in the **Map security roles to users and groups** panel. If authentication fails, verify that the user and password are correct and that the active registry configuration is correct.
6. To remove a user from a RunAs role, select the roles and click **Remove**.

The RunAs role user is added to the binding file in the application. This file is used for delegation purposes when accessing J2EE resources. This step is required to assign users to RunAs roles so that during delegation the appropriate user is used to invoke the EJB methods.

If you are installing the application, complete installation. Once the application is installed and running you can access your resources according to the RunAS role mapping. Save the configuration.

If you are managing applications and have modified the RunAs roles to users mapping, make sure you save, stop and restart the application so that the changes become effective. Try accessing your J2EE resources to verify that the new changes are in effect.

Unprotected EJB 2.0 methods protection settings:

Use this page to verify that unprotected EJB 2.0 methods have the correct level of protection before you map users to roles.

To view this administrative console page, click **Application > Install New Application**. While running the Install New Application Wizard, prompts appear to help you map security roles to users.

Exclude:

Specifies that the method is completely protected.

Data type: Check box
Default: Cleared

Uncheck:

Specifies that everyone can access the security method.

Data type: Check box
Default: Uncheck

Specify role:

Specifies the EJB level of protection based on the security role.

The roles listed in this menu are obtained from the application scope. If the selected role is not in the module, then it is added to the modules or Java archive (JAR) files.

Data type: String
Units: Role

Module name:

Specifies the name of the module.

If a module name appears in this list, then the module contains unprotected EJB methods.

Data type: String
Units: Module name

Protection:

Specifies the level of protection assigned to a particular module name.

Data type: String
Default: Cleared

EJB 2.1 method protection level settings:

Use this page to verify that all unprotected EJB 2.1 methods have the correct level of protection before you map users to roles.

To view this administrative console page, click **Applications > Install New Application**. While running the Install New Application Wizard, prompts appear to help you determine that all unprotected EJB 2.1 methods have the correct level of protection.

EJB Module:

Specifies the enterprise bean module name.

Data Type: String
Units: EJB module name

Module URI:

Specifies the Java archive (JAR) file name.

Data Type: String
Units: JAR file name

Method protection:

Specifies the level of protection assigned to the EJB module.

A selected box means to *Deny All* and that the method is completely protected.

Data Type: Check box
Default: Cleared
Range: Yes or No

RunAs roles to users mapping:

Use this page to map RunAs roles to users. You can change the RunAs settings after an application deploys.

To view this administrative console page, click **Applications > Install New Application**. While running the application installation wizard, prompts appear to help you map RunAs roles to users. You can change the RunAs roles to users mappings for deployed applications by completing the following steps:

1. Click **Applications > Enterprise Applications > *application_name***.
2. Under Additional properties, click **Map RunAs roles to users**.

The enterprise beans you are installing contain predefined RunAs roles. RunAs roles are used by enterprise beans that need to run as a particular role for recognition while interacting with another enterprise bean.

User name:

Specifies a user name for the RunAs role user.

This user already maps to the role specified in the Mapping users and groups to roles panel. You can map the user to its appropriate role by either mapping the user to that role directly or mapping a group that contains the user to that role.

Data type: String

Password:

Specifies the password for the RunAs user.

Data type: String

Confirm password:

Specifies the confirmed password of the administrative user.

Data type String

Role:

Specifies administrative user roles.

A number of administrative roles have been defined to provide degrees of authority needed to perform certain WebSphere administrative functions from either the web based administrative console or the system management scripting interface. The authorization policy is only enforced when global security is enabled. The following roles are valid:

- **Monitor**--least privileged that basically allows a user to view the WebSphere configuration and current state
- **Configurator**--monitor privilege plus the ability to change the WebSphere configuration
- **Operator**--monitor privilege plus the ability to change runtime state, such as starting or stopping services for example
- **Administrator**--operator plus configurator privilege

Updating and redeploying secured applications

Before you perform this task, secure Web applications, secure EJB applications, and deploy them in WebSphere Application Server. This section addresses the way to update existing applications.

1. Use the administrative console to modify the existing users and groups mapping to roles. For information on the required steps, see "Assigning users and groups to roles" on page 745.
2. Use the administrative console to modify the users for the RunAs roles. For information on the required steps, see "Assigning users to RunAs roles" on page 752.
3. Complete the changes and save them.
4. Stop and restart the application for the changes to become effective.

5. Use the an assembly tool. For more information, see *Assembling applications*.
6. Use an assembly tool to modify roles, method permissions, auth-constraints, data-constraints and so on. For more information, see *Assembling applications*.
7. Save the Enterprise Archive (EAR) file, uninstall the old application, deploy the modified application and start the application to make the changes effective.

The applications are modified and redeployed. This step is required to modify existing secured applications.

If information about roles is modified make sure you update the user and group information using the administrative console. Once the secured applications are modified and either restarted or redeployed, make sure that the changes are effective by accessing the resources in the application.

Testing security

After configuring global security and restarting all of your servers in a secure mode, it is best to validate that security is properly enabled.

There are a few techniques that you can use to test the various security login types. For example, you can test the Web-based BasicAuth login, Web-based form login, and the Java client BasicAuth login.

There are basic tests that show that the fundamental security components are working properly. Complete the following steps to validate your security configuration:

1. Test the Web-based BasicAuth with *Snoop*, by accessing the following URL:
`http://hostname.domain:9080/snoop`. A login panel appears. If a login panel does not appear, then a problem exists. If the panel appears, type in any valid user ID and password in your configured user registry.

Note: In a Network Deployment environment, the Snoop servlet is only available in the domain if you included the **DefaultApplication** option when adding the application server to the cell. The **-includeapps** option for the **addNode** command migrates the **DefaultApplication** option to the cell. Otherwise, skip this step.
2. Test the Web-based form login by bringing up the administrative console:
`http://hostname.domain:9060/ibm/console`. A form-based login page appears. If a login page does not appear, try accessing the administrative console by typing `https://myhost.domain:9043/ibm/console`. Type in the administrative user ID and password used for configuring your user registry when configuring security.

When the authentication mechanism is set as Lightweight Third Party Authentication (LTPA), represent the host name as a fully qualified host name (that is, `myhost.mycompany.com:9060` rather than just `myhost:9060`).
3. Test Java Client BasicAuth with *dumpNameSpace* by executing the `install_root\bin\dumpNameSpace.bat` file. A login panel appears. If a login panel does not appear, there is a problem. Type in any valid user ID and password in your configured user registry.
4. Thoroughly test all of your applications in secure mode.
5. After enabling security, verify that your system comes up in secure mode.
6. If all tests pass, proceed with more rigorous testing of your secured applications. If you have any problems, review the output logs in the WebSphere Application Server `/logs/nodeagent` or WebSphere Application Server `/logs/server_name` directories, respectively. Then check the security troubleshooting article in the information center to see if it references any common problems.

The results of these tests, if successful, indicate that security is fully enabled and working properly.

Naming and directory

Learn about naming and directory

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Develop and assemble applications that use naming support

- Develop applications that use JNDI
- Develop applications that use the CORBA Naming (CosNaming) interface
- Assemble applications for deployment (same as any application type)

Deploy and administer your applications

- Deploy applications (same as any application type)
- Deploy applications (Education on Demand)
- Configure name space bindings
- Configure name space bindings (Education on Demand)
- Administer applications (same as any application)
- Administer applications (Education on Demand)

Troubleshooting name space bindings

Refer to the *Troubleshooting and support* PDF.

Conceptual overviews

Documentation

"Naming" on page 758

Presentations

Education on Demand offers:

- Naming

See Chapter 4 of the IBM Redbook IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

Samples are not available at this time.

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 application 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. 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 762.
4. 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 762. For more information on configured name bindings, refer to "Configured name bindings" on page 765.

5. 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. For details, refer to the *Troubleshooting and support* PDF.

Naming

Naming is used by clients of WebSphere Application Server applications to obtain references to objects related to those applications, such as Enterprise JavaBeans (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.

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

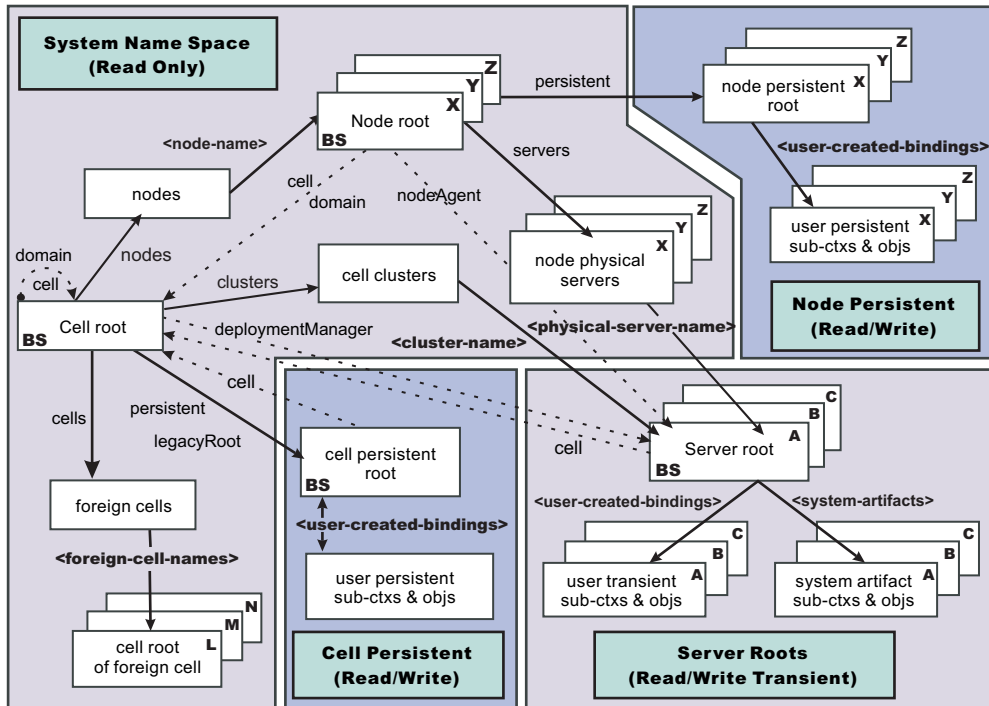


Figure 12. Name Space Logical View

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.

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.

Distributing application objects among many server roots is a departure from previous WebSphere Application Server releases, where all system artifacts were bound under a single root. This change can affect the names that clients use to look up these objects.

Server-scoped bindings are relative to a server's server root.

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.

An important role of the cell persistent root is as the initial context for clients running in previous WebSphere Application Server versions. If you want to access an enterprise bean by WebSphere Application Server v4.0.x and 3.5.x clients, you must ensure that a binding for it has been added to the cell persistent root. You can configure these additional bindings as cell-scoped bindings.

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.

Unlike the cell persistent root, the node persistent root plays no special role in interoperability with WebSphere Application Server clients of previous releases. Node-scoped 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 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.

- **WebSphere Application Server JNDI interface implementation prior to V5**

WebSphere Application Server clients running in releases prior to WebSphere Application Server V5 by default use WebSphere Application Server's v4.0 CosNaming JNDI plug-in implementation. The default initial context for clients of this type is the cell persistent root, also known as the *legacy root*.

- **Other JNDI implementation**

Some applications can perform name space lookups with a non-WebSphere Application Server CosNaming JNDI plug-in implementation. Assuming the key **NamingContext** 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 **NamingContext**. The initial context in this case is the cell root.

Lookup names support in deployment descriptors and thin clients

Server 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 is

a departure from previous versions of WebSphere Application Server, where these objects were all bound under a single root context. This section 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 speaking, 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 Sun's 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 Sun's 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 interoperability with JNDI clients running on previous versions of WebSphere Application Server. Additionally, you can configure cell-scoped bindings to create a fixed qualified name for server objects.

Scope

You can configure a binding at one of the following three scopes: cell, node, or server. 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. If the target server of a server-scoped binding is a cluster, the binding is created under the server root context of each cluster member.

Note: The term *server* includes clusters and can be used interchangeably with the term *cluster* with respect to configured bindings. When applied to a cluster, a server-scoped binding is created in the server root for all member servers.

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 or server, or if you do not want the binding to be associated with any specific node 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 if you want to configure a binding with the same name on different servers which resolve to different objects, a server-scoped binding would be appropriate. Note that two servers 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.

This type of binding is of special significance because you can use it to provide interoperability with WebSphere Application Server v3.5.x and v4.0.x JNDI clients. The default initial context for these earlier clients is the cell persistent root, which is different from the initial context of the server root for WebSphere Application Server V5 JNDI clients. If you migrate an application to the current release, you can configure an EJB binding at the cell scope so that the lookup names for the enterprise bean do not change for clients still running in a earlier WebSphere Application Server version.

A cell-scoped EJB binding is also 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.

Note: WebSphere Application Server v3.5.x clients cannot access this type of binding .

String: String constant

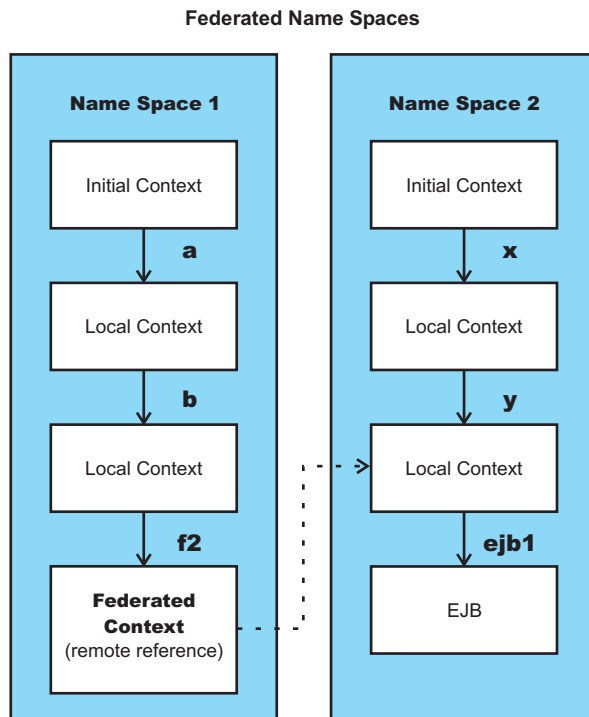
You can configure a binding of a string constant. The following data is required to configure a string constant binding:

- 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 JavaBeans (EJB) home bound under the name ejb1, the EJB home could 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.



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 WebSphere Application Server's initial context factory to obtain the reference to the federated context. If you use some other initial context factory implementation, you either may not be able to create the binding, or the level of transparency may 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 may result. If you want to federate WebSphere Application Server name spaces, you should use servers running under the Network Deployment or Enterprise packages of WebSphere Application Server.

Name space bindings

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 are created each time a server starts, even when the binding is created in a transient partition of the name space. One major use of configured bindings is to provide interoperability with JNDI clients running on previous versions of the WebSphere Application Server.

There are four different kinds of bindings that you can configure:

- Enterprise JavaBeans (EJB)
- CORBA object
- Indirect Lookup
- String

Naming and directories: Resources for learning

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 the information center 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

Programming specifications

- Java Naming and Directory Interface™ 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 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 name space.

These 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 WebSphere's JNDI Context implementation is used.

WebSphere Application Server's 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's initial context factory. To set JNDI caching or name syntax properties which will be visible to WebSphere Application Server's initial context factory, follow the following steps.

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); ...
javadoc.naming.Context initialContext = new javadoc.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 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.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
...
javadoc.naming.Context initialContext = new javadoc.naming.InitialContext(env);

```

Example: Getting the default initial context

This example below gets the default initial context. That is, no provider URL is passed to the `javax.naming.InitialContext` constructor. The following section explains the process of determining the address of the bootstrap server to use to obtain the initial context.

```

...
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. The initial context returned in the various environments are listed below:

- Thin client: The server root context of the server running on the local host at port 2809.
- Pure client:
 - The context specified by the `java.naming.provider.url` property passed to `launchClient` command with the `-CCD` command line parameter. The context usually will be 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 was specified, 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 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, which has changed from previous releases.

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.

• Understanding the 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**

Previous versions of WebSphere Application Server 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 previous releases of WebSphere Application Server, 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. Today, 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:

1. The InitialContext constructor. This does not apply to the above example since the example uses the empty InitialContext constructor.
2. 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.
3. `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 may 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: 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 should only contain the bootstrap addresses of members of the same cluster. Otherwise, incorrect behavior may occur.

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.

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 which 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. The examples below show lookups from each type of application. 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. An example of a lookup with a corbaname URL is also included in this section.

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.

```
// 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 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)javadoc.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)javadoc.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 qualified 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

The example below shows a lookup of a JavaMail resource. The actual lookup name is determined by the application's deployment descriptors.

```

// Get the initial context as shown above
...
Session session =
    (Session) initialContext.lookup("java:comp/env/mail/MailSession");

```

JNDI interoperability considerations

This section explains considerations to take into account when interoperating with WebSphere Application Server V4.0 and with non-WebSphere Application Server JNDI clients. Also, the way resources from MQSeries must be bound to the name space changed after V4.0 and is described below.

Interoperability with WebSphere Application Server V4.0

- **EJB clients running on WebSphere Application Server V4.0 accessing EJB applications running on WebSphere Application Server V5 or V6**

Applications migrated from previous versions of WebSphere Application Server may still have clients still running in a previous release. The default initial JNDI context for EJB clients running on previous versions of WebSphere Application Server is the cell persistent root (legacy root). The home for an enterprise bean deployed in version 5 or 6 is bound to its server's server root context. In order for the EJB lookup name for down-level clients to remain unchanged, configure a binding for the EJB home under the cell persistent root.

- **EJB clients running on WebSphere Application Server V5 or V6 accessing EJB applications running on WebSphere Application Server V4.0 servers**

The default initial context for a WebSphere Application Server V4.0 server is the correct initial context. Simply look up the JNDI name under which the EJB home is bound.

Note: To enable WebSphere Application Server V5 or V6 clients to access version 4.x servers, the down-level installations must have e-fix PQ60074 installed.

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 will be 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, as shown below.

- **Single server**

```
initialContext.lookup(
    "corbaname:iiop:myHost:2809#cell/nodes/node1/servers/server1/myEJB");
```

According to the URL above, the bootstrap host and port are myHost and 2809, and the enterprise bean is installed in a server **server1** in node **node1** and bound in that server under the name **myEJB**.

- **Server cluster**

```
initialContext.lookup(
    "corbaname:iiop:myHost:2809#cell/clusters/myCluster/myEJB");
```

According to the URL above, the bootstrap host and port are **myHost** and **2809**, and the enterprise bean is installed in a server cluster named **myCluster** and bound in that cluster under the name **myEJB**.

The above lookup will work with any name server bootstrap host and port configured in the same cell.

The above lookup will also work if the bootstrap host and port belongs to a member of the cluster itself.

To avoid a single point of failure, the bootstrap server host and port for each cluster member could 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 bootstrapping 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 bootstrap host is normalized to a fully qualified 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

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 which must interoperate with CORBA applications may need to use INS name syntax to represent names in string format. The name syntax property may 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 will 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 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.

Note: To enable WebSphere Application Server Version 6 or 5.x clients to access Version 4.0.x servers, the earlier installations must have e-fix PQ60074 installed.

1. Get an initial context.
2. Perform desired CosNaming operations.

Example: Getting an initial context with CosNaming

In the 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. Invoke a method on the ORB to obtain the initial reference

These steps are now explained in more detail.

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.iop.ORB` is included with the WebSphere Application Server. 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 WebSphere Application Server's ORB class.
3. If the bootstrap server is INS-compliant, set the initial reference properties. If the bootstrap server is not INS-compliant (meaning, WebSphere Application Server v4.0.x or earlier), set bootstrap host and port for bootstrap server.
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.iop.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);
...
...
import java.util.Properties;
import org.omg.CORBA.ORB;
...
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass","com.ibm.ws390.orb.ORB");
props.put("com.ibm.CORBA.ORBInitRef.NameService",
"corbaloc:iiop:myhost.mycompany.com:2809/NameService");
props.put("com.ibm.CORBA.ORBInitRef.NameServiceServerRoot",
"corbaloc:iiop:myhost.mycompany.com:2809/NameServiceServerRoot");
ORB _orb = ORB.init((String[])null, props);
...

```

Notice the initial reference definitions for NameService and NameServiceServerRoot. The initial context returned for NameService depends on the type of bootstrap server. The key NameServiceServerRoot is a key introduced in WebSphere Application Server V5. For more information on initial contexts, see the section Initial Contexts.

Note: The properties com.ibm.CORBA.BootstrapHost and com.ibm.CORBA.BootstrapPort are deprecated. They are needed, however, to connect to WebSphere Application Servers of Version 4.0.x or earlier. The default bootstrap host is the local host and the default port is 2809.

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 WebSphere Application 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, see the section [Initial Contexts](#).

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 WebSphere Application 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 unqualified 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);
```

Configuring and viewing name space bindings

To view or configure an EJB, CORBA, Indirect lookup or string name space binding, complete the following:

1. In the administrative console, click **Environment > Manage Name Space Bindings**.
2. Select the desired scope by entering in a node name for node-scoped bindings, or a node name and server name for server-scoped bindings, and click **Apply**.
3. To create a new binding, click **New** and follow the instructions. To edit a previously created binding, click the binding you want to edit and proceed to the next step.
4. Edit the **Binding identifier**, the **Name in name space**, and the **String value** fields as desired.

Note: All of these fields are required.

5. Click **Finish** to register the changes.

String binding settings

Use this page to configure a new string binding or to view or edit an existing string binding.

To view this administrative console page, click **Environment > Naming > Name Space Bindings > string_namespace_binding**.

Scope:

Shows the scope of the configured binding. This value indicates the configuration location for the `namebindings.xml` file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type:

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier:

Specifies the name that uniquely identifies this configured binding.

Name in Name Space:

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

String Value:

Specifies the string to be bound into the name space.

CORBA object binding settings

Use this page to configure a new name binding of a CORBA object binding, or to view or edit an existing CORBA object binding.

To view this administrative console page, click **Environment > Naming > Name Space Bindings > CORBA_namespace_binding**.

Scope:

Shows the scope of the configured binding. This value indicates the configuration location for the namebindings.xml file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type:

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier:

Specifies the name that uniquely identifies this configured binding.

Name in Name Space:

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

Corbaname URL:

Specifies the CORBA name URL string identifying where the object is bound in a CosNaming server.

Federated Context:

Specifies whether the target is a CosNaming context (true) or a leaf node object (false).

true	The target object is bound with a context CORBA binding type. If the corbaname URL does not resolve to a NamingContext, an error occurs when the binding is first used (which is when the URL is first resolved).
false	The target object is bound with an object CORBA binding type.

Indirect lookup binding settings

Use this page to configure a new indirect lookup name binding, or to view or edit an existing indirect lookup binding.

To view this administrative console page, click **Environment > Naming > Name Space Bindings > indirect_lookup_namespace_binding**.

Scope:

Shows the scope of the configured binding. This value indicates the configuration location for the namebindings.xml file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type:

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier:

Specifies the name that uniquely identifies this configured binding.

Name in Name Space:

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

Provider URL:

Specifies the provider URL string needed to obtain a JNDI initial context.

JNDI Name:

Specifies the name used to look up the target object from the initial context.

EJB binding settings

Use this page to configure a new EJB binding, or to view or edit an existing EJB binding.

To view this administrative console page, click **Environment > Naming > Name Space Bindings > EJB_namespace_binding**.

Scope:

Shows the scope of the configured binding. This value indicates the configuration location for the namebindings.xml file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type:

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier:

Specifies the name that uniquely identifies this configured binding.

Name in Name Space:

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

Enterprise Bean Location:

Specifies whether the enterprise bean is running in a server cluster or a single server. If Single Server is specified, type the node name.

Server:

Specifies the name of the cluster or non-clustered server in which the enterprise bean is configured.

JNDI Name:

Specifies the JNDI name of the deployed enterprise bean (the bean's JNDI name that is in the enterprise bean bindings--not the java:comp name)

Name space binding collection

Use this page to configure a name binding of an EJB, a CORBA CosNaming NamingContext, a CORBA leaf node object, an object that you can look up using JNDI, or a constant string value.

Binding information for configured bindings is stored in the configuration and applied upon startup of the name server for each server within the scope of the binding.

To view the Name Space Bindings page, click **Environment > Naming > Name Space Bindings**.

Click the check boxes to select one or more of the users in your collection. Use the buttons to control the selected users.

Name:

Shows the names given to uniquely identify these configured bindings.

Scope:

Shows the scope of the configured binding. This value indicates the configuration location for the namebindings.xml file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type:

Shows the type of binding configured. Valid values are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Configuring name servers

To configure a name server, complete the following:

1. In the administrative console, click **Servers > Application Servers > Server Components > Name Server**.
2. Edit the fields as desired.

Note: All of these fields are mandatory.

3. To make other changes, click **Custom Properties** and configure a custom property.
4. Click **OK** to register your changes.

Name server settings

Use this page to configure Naming Service Provider settings for the application server.

To view this administrative console page, click **Servers > Application Servers > *server_name* > Administration > Server Components > Name Server**.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers > *server_name* > Administration > Server Components > Name Server**
- **Servers > JMS Servers > *server_name* > Administration > Server Components > Name Server**

Name:

Specifies the display name for the server.

Data type	String
------------------	--------

Initial State:

Specifies the execution state. The options are: *Started* and *Stopped*.

Data type	String
Default	Started

Object Request Broker

Learn about Object Request Brokers (ORB)

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Tune, configure and use the ORB

- Learn how to tune the ORB
- Adjust the ORB timeout to improve handling of network failures
- Configure ORB settings not available in the console
- Use the ORB to convert character code sets
- Have your application communicate with the client-side ORB
- Read the ORB trace
- Troubleshoot ORB problems

Conceptual overviews

Documentation

“Object Request Brokers” on page 794

Tutorials

Tutorials are not available at this time.

Samples

Samples are not available at this time.

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:

- Adjust timeout settings to improve handling of network failures. See “Object Request Broker service settings” on page 796 for more information. Before making these adjustments, read “Object Request Broker tuning guidelines” on page 794.
- Adjust thread-pool settings used by the ORB for handling Internet InterORB Protocol (IIOP) connections. See “Thread pool settings” in the information center.
- If problems with the ORB arise, see “Object request broker component troubleshooting tips” in the information center.

For help in troubleshooting, see “Object Request Broker communications trace” on page 804.

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.

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 tuning guidelines

The following options exist for improving the performance of the Object Request Broker (ORB). Tuning results vary among systems and applications.

- **Logical pool distribution (LPD) mechanism**

Do not configure LPD unless you have already configured it with a previous version of WebSphere Application Server.

If you suspect that requests with longer completion times are elongating the response times for shorter completion time requests by denying them adequate access to threads in the thread pool, LPD provides a mechanism to allow the shorter requests greater access to execution threads.

For more information, see “Logical pool distribution” on page 794

- **ORB timeout**

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

- **com.ibm.CORBA.ConnectTimeout system property**

The `com.ibm.CORBA.ConnectTimeout` property specifies the maximum time in seconds that the client ORB waits before timing out when establishing a new socket connection with a remote server ORB. This property is intended for use by client applications. It is not used by the application server.

You can specify the `com.ibm.CORBA.ConnectTimeout` property in the `orb.properties` file, or you can add the property when running the `launchclient` script. This example specifies a maximum timeout value of ten seconds:

```
launchclient clientapp.ear -CCDcom.ibm.com.CORBA.ConnectTimeout=10...
```

You can 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 may 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.numJNIReaders system property**

You can improve performance by setting the `com.ibm.CORBA.numJNIReaders` system property through a command-line script. This property specifies the number of threads to be shared for request handling when the native selector mechanism is enabled.

Valid Range	0-2147483647
Default	2

- **Determining the ORB message size**

The ORB separates messages into fragments to send over the ORB connection. You can configure this fragment size through the `com.ibm.CORBA.FragmentSize` parameter.

To determine the size of the messages that transfer over the ORB and the number of required fragments:

1. In the administrative console, enable ORB tracing in the ORB Properties page. See “Object Request Broker service settings” for more information.
2. Enable ORBRas tracing from the logging and tracing page.
3. Increase the trace file sizes because tracing can generate a lot of data.
4. Restart the server and run at least one iteration (preferably several) of the case that you are measuring.
5. Look at the traceable file and do a search for `Fragment to follow: Yes`.

This indicates that the ORB transmitted a fragment, but it still has at least one remaining fragment to send before the entire message has arrived. A `Fragment to follow: No` value indicates that the particular fragment is the last in the entire message. This fragment can also be the first, if the message fit entirely into one fragment.

If you go to the spot where `Fragment to follow: Yes` is located, you find a block that looks similar to the following example:

```
Fragment to follow:           Yes
Message size:                4988 (0x137C)
--
Request ID:                   1411
```

This example indicates that the amount of data in the fragment is 4988 bytes and the Request ID is 1411. If you search for all occurrences of `Request ID: 1411`, you can see the number of fragments used to send that particular message. If you add all the associated message sizes, you have the total size of the message that is being sent through the ORB.

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 to 300

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.

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.

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

Connection cache maximum:

Specifies the largest number of supported connections that can occupy the connection cache for the service. If simultaneous clients connect to the server-side ORB, this parameter can be increased up to 1000 clients to support the heavy load.

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	0-255

Connection cache minimum:

Specifies the smallest number of connections to be kept in the connection cache for the ORB service.

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	0-255

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.

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.

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 web address must be a proper format:

`http://w3.mycorp.com:81/servlet/com.ibm.CORBA.services.IIOP TunnelServlet`

For applets: `http://applethost:port/servlet/com.ibm.CORBA.services.IIOP TunnelServlet`.

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 pass by reference, the parameters are copied to the stack before every remote method call is made, which can be expensive.

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 Pass by reference can improve performance up to 50%. Pass by reference helps performance only where non-primitive object types are passed as parameters. Therefore, int and floats are always copied, regardless of the call model.

Enable this property with caution, because unexpected behavior can occur. If an object reference is modified by the remote method, the caller might change.

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 EJB Specification, Version 2.0 (see section 5.4). Object references passed to 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 Pass by reference is enabled. Setting Pass by reference 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 Pass by reference 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.

Object Request Broker custom properties

Use the this page to set and monitor settings associated with the Java Object Request Broker (ORB) service that do not appear on the main settings page by default.

Setting ORB properties through the administrative console

1. In the administrative console, click **Servers >Application servers > server_name >Container services > ORB service >Custom properties** .
2. To add properties to the page, click **New** and enter at least a name (case-sensitive) and a value for the property. Then click **Apply**.
3. When you are finished entering properties, click **OK**.

Setting ORB Properties through the command line

If you use the `java` command, use the `-D` option, for example:

```
java -Dcom.ibm.CORBA.propname1=value1 -Dcom.ibm.CORBA.propname2=value2 ... application name
```

If you use the `launchclient` command, prefix the property with `-CC`, for example (shown on 2 lines for publication):

```
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 the product setup. A list of additional properties associated with the Java ORB service follows:

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 807.

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 807.

Default	2809
----------------	------

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. The setting of this property is valid only for client-side ORBs. Otherwise, the ORB obtains a value at run time by calling `InetAddress.getLocalHost().getHostAddress()` method.

com.ibm.CORBA.ServerSocketQueueDepth:

Corresponds to the length of a TCP/IP stack listen queue and prevents WebSphere Application Server from rejecting requests when space is not available in the listen queue. If several simultaneous clients connect to the server-side ORB, you can increase this parameter to support up to 1000 clients.

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.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: Do not configure logical pool distribution unless you have already configured it with 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 “Logical pool distribution” on page 794.

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.

LPD must be enabled (see `com.ibm.websphere.threadpool.strategy.implementation`).

Data type Integer
Units Milliseconds
Default 30
Range 20,000 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.

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.

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.

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.

LPD must be enabled (see `com.ibm.websphere.threadpool.strategy.implementation`).

Data type	Integer
Default	96
Range	10 minimum

com.ibm.CORBA.numJNIReaders:

You can improve performance by setting the `com.ibm.CORBA.numJNIReaders` system property through a command-line script. This property specifies the number of threads to share for request handling when the native selector mechanism is enabled.

Valid Range	0-2147483647
Default	2

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 IOP 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.

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.websphere.ObjectIDVersionCompatibility:

This property applies when you have a mixed release cluster that has V6 and V5.1.0 or earlier and you are performing an incremental cell upgrade.

In mixed release cells, the migration program sets this property to 1.

After all of the cluster members are upgraded to V6, you can improve performance by removing this property or by setting the value to 2.

Setting the value to 1 indicates that the ORB runs using version 1 object identities, which is required to for mixed cells that contain application servers with releases prior to V5.1.1. In V6, not setting the property or changing the property value to 2 causes the ORB to run using version 2 object identities. Changing to version 2 object identities results in improved performance.

For incremental cell upgrade instructions, see "Migrating a V5.x managed node to a V6 managed node" and "Migrating Network Deployment, V5.x to a V6 deployment manager" in the information center.

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 *install_root/logs/trace* 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 reply example 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 reply example 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 reply example, 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=842115:0=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 more information about the two properties and the order of precedence that the ORB uses to locate services, read the CORBA/IIOP specification, cited in “Object Request Brokers: Resources for learning” on page 810.

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 (split here for publication only):

```
java program -ORBInitRef
             NameService=corbaloc::sample.server.com:2809/NameService
```

After these properties are set for services supported by the ORB, Java 2 Platform, Enterprise Edition (J2EE) applications obtain the initial reference to a given service by calling the `resolve_initial_references` function on the ORB, as defined in the CORBA/IIOP specification.

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 *installation_root/java/ibm_bin* directory, where *installation_root* is the installation directory for the product. 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 *installation_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 *installation_root/java/ibm_bin* directory instead of the JRE standard *installation_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 *installation_root/java/ibm_lib* directory.

Before using either the rmic or idlj tool, ensure that the *installation_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 *installation_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. (In contrast, the default value for versions 3.5.x and 4.0.x of this product was ISO8859_1, also known as Latin-1.) 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	
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	

Java name	OSF registry ID	Comments
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

For more information, read the CORBA and IIOP specification, cited in “Object Request Brokers: Resources for learning”

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” on page 811

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.

Transactions

Learn about transactions

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Develop and assemble applications that use transactions

- Use the transaction service
- Develop components to use transactions
- Configure transactional deployment descriptor attributes for EJB or Web modules
- Enable a session bean, servlet, or application client component to use bean-managed transactions
- Assemble applications for deployment (same as any application type)
- Use one-phase and two-phase commit resources in the same transaction
- Assemble an application to use one-phase and two-phase commit resources in the same transaction

Deploy and administer your applications

- Deploy applications (same as any application)
- Deploy applications (Education on Demand)
- Administer applications (Education on Demand)
- Use local transactions
- Configure transaction properties for an application server
- Manage active and prepared transactions
- Manage transaction logging for optimum server availability
- Interoperate transactionally between application servers
- Configure an application server to log heuristic reporting

Troubleshoot transactions

- Learn about transaction service exceptions

Conceptual overviews

Documentation

"Transaction support in WebSphere Application Server" on page 813

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Last Participant Support and ActivitySessions

See Chapter 9 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

- **JTA extensions - Transaction tracker**

A servlet that registers for synchronization callbacks when it is initialized. The servlet displays a table detailing every completed transaction on the server since registration. For the user's convenience, an option is provided to run a new transaction that can either be committed or rolled back.

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 that 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
- Using local transactions
- Developing a WebSphere application to use transactions
- Configuring transaction properties for an application server
- Managing active transactions
- Managing transaction logging for optimum server availability
- Interoperating transactionally between application servers
- Troubleshooting transactions

- Transaction service exceptions
- UserTransaction interface - methods available
- Coordinating access to 1-PC and 2-PC-capable resources within the same transaction
- Implementing WebSphere enterprise applications that use ActivitySessions

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 (for example, application servers) or Web Service Atomic Transaction (WS-AtomicTransaction) protocol. WebSphere Application Server also participates in transactions imported through J2EE Connector 1.5 resource adapters. WebSphere applications can also be configured interact with (or to direct the WebSphere transaction service to interact with) databases, JMS queues, and JCA connectors through their local transaction support when distributed transaction coordination is not required.

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 transaction.

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, see Editing deployment descriptors.

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 WTRN0105I 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.

Web Services – Atomic Transaction for WebSphere Application Server:

The Web Services - Atomic Transaction for 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 823.

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 13, shows a transaction context shared between two WebSphere application servers for a Web service request that contains a WS-AT CoordinationContext.

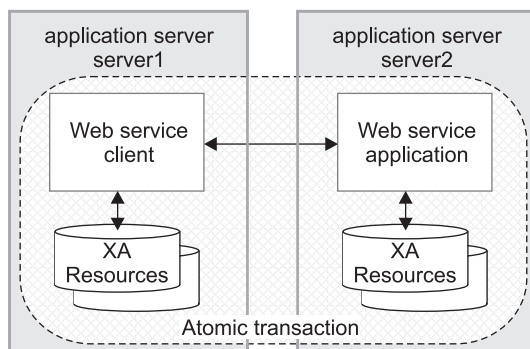


Figure 13. Transaction context shared between two WebSphere application servers.

WS-AT support restrictions

In WebSphere Application Server version 6.0, WS-AT contexts cannot be propagated through firewalls and 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 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 823.

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.

Using 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 unspecified 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.

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 manager 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 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.

If you want a Web component to use transactions, you must write the code that explicitly demarcates the boundaries of a transaction as described in Using component-managed transactions.

Similarly, if you want an application client component to use 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 Assembling applications.

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.
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

Specifies the duration of a local transaction context. You can set this attribute to **Bean method** or **ActivitySession**.

Note: The ActivitySession option is not supported in the web container.

Resolver

Specifies how the resource manager local transaction is to be resolved by the application through user code or by the container. You can set this attribute to either **Application** or **ContainerAtBoundary**.

Unresolved action

Specifies the action that the container must take when the local transaction context scope

ends, if resources are uncommitted by an application in a local transaction and the **Resolution control** is set to Application. You can set this attribute to either **Commit** or **Rollback**.

For a value of **Commit**, the container will take the commit action only in the absence of an un-handled exception. If the application method executing under the local transaction context ends with an exception, then the local transaction context is 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 for WebSphere Application Server” on page 817. 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 for WebSphere Application Server” on page 817. 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 Configuring ActivitySession deployment attributes for an enterprise JavaBean. For more detail about the relationship between these two properties, see Combining transaction and ActivitySession container policies.

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 for WebSphere Application Server” on page 817, 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 necessary 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 816.
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.
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 Deploying and managing applications.

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 to 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 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 ();
    ...
}
...
}
```

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.

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(one_phase)`. 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 use of one-phase and two-phase commit resources within the same transaction
- Assembling an application to use one-phase and two-phase commit resources in the same transaction
- Configuring an application server to allow logging for heuristic reporting

Coordinating access to 1-PC and 2-PC-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.

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(one_phase). 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 *Assembling applications*.

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.
2. Create or edit the application EAR file.

Note: Ensure that you set the target server as WebSphere Application Server version 6.

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 v6.0
 - d. Click **Finish**
3. In the J2EE Hierarchy view, right-click the Enterprise Application instance, then 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.
 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 *Deploying and managing applications*.

Last participant support extension settings:

Use this page to configure last participant support extensions.

Last participant support is an extension to the transaction service to allow a single one-phase resource to participate in a two-phase transaction with one or more two-phase resources.

To view this administrative console page, click **Applications** *application_name* → **Last Participant Support Extension**

Accept Heuristic Hazard:

Specifies whether the application accepts the possibility of an heuristic hazard occurring in a two-phase transaction containing 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 Local Server** This displays the properties of the application 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 WebSphere Application Server application programming interface reference information (Javadoc).

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.

View links to additional information about:

- “Programming specifications”
- “Other”

Programming specifications

- J2EE Activity Service for Extended Transactions
- Java Transaction API (JTA) 1.0.1

Other

- WebSphere Application Server Enterprise Version 5 Overview: Advanced Transactional Connectivity
- Listing of PDF files to learn about WebSphere Application Server Version 5
- Listing of all IBM WebSphere Application Server Redbooks
- Listing of all IBM WebSphere Application Server Whitepapers
- WebSphere Application Server Enterprise Edition 4.0: A Programmer’s Guide

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 2 for an introduction to each WebSphere extension.

ActivitySessions	How do I?...	Overview		Samples
Application profiling	How do I?...	Overview		Samples
Asynchronous beans	How do I?...	Overview		Samples
Dynamic caching	How do I?...	Overview		
Dynamic query	How do I?...	Overview		Samples
Internationalization	How do I?...	Overview		Samples
Object pools	How do I?...	Overview		
Scheduler	How do I?...	Overview		Samples
Startup beans	How do I?...	Overview		
Work areas	How do I?...	Overview		

ActivitySessions

Learn about ActivitySessions

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and “How do I?...” topics, pending their availability.

How do I?...

Develop and assemble applications that use ActivitySessions

- Use ActivitySessions with HTTP sessions
- Develop a J2EE application to use an ActivitySession
- Develop an enterprise bean or J2EE client to manage ActivitySessions
- Set EJB module ActivitySession deployment attributes

Deploy and administer your applications

- Deploy applications (same as any application type)
- Deploy applications (Education on Demand)
- Disable or enable ActivitySessions
- Set the default ActivitySession timeout for an application server
- Administer applications (same as any application)
- Administer applications (Education on Demand)

Troubleshoot ActivitySessions

Refer to the *Troubleshooting and support* PDF.

Conceptual overviews

Documentation

“The ActivitySession service” on page 833

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Last Participant Support and ActivitySessions

See Chapter 9 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

- **Http session association (MasterMind)**

A servlet provides the user interface for a game called MasterMind. The game uses an HTTP session to control the `ActivitySession` lifecycle, and talks to an enterprise bean, which holds the state and provides the logic for the game. The aim of the game is to guess the four-element code that is generated at the start. On each guess, clues are given to the identity of the target code by how many elements in the guess are present in the target and how many of these elements are correctly placed.

- **Container-managed `ActivitySessions`**

This Sample consists of a client, which begins and ends an `ActivitySession`, updating an entity bean. The sample demonstrates client access to the `UserActivitySession` interface, container-managed `ActivitySessions` and container resolution of resource-managed local transactions. These transactions start within the enterprise beans that have a local transaction containment (LTC) boundary of `ActivitySession`. The client verifies that updates to bean instances are committed when the `ActivitySession` is completed with the `EndModeCheckpoint` and rolled back when the `EndModeReset` is used.

- **Bean-managed `ActivitySessions`**

This Sample consists of a client that invokes a method on a stateless session bean. This session bean uses bean-managed `ActivitySessions`, beginning and ending the `ActivitySessions` with the `UserActivitySession` interface. During these `ActivitySessions`, a stateful session bean is accessed. This stateful session bean, which uses container-managed `ActivitySessions`, an LTC boundary of `ActivitySession`, and an LTC resolution control of application, is called several times to update data in a database. Sometimes the stateful session bean is instructed to complete the resource manager local transactions (RMLTs), either to commit them or roll them back. Sometimes the RMLTs are left incomplete. The stateless session bean then completes the `ActivitySession` and reports back to the client whether the results are consistent with the expected behavior.

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 (UOW) scope to that provided by global transaction contexts. `ActivitySessions` provide a scoping mechanism for units of work, and both an `ActivitySession` and a transaction has the same following characteristics:

- It can be bean-managed or container-managed
- It can be distributed across application servers
- It 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
 - `ActivitySessions` and transaction contexts
 - Using `ActivitySessions` with HTTP sessions
- The `ActivitySession` service programming interfaces
- Developing a J2EE application to use an `ActivitySession`
- `ActivitySessions` samples
- Configuring EJB module `ActivitySession` deployment attributes
- Configuring Web module `ActivitySession` deployment attributes
- Disabling or enabling the `ActivitySession` service
- Configuring the default `ActivitySession` timeout
- 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:

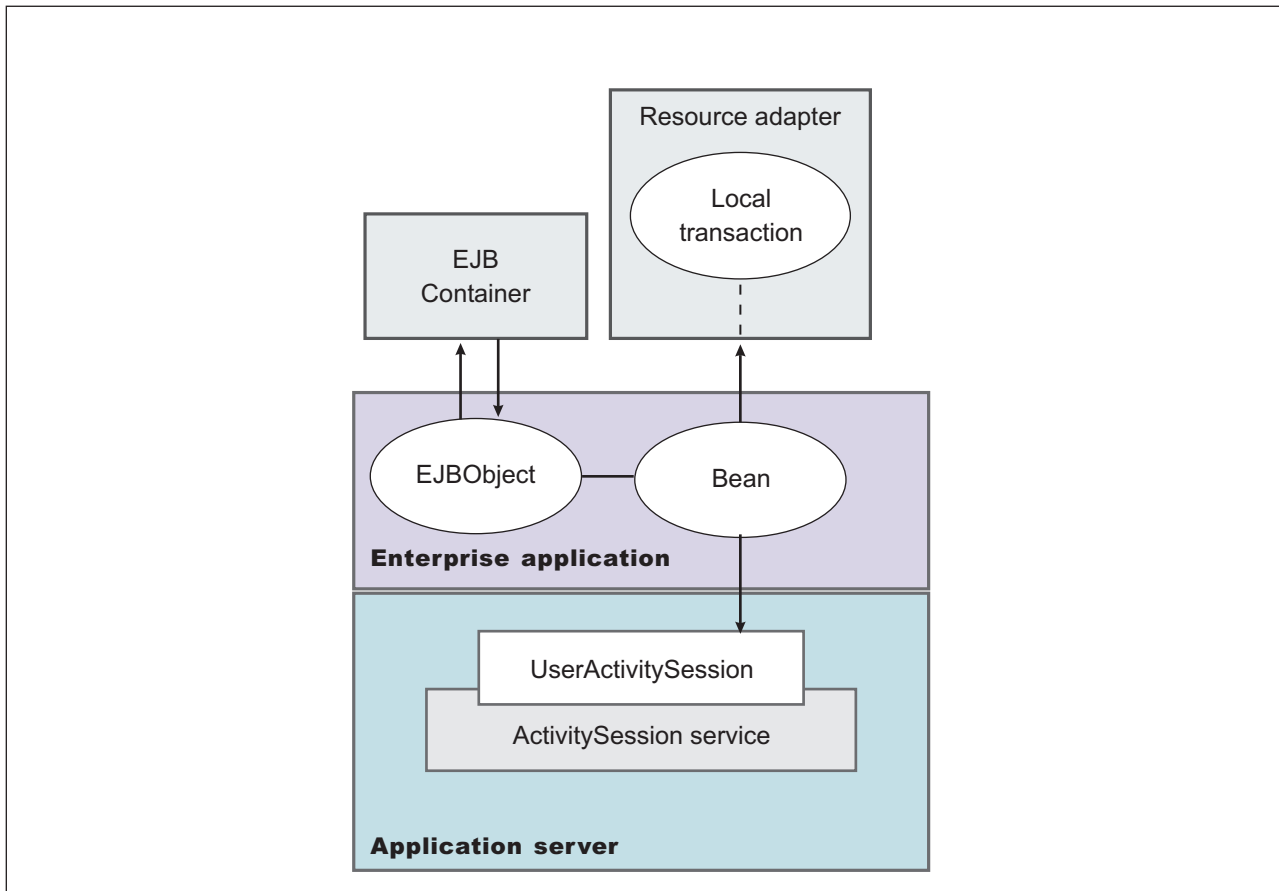


Figure 14. The ActivitySession service. This figure show the main components of the ActivitySession service within WebSphere Application server. For an overview of these components, see the text that accompanies this figure.

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` exceptions.

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`.

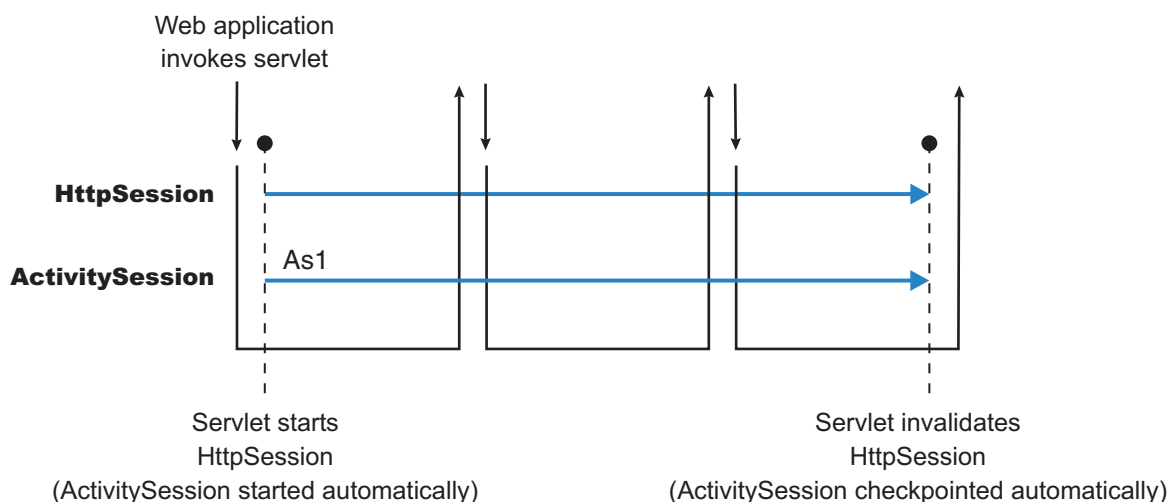
An `ActivitySession` is associated automatically with an `HttpSession`, so can be used to extend access to the `ActivitySession` over multiple HTTP invocations, over inclusion or forwarding of servlets, and to support EJB activation periods that can be determined by the lifecycle of the web HTTP client. 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:



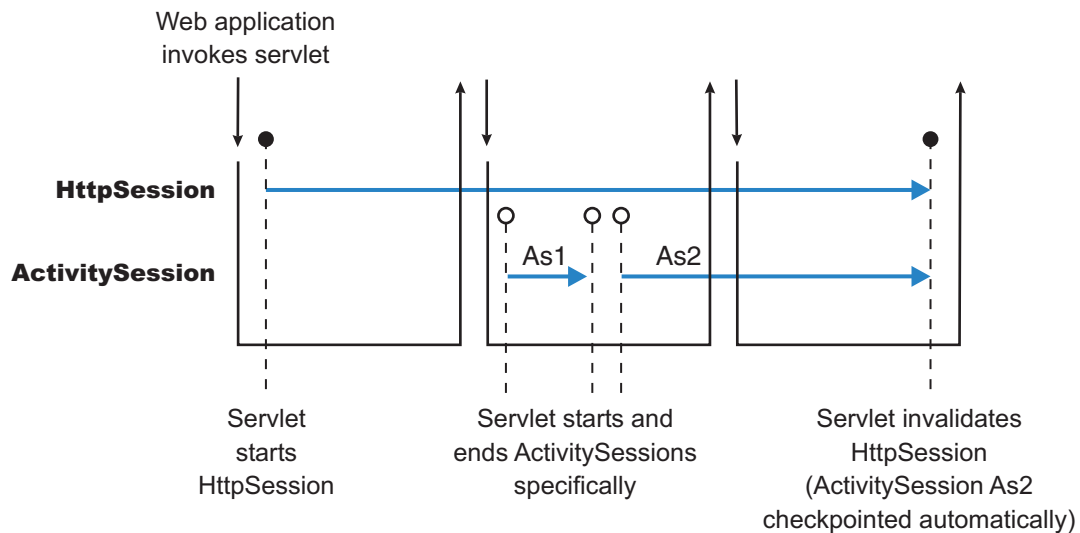
- The Web application starts and ends `ActivitySessions`.

The Web application invokes a servlet that has been configured for application control of ActivitySessions.

- If an HttpSession exists and has an associated ActivitySession, the servlet can use or end that ActivitySession context.
- If an HttpSession does not exist, the servlet can start an HttpSession, but this does not automatically start an ActivitySession.
- If an HttpSession exists but does not have an associated ActivitySession, the servlet can start a new ActivitySession. This automatically associates the ActivitySession with the HttpSession. The ActivitySession lasts either until the ActivitySession is specifically ended or until the HttpSession is ended.

The servlet cannot start a new ActivitySession until an existing ActivitySession has been ended. The servlet cannot start a new HttpSession until an existing HttpSession has been ended.

Within an HttpSession, the Web application can invoke other servlets that can use or end an existing ActivitySession context or, if no ActivitySession exists start a new ActivitySession. When the Web application invokes a servlet that ends the HttpSession, the ActivitySession is ended automatically. This is shown in the following diagram:



A Web application can invoke servlets configured for either usage model.

The following points apply to both usage models:

- To end an HttpSession (and any associated ActivitySession), the Web application must invalidate that session. This causes the ActivitySession to be checkpointed.
- Any downstream 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 reset³. 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` context. This relationship, defined by the `ActivitySession` service, requires that any transaction context be either wholly inside or wholly outside an `ActivitySession` context.

An `ActivitySession` context is very similar to a transaction context and extends the lifecycle choices for activation of enterprise beans; it can encapsulate one or more transactions. The `ActivitySession` context is a distributed context that, like the transaction context, can be bean- or container-managed. An `ActivitySession` context is used mainly by a client to scope the lifecycle of an enterprise bean that it uses either beyond or in the absence of individual transactions started by that client.

`ActivitySessions` have a lower overhead than transactions and can be used instead of transactions that are only used to scope the lifecycle of a called enterprise bean. For a bean with an activation policy of `ActivitySession`, the duration of any resource manager local transactions (RMLTs) started by that bean can be bounded by the duration of the `ActivitySession` instead of the bean method in which the RMLT was started. This provides flexibility and potential for using RMLTs in an enterprise bean beyond the scenarios described in the 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

3. 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 being called. The `ActivitySession` resources remain associated with the `ActivitySession` although they cannot participate further in the `ActivitySession`

transaction context results in a `com.ibm.websphere.ActivitySession.NotSupportedException` on `UserActivitySession.beginSession()`. Any attempt to call `endSession(EndModeCheckpoint)` on an `ActivitySession` that contains an incomplete global transaction results in a `com.ibm.websphere.ActivitySession.ContextPendingException`. Neither the global transaction nor the `ActivitySession` context are affected. If `endSession(EndModeReset)` is called then the `ActivitySession` is reset and the global transactions marked `rollback_only`.

- Each global transaction wholly encapsulated by an `ActivitySession` is independent of every other global transaction within that `ActivitySession`. A rollback of one global transaction does not affect any others or the `ActivitySession` itself.
- `ActivitySession` and global transaction contexts can coexist with an `ActivitySession` encapsulating one or more serially-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 4. Container behavior for activitysession and transaction policies deployment settings

Bean <code>ActivitySession</code> policy(<code>ActivitySession kind</code>)	Bean transaction policy(<code>Container transaction type</code>)	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		

Table 4. 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
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		
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	

Table 4. 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
S1	Suspend S1, Start S2		
T1	Suspend T1, Start S1		
S1, T1	Suspend S1 + T1, Start S2		
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		
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	

Table 4. 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
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		
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	

Table 4. 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
S1	No Action		
T1	Exception		
S1,T1	Exception		
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 control` 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 Control 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 separate 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.

View links to additional information about:

- “Programming model and decisions”
- “Programming specifications” on page 845
- “Other” on page 845

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 Application Server Enterprise Version 5 Overview: Advanced Transactional Connectivity
- Listing of PDF files to learn about WebSphere Application Server Version 5
- Listing of all IBM WebSphere Application Server Redbooks
- Listing of all IBM WebSphere Application Server Whitepapers
- WebSphere Application Server Enterprise Edition 4.0: A Programmer's Guide

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 `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.
2. Create or edit the application EAR file.

Note: Ensure that you set the target server as WebSphere Application Server version 6.

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 v6.0
- 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**.
- f.

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 Setting transactional attributes in the deployment descriptor. For more detail about the relationship between these two properties, see Combining transaction and `ActivitySession` container policies.

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.
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 Deploying and managing applications.

Setting Web module `ActivitySession` deployment attributes

Use this task to set the `ActivitySession` deployment attributes for a Web application to start `UserActivitySessions` and perform work scoped within `ActivitySessions`.

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.
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.
 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

Learn about application profiling

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

- | | |
|--|--|
| Develop and assemble applications for application profiling | • Assemble applications for application profiling |
| Deploy, administer, and troubleshoot your applications | • Deploy applications (same as any application type) |
| | • Administer application profiles |
| | • Automatically configure application profiles and tasks |
| | • Develop TaskNameManagers |

Conceptual overviews

Documentation "Application profiling: Overview" on page 852

Presentations Education on Demand offers:

- WebSphere programming model extensions overview
- Application profiles and dynamic access intent

See Chapter 8 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

No tutorials are available at this time.

Samples

The Samples Gallery offers:

- **Account management**

The account management application is provided to demonstrate a simple example of how to use the application profiling service to configure J2EE applications. The account management application uses a servlet, a stateless session bean, and a container-managed entity bean (EJB 2.1 CMP) to drive a very basic business scenario. One of the paths run by the application is a read-only operation; another path is an update operation. Application profiles, access intent policies, tasks, and task policies are configured such that the persistence manager and container can accurately determine the intentions different operations of the application and optimize the behavior of the persistence operations without compromising data integrity.

Task overview: Application profiling

Application profiling enables you 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 run time environment. For more information, see [Application Profiling: Overview](#).

1. Assembling applications for application profiles. This topic describes how to configure tasks, create application profiles, and configure tasks on profiles.
2. Manage application profiles. This topic describes how to add and remove tasks from application profiles using the administrative console.
3. Use 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: Overview: Application profiling enables you to identify particular units of work to the WebSphere Application Server run time environment. The run time can tailor its support to the exact requirements of that unit of work. Access intent is currently the only run time 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 6.0 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.

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 run time environment uses either a method-level access intent (if any) or a bean-level default access intent.

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 6.0 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.

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 runtime, 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 run-time 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 runtime 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 IOP 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 run time 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 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.

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 run time 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.

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 are initiated after the task name is set. Application managed tasks are configured on all J2EE components.

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.

Application profiling interoperability: **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 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. For information about the 5.x Compatibility Mode, see "Application profiling service settings" in the information center.

Similarly, application profiles that you create using WebSphere Application Server Version 6 are not compatible with Version 5 or earlier versions. Even applications configured with application profiles run on Version 6 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 6.0 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.

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.

If a clustered environment contains both Version 5.x and 6.0 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 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

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, as described in the topic, [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` will cause the task to be associated with the new unit of work. This interface cannot be used within Enterprise Java Beans 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" in the information center.

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.

Automatic configuration of application profiling: 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.

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.

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 Java Server 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 fresh. 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:

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:

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:

Configure an application client's container-managed task to associate requests from the client with an application profile. This task should be performed only for application clients that programmatically begin either a transaction or `ActivitySession`. If a unit of work is not begun, then the configured task is ignored.

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 run time environment.

8. Select **OK**.

Configuring container managed tasks for Web components:

Configure a web component's container managed task to associate requests from a servlet or JavaServer Pages (JSP) file with an application profile. This task should be performed only for web components that programmatically begin either a transaction or ActivitySession. If a unit of work is not begun, then the configured task is ignored.

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 run time environment.

10. Select **OK**.

Configuring container managed tasks for Enterprise Java Beans:

Configure an Enterprise Java Bean's (EJB) container managed tasks to associate requests from the bean with application profiles. This task should be performed only for methods that cause a new transaction or ActivitySession to be begun either by the container or programmatically by the bean developer. 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.

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 run time environment.

13. Select **OK**.

Configuring application managed tasks for application clients:

Configure application managed tasks for an application client to associate requests from the client with application profiles. This task should be performed only for application clients that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession. If a unit of work is not begun after the task is set, then the set task is ignored.

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:

Configure web components' application managed tasks to associate requests from a servlet or JavaServer Pages (JSP) file with application profiles. . This task should be performed only for web components that programmatically set the configured task and then programmatically begin either a transaction or `ActivitySession`. If a unit of work is not begun after the task is set, then the set task is ignored.

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 run time environment.

12. Select **OK**.

Configuring application managed tasks for Enterprise JavaBeans:

Configure an Enterprise JavaBeans (EJB) application managed tasks to associate requests from the bean with application profiles. This task should be performed only for enterprise Java beans that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession. If a unit of work is not begun after the task is set, then the set task is ignored. The task should never be performed for an Enterprise Bean which uses container-managed transactions or container-managed ActivitySessions.

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 run time 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 run time environment.
11. Select **OK**.

Asynchronous beans

Learn about asynchronous beans

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Configure asynchronous beans interfaces and assemble applications that use them

- Configure timer managers
- Configure work managers
- Assemble applications that use asynchronous beans work managers and timer managers
- Assemble applications that use asynchronous beans work managers
- Assemble applications that use CommonJ work managers
- Assemble applications that use CommonJ timer managers
- Develop work objects to run code in parallel
- Develop event listeners
- Develop asynchronous scopes
- Interoperate with asynchronous beans

Develop asynchronous beans objects

Conceptual overviews

Documentation

"Asynchronous beans" on page 870

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Asynchronous beans

See Chapter 7 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

- **Greenhouse by WebSphere**

Using the Greenhouse by WebSphere online supplier, customers can open accounts, select items and amounts to order, and check their order status. The Greenhouse by WebSphere application uses Web services, the Java message service (JMS) API, scheduler, asynchronous beans, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, message-driven beans (MDB), Java server pages (JSP) files, and the struts framework.

- **Asynchronous beans - WebSphere Trader**

This Sample illustrates how to implement a streaming stock ticker server and client using asynchronous beans and J2EE services such as:

- Servlets
- Java Message Service (JMS)
- Session enterprise beans
- Container-managed persistence (CMP) 2.0 enterprise beans
- Message-driven beans (MDB)

This Sample uses several parts to maximize the utilization of a server:

- Work - Runs J2EE context-aware code on a thread.
- Alarm - Runs J2EE context-aware code at a given time interval.
- EventSource - A method of broadcasting events to registered listeners.
- SubsystemMonitor - A thread that monitors the status of any asynchronous system and uses an EventSource method to inform registered listeners of the system status.
- WorkManager - Thread configuration and J2EE context policies that are used by various asynchronous beans parts.
- AsynchScope - A collection of alarms, subsystem monitors and other asynchronous scopes that support relationships. This collection utilizes a single WorkManager thread and is also an event source.
- Startup Bean - A specialized, stateful session enterprise bean that supports bootstrapping asynchronous work when the application starts.

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 be executed 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. (Support for application profiling context is deprecated.)
- Work areas

Asynchronous bean interfaces

Three 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 Javadoc.

Timer listener

This interface is an object that implements the `commonj.timers.TimerListener` interface. Timer listeners are called when a high-speed transaction timer expires. For more information on the `TimerListener` interface, see the Javadoc.

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 Javadoc.

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 Javadoc.

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 Javadoc.

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 run-time environment 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 execute some work within this context. The `WorkWithExecutionContext` object can be executed 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 executed 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"](#) in the information center.

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 topic.

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 topic 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 JVM only if an application within that Java virtual machine (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.

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. Any work manager that implements the `java.io.Serializable` does not have the remote start capability.

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)

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 depends 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

	<p>schedule(Work, WorkListener) throws WorkException, IllegalArgumentException</p> <p>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.</p>		<p>startWork(Work, timeout_ms, WorkListener) throws WorkException, IllegalArgumentException</p>
	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 XD 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)
WorkCompleted-Exception	(Extends WorkException)	WorkCompleted-Exception	(Extends WorkException)
WorkRejected-Exception	(Extends WorkException)	WorkRejected-Exception	(Extends WorkException)

For more information on work manager APIs, refer to the Javadoc.

Work manager examples

Table 5. Look up work manager

Asynchronous beans	CommonJ
<pre>InitialContext ctx = new InitialContext(); com.ibm.websphere.asynchbeans.WorkManager wm = (com.ibm.websphere.asynchbeans.WorkManager) ctx.lookup("java:comp/env/wm/MyWorkMgr");</pre>	<pre>InitialContext ctx = new InitialContext(); commonj.work.WorkManager wm = (commonj.work.WorkManager) ctx.lookup("java:comp/env/wm/MyWorkMgr");</pre>

Table 6. Create your work using MyWork

Asynchronous beans	CommonJ
<pre>public class MyWork implements com.ibm.websphere.asynchbeans.Work { public void release() { } public void run() { System.out.println("Running....."); } }</pre>	<pre>public class MyWork implements commonj.work.Work{ public boolean isDaemon() { return false; } public void release () { } public void run () { System.out.println("Running....."); } }</pre>

Table 7. Submit the work

Asynchronous beans	CommonJ
--------------------	---------

Table 7. Submit the work (continued)

<pre> 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 coll = new ArrayList(); Coll.add(item1); Coll.add(item2); wm.join(coll, 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(coll, WorkManager.JOIN_OR, 1000); </pre>	<pre> 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 coll = new ArrayList(); coll.add(item1); coll.add(item2); wm.waitForAll(coll, 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(coll,1000); // 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()); } } } } </pre>
--	---

Table 8. Create a timer manager

Asynchronous beans	CommonJ
<pre> 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(); </pre>	<pre> InitialContext ctx = new InitialContext(); Commonj.timers.TimerManager tm = (commonj.timers.TimerManager) ctx.lookup("java:comp/env/tm/MyTimerManager"); </pre>

Table 9. Fire the timer

Asynchronous beans	CommonJ

Table 9. Fire the timer (continued)

<pre>// create alarm ABAlarmListener listener = new ABAlarmListener(); Alarm am = mgr.create(listener, "SomeContext", 1000*60);</pre>	<pre>// 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);</pre>
---	--

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 10. 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: Asynchronous bean connection management:

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...
    }
}

```

Developing work objects to run code in parallel

Your administrator must have configured at least one work manager using the administrative console.

To run code in parallel, or in a different J2EE context, 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
```

- Determine the number of work managers needed by this application component.
- 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.

- 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.

- [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.

- Use the `release()` method to set a variable in a synchronized block. For 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:

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 (JMSEException ex)
        {
            // handle the exception here
            throw ex;
        }
        finally

```

```

    {
    if (tc != null)
    {
    try
    {
    tc.close();
    }
    catch (JMSEException 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. The JNDI name for this event source 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:

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. 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 AsyncScope object owns and controls the life cycle of the following objects:

Child asynchronous scopes

Each AsyncScope object extends the AsyncScopeManager interface, which is a factory for AsyncScope objects. (For more information on the AsyncScopeManager interface, see the Javadoc 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 AsyncScope 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 AsyncScope 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" in the information center 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 AsyncScope object is destroyed.

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 `SubsystemMonitorEvents` 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. 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.

Assembling applications that use work managers and timer managers

Configure at least one work manager or timer manager using the administrative console.

The work manager and timer manager objects are both supported for assembling applications that implement the asynchronous bean technology. 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:

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:

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:

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.

Dynamic cache

Learn about dynamic caching

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Configure and administer caching

- Use the dynamic cache service to improve performance
- Enable the dynamic cache service
- Configure servlet caching
- Configure Edge side include (ESI) caching
- Control caches outside of the application server
- Configure disk offload
- Configure Web services caching
- Configure the Web services client cache
- Use programming interfaces to cache and share objects

Monitor dynamic cache activity

Develop cache policies

- Display cache information
- Develop a cachespec.xml file to configure cacheable objects
- Understand cachespec.xml file contents (reference)

Troubleshoot the dynamic cache

- Troubleshoot the dynamic cache service
- Dynamic cache troubleshooting tips

Conceptual overviews

Documentation

“Dynamic cache” on page 892

Tutorials

Tutorials are not available at this time.

Samples

Samples are not available at this time.

Task overview: Using the dynamic cache service to improve performance

Use the dynamic cache to improve application performance by caching the output of servlets, commands, and JavaServer Pages (JSP) files.

On distributed platforms, WebSphere Application Server, Version 4.0, supported the configuration of dynamic servlet caching through the `servletcache.xml` file. To utilize the new and improved functionality of the dynamic cache service, configure your cache policy using the `cachespec.xml` format. See “Configuring cacheable objects with the cachespec.xml file” for more information.

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” in the information center.
2. Configure the type of caching that you are using.
 - Configure servlet caching.
 - Configure Edge Side Include caching.
 - Configure command caching.
 - “Example: Caching Web services” on page 892.

- Configure the Web services client cache.
3. You can monitor the results of your configuration using the dynamic cache monitor. For more information, see "Displaying cache information" in the information center.
 4. If you have any problems with your configuration, see "Troubleshooting the dynamic cache service" in the information center.

To use the DistributedMap and DistributedObjectCache interfaces for the dynamic cache, see "Using the DistributedMap and DistributedObjectCache interfaces for the dynamic cache" on page 896.

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 servlet after it runs, and metadata.

Example: Caching Web services: The following is an 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 Simple Object Access Protocol (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.

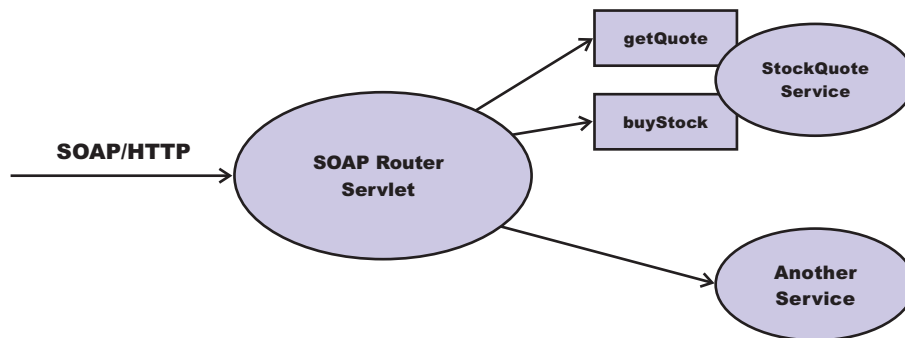
Message example 2

```

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 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>
</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>
</cache-id>
</cache-entry>
</cache>

```

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>
```

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" in the information center for more information.

The DistributedMap and DistributedObjectCache interfaces are a 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 899. There are three methods for configuring and using cache instances.

Note: For more information about the DistributedMap and DistributedObjectCache interfaces, see the API documentation for the com.ibm.websphere.cache package. See Javadoc for more information.

- **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=${WAS_INSTALL_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.xml 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.xml

```
<?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.

Note: DistributedMap and DistributedObjectCache do not have authorization or access control associated with the cache entries.

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 - no set maximum value

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.

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
---------	-------

Disk offload location:

Specifies the directory that is used for disk offload.

If disk offload location is not specified, the default location, `$install_root/temp/node/servername/_dynacache/cacheJNDIname` is used. If disk offload location is specified, the node, server name, and cache instance name are appended. For example, `$install_root/diskoffload` generates the location as `$install_root/diskoffload/node/servername/cacheJNDIname`. This value is ignored if `enableDiskOffload` is false.

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
---------	-------

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
---------	------

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**.

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 not be available for cache instances that were created with a previous version of WebSphere Application Server.

Replication settings:

Click **Replication settings** to go to the replication settings panel to configure data replication for this cache instance.

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 Javadoc for the `com.ibm.websphere.cache` package.

Dynamic query

Learn about dynamic query

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Develop and assemble applications that use EJB or dynamic queries

Use the EJB query language

See examples of EJB queries

Use the dynamic query service

Run a dynamic EJB query using the remote interface

Run a dynamic EJB query using the local interface

Use dynamic query `prepareQuery()` and `executePlan()` methods

Assemble applications for deployment (same as any application type)

Deploy and administer your applications

Deploy and administer applications (same as any application type)

Deploy applications (Education on Demand)

Administer applications (Education on Demand)

Troubleshoot queries

Improve dynamic query performance

Conceptual overviews

Documentation

"Using the dynamic query service" on page 926

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Dynamic query language

See Chapter 11 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

- **Dynamic query - employee finder**
An application that locates department and employee records by running dynamic queries that you supply or choose from among a variety of prepared queries.

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](#).

For your WebSphere application, you can define an EJB query in 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.

Before using EJB query, familiarize yourself with query language concepts, starting with the topic, EJB Query Language.

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.

The SELECT clause is optional in order to maintain compatibility with WebSphere Application Server Version 4.

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 11. 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)	<ul style="list-style-type: none"> • deptno - Integer (key) • name - String • budget - BigDecimal
Relationships	<ul style="list-style-type: none"> • emps - 1:Many with EmpEJB • mgr - Many:1 with EmpEJB

Table 12. EmpBean schema

Entity bean name (EJB name)	EmpEJB (not used in query)
Abstract schema name	EmpBean
Implementation class	com.acme.hr.empBean (not used in query)
Persistent attributes (cmp fields)	<ul style="list-style-type: none"> • empid - Integer (key) • name - String • salary - BigDecimal • bonus - BigDecimal • hireDate - java.sql.Date • birthDate - java.util.Calendar • address - com.acme.hr.Address
Relationships	<ul style="list-style-type: none"> • 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 923. 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 millisecond 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 identified either by an abstract schema name and an identification variable, called a range variable, or by a collection member declaration that identifies a multivalued relationship and an identification variable.

Conceptually, the semantics of the query is to first form a temporary collection of tuples R. Tuples are composed of elements from the collections identified in the FROM clause. Each tuple contains one element from each of the collections in the FROM clause. All possible combinations are formed subject to the constraints imposed by the collection member declarations. If any schema name identifies a collection for which there are no records in the persistent store, then the temporary collection R will be empty.

Example: FROM clause

DeptBean contains records 10, 20 and 30 in the persistent store. EmpBean contains records 1, 2 and 3 that are related to department 10 and records 4, 5 that are related to department 20. Department 30 has no related employees.

```
FROM DeptBean d, EmpBean e
```

This forms a temporary collection R that contains 15 tuples.

```
FROM DeptBean d, DeptBean d1
```

This forms a temporary collection R that contains 9 tuples.

```
FROM DeptBean d, IN (d.emps) AS e
```

This forms a temporary collection R that contains 5 tuples. Department 30 because it contains no employees will not be in R. Department 10 will be contained in R three times and department 20 will be contained in R twice.

After forming the temporary collection the search conditions of the WHERE clause will be applied to R and this will yield a new temporary collection R1. The ORDER BY and SELECT clauses are applied to R1 to yield the final result set.

An identification variable is a variable declared in the FROM clause using the operator IN or the optional AS.

```
FROM DeptBean AS d, IN (d.emps) AS e
```

is equivalent to:

```
FROM DeptBean d, IN (d.emps) e
```

An identification variable that is declared to be an abstract schema name is called a range variable. In the query above "d" is a range variable. An identification variable that is declared to be a multivalued path expression is called a collection member declaration. "d" and "e" in the example above are collection member declarations.

Note that the following path expression is illegal as a collection member declaration because it is not multivalued:

```
e.dept.mgr
```

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 data store.

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 data store.

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.

A Calendar type can be compared to another Calendar type, an exact numeric literal or input parameter of type long whose value is the standard Java long millisecond value.

The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate < 631180800232
```

Date expressions can be used in comparison expressions and can be composed of operators + - , date duration expressions and date functions, path expressions that evaluate to a date value, string representation of a date and date input parameters.

Time expressions can be used in comparison expressions and can be composed of operators + - , time duration expressions and time functions, path expressions that evaluate to a time value, string representation of time and time input parameters.

Timestamp expressions can be used in comparison expressions and can be composed of operators + - , timestamp duration expressions and timestamp functions, path expressions that evaluate to a timestamp value, string representation of a timestamp and timestamp input parameters.

Standard bracketing () for ordering expression evaluation is supported.

The operators and their precedence order from highest to lowest are:

- Navigation operator (.)
- Arithmetic operators in precedence order:
 - + - unary
 - * / multiply, divide
 - + - add, subtract
- Comparison operators: =, >, <, >=, <=, <>(not equal)
- Logical operator NOT
- Logical operator AND
- Logical operator OR

Null value semantics: The following describe the semantics of NULL values:

- Comparison or arithmetic operations with an unknown (NULL) value yield an unknown value
- In a Java 2 platform, Enterprise Edition (J2EE) version 1.3 application, a path expression uses an outer-join semantic where a NULL field or cmr value evaluates to NULL. In J2EE version 1.4, the path expression uses an inner-join semantic.
- The IS NULL and IS NOT NULL operators can be applied to path expressions and return TRUE or FALSE. Boolean operators AND, OR and NOT use three valued logic.

AND	True	False	Unknown
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

OR	True	False	Unknown
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

	NOT
True	False
False	True
Unknown	Unknown

Example: Null value semantics

```
select object(e) from EmpBean where e.salary > 10 and e.dept.budget > 100
```

If salary is NULL the evaluation of `e.salary > 10` returns unknown and the employee object is not returned. If the cmr field dept or budget is NULL evaluation of `e.dept.budget > 100` returns unknown and the employee object is not returned.

```
select object(e) from EmpBean where e.dept.budget is null
```

In J2EE 1.3 if dept or budget is NULL evaluation of `e.dept.budget is null` returns TRUE and the employee object is returned. In J2EE 1.4 the employee object is returned only if budget is NULL.

```
select object(e) from EmpBean e , in (e.dept.emps) e1 where e1.salary > 10
```

If dept is NULL, then the multivalued path expression `e.dept.emps` results in an empty collection (not a collection that contains a NULL value). An employee with a null dept value will not be returned.

```
select object(e) from EmpBean e where e.dept.emps is empty
```

If dept is NULL the evaluation of the predicate in unknown and the employee object is not returned.

```
select object(e) from EmpBean e , EmpBean e1 where e member of e1.dept.emps
```

If dept is NULL evaluation of the member of predicate returns unknown and the employee is not returned.

Date time arithmetic and comparisons: DATE, TIME and TIMESTAMP values 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

Format	Date format	Date examples	Time format	Time examples
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
```

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.

```
HOURL ( Time ) -> Integer
HOURL ( < time-string > ) -> Integer
HOURL ( < time-duration > ) -> Integer
HOURL ( Timestamp ) -> Integer
HOURL ( < timestamp-string > ) -> Integer
HOURL ( < 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 ObjectNotFoundException.

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.

A scalar-subselect is a subselect that returns a single value.

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
```

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
```

The following is a valid dynamic query:

```
SELECT OBJECT(e), e.salary+e.bonus as total_pay FROM EmpBean e ORDER BY 2 DESC
```

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 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.

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 ::=select_clause_dynamic from_clause [where_clause]
    [group_by_clause] [having_clause] [order_by_clause]
from_clause::=FROM identification_variable_declaration
    [,identification_variable_declaration]*
identification_variable_declaration::=collection_member_declaration |
    range_variable_declaration
```

```

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) |
    AVG([ALL|DISTINCT] expression) |
    BIGINT(expression) |
    CHAR({expression [, {ISO|USA|EUR|JIS}] } ) |
    CONCAT (expression , expression ) |
    COUNT({[ALL|DISTINCT] 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] ) |
    MAX([ALL|DISTINCT] expression) |
    MICROSECOND( expression ) |
    MIN([ALL|DISTINCT] expression) |
    MINUTE ( expression ) |
    MOD (expression, expression) |
    MONTH( expression ) |
    REAL( expression ) |
    SECOND( expression ) |
    SMALLINT( expression ) |
    SQRT ( expression ) |
    SUBSTRING( expression, expression[, expression]) |
    SUM([ALL|DISTINCT] expression) |
    TIME( expression ) |
    TIMESTAMP( expression ) |
    UCASE ( expression ) |
    YEAR( expression )

```

Comparison of EJB 2.1 specification and WebSphere query language: WebSphere Application Server Version 6.0 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

Item	
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 Javadoc (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 `<WAS_HOME>/installableApps` 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 `ejbbean.ear`
 - b. map the `ejbbean.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 Javadoc 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 Javadoc 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 Javadoc 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 113 and "Concurrency control" on page 114.

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 435 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 runtime, rather than defining them at deployment time. By using dynamic query you gain the flexibility of queries defined at runtime 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

Learn about internationalization

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Develop and assemble internationalized applications

- Internationalize applications
- Internationalize interface strings (localizable-text API)
- Internationalize application components (internationalization service)
- Migrate internationalized applications
- Use the internationalization context API
- Assemble internationalized applications

Develop and deploy internationalized applications

- Deploy and administer applications (same as any application type)
- Deploy applications (Education on Demand)
- Administer applications (Education on Demand)
- Administer the internationalization service
- Enable the internationalization service for servlets and enterprise beans
- Enable the internationalization service for EJB clients

Troubleshoot internationalized applications

Refer to the *Troubleshooting and support* PDF.

Conceptual overviews

Documentation

"Internationalization" on page 934

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Internationalization

See Chapter 16 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

- **Internationalization service currency exchange sample**

The Currency exchange application exchanges U.S. dollars to foreign currencies, or vice versa, and illustrates the use of the Internationalization service in performing server-side localizations that are sensitive to client-side locale information. The Internationalization service manages the propagation of locale and time zone information, collectively referred to as Internationalization context, between client and server application components. Within server components, Currency exchange uses internationalization context with the Java 2 Platform, Standard Edition (J2SE) Internationalization API to perform localizations, such as resource isolation, collation, and message formatting.

Task overview: Internationalizing applications

An application that can present information to users according to regional cultural conventions is said to be *internationalized*. The application can be configured to interact with users from different localities in culturally appropriate ways. In an internationalized 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.

This product supports internationalization through use of its localizable-text API and internationalization service.

- 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 936.

- 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 937.

Internationalization:

An application that can present information to users according to regional cultural conventions is said to be *internationalized*: The application can be configured to interact with users from different localities in culturally appropriate ways. In an internationalized 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.

Historically, the creation of internationalized 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 internationalize a much wider variety of applications. This trend requires making internationalization 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 internationalized, 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.

Internationalization 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 937.

Internationalization: Resources for learning: Use the following links to find relevant supplemental information about internationalization. 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.
- International Components for Unicode for Java
The portal site for IBM’s open-source API to extend basic Unicode support in Java application components.

Programming specifications

- Java 2 SDK, Standard Edition Documentation: Internationalization
The Java internationalization documentation from Sun Microsystems, including a list of supported locales and encodings.
- 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, Web Services Internationalization Task Force
The task force of the W3C’s Internationalization 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 product supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

This topic summarizes the steps involved in implementing message catalogs through the localizable-text API.

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.

Task overview: Internationalizing application components (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.

This topic summarizes the steps involved in using the internationalization service.

1. If you have an application that uses the WebSphere Application Server Version 4.0 internationalization service, migrate your application as needed.
2. 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.

3. 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.

4. 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.

5. 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 internationalization 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. For an understanding of how these differences impact application development, read “Internationalization” on page 934.

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.

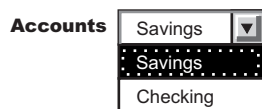
Every application component has a default policy, which can be overridden and tailored for servlets and enterprise beans at assembly time.

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.

Identifying localizable text

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

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

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 964.

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 "Resources for learning."

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 resourceName, String patternKey, String appName)`
- `LocalizableTextFormatter(String resourceName, 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 resourceName)`
- `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 942 at the end of this topic. For a usage example, see “Composing complex strings.”

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."

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:

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."
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:

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:

Identify strings that need to be localized.

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.

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:

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(1tf.format(this.applicationLocale), Label.CENTER);
}
catch (LocalizableException le) {
    msg = new Label(1tf.getFallbackString(), Label.CENTER);
}
...
}

```

When the application is finished, deploy your application. For more information, see “Preparing the localizable-text package for deployment” on page 964.

Customizing the behavior of a formatting method:

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.

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. Classes specific to the internationalization service reside in the `install_root/lib/i18nctx.jar` file. Before compiling application components that import internationalization service classes, add the `i18nctx.jar` file to your CLASSPATH setting.

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 960.

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:

This topic describes how 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:

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.

Perform the following task to access elements of the Caller internationalization context.

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:

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.

Perform the following task to access elements of the invocation internationalization context.

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: The following code example illustrates how to use the internationalization context API within a contained Enterprise JavaBeans (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: The following code example illustrates how to use the internationalization context API within a servlet. Note comments in the init and doPost methods.

```

...
//-----
// INTERNATIONALIZATION SERVICE: Imports.
//-----
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;

```

```

import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;

public class J2eeServlet extends HttpServlet {

    ...
    //-----
    // INTERNATIONALIZATION SERVICE: API references.
    //-----
    protected UserInternationalization userI18n = null;
    protected Internationalization i18n = null;
    protected InvocationInternationalization invI18n = null;

    //-----
    // INTERNATIONALIZATION SERVICE: JNDI name.
    //-----
    public static final String UserI18NUrl =
        "java:comp/websphere/UserInternationalization";

    protected Locale callerLocale = null;
    protected Locale invocationLocale = null;

    /**
     * Initialize this servlet.
     * Resolve references to the JNDI initial context and the
     * internationalization context API.
     */
    public void init() throws ServletException {

        //-----
        // INTERNATIONALIZATION SERVICE: Resolve API.
        //
        // Under Container-managed Internationalization (CMI), servlets have
        // read-only access to invocation context elements. Attempts to set these
        // elements result in an IllegalStateException.
        //
        // Suggestion: cache all internationalization context API references
        // once, during initialization, and use them throughout the servlet
        // lifecycle.
        //-----
        try {
            Context initialContext = new InitialContext();
            userI18n = (UserInternationalization)initialContext.lookup(UserI18NUrl);
            callerI18n = userI18n.getCallerInternationalization();
            invI18n = userI18n.getInvocationInternationalization();
        } catch (NamingException ne) {
            throw new ServletException("Cannot resolve UserInternationalization" + ne);
        } catch (IllegalStateException ise) {
            throw new ServletException ("Error: UserInternationalization is not
                available: " + ise);
        }
        ...
    } // init

    /**
     * Process incoming HTTP GET requests.
     * @param request Object that encapsulates the request to the servlet
     * @param response Object that encapsulates the response from the
     * Servlet.
     */
    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    } // doGet

```

```

/**
 * Process incoming HTTP POST requests
 * @param request Object that encapsulates the request to
 *   the Servlet.
 * @param response Object that encapsulates the response from
 *   the Servlet.
 */
public void doPost(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    ...
    //-----
    // INTERNATIONALIZATION SERVICE: Get caller context.
    //
    // The internationalization service extracts the accept-languages
    // propagated in the HTTP request and associates them with the
    // current thread as a list of locales within the caller context.
    // These locales are accessible within HTTP Servlet service methods
    // using the caller internationalization object.
    //
    // If the incoming HTTP request does not contain accept languages,
    // the service associates the server's default locale. The service
    // always associates the GMT time zone.
    //
    //-----
    try {
        callerLocale = callerI18n.getLocale(); // caller locale
        // the following code enables you to get invocation locale,
        // which depends on the Internationalization policies.
        invocationLocale = invI18n.getLocale(); // invocation locale
    } catch (IllegalStateException ise) {
        log("An anomaly occurred accessing Invocation context: " + ise);
    }
    // NOTE: Browsers may propagate accept-languages that contain a
    // language code, but lack a country code, like "fr" to indicate
    // "French as spoken in France." The following code supplies a
    // default country code in such cases.
    if (callerLocale.getCountry().equals(""))
        callerLocale = AccInfoJBean.getCompleteLocale(callerLocale);

    // Use iLocale in JDK locale-sensitive operations, etc.
    ...
} // doPost

...
void log(String s) {
    System.out.println ((s == null) ? "null" : s);
}
} // CLASS J2eeServlet

```

Example: Internationalization context in a session bean: The following 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.TimeZone timeZone);
    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 invocation has two internationalization contexts under which it runs. For each application component invocation, the container enters the caller context and the invocation context, as indicated by the pertinent internationalization policy, into scope before the container delegates to the actual implementation. When the implementation returns, the service removes these contexts from scope. The internationalization service supplies no programmatic mechanism for components to explicitly manage the scope of internationalization context.

The service scopes internationalization context differently with respect to application component type:

- “EJB client programs (contained)”
- “Servlets”
- “Enterprise beans” on page 959
- “Web service client programs (contained)” on page 959
- “Stateless session beans that are enabled for Web services” on page 959

Internationalization context observes by-value semantics over remote method requests. Changes to internationalization context elements that are scoped to an invocation 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 zone](#).

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 invokes 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 Simple Object Access Protocol (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."

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 runs 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: The following code example illustrates how internationalization context is represented within the Simple Object Access Protocol (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" in the information center.

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.

Preparing the localizable-text package for deployment

Write code to compose the language-specific strings.

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.

1. Verify that the `LocalizableTextEJBDeploy` tool (`ltext.jar`) exists in the `lib` directory under the installation root directory for the product.
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. The file that contains this tool (ltext.jar) must be located in the lib directory of the product installation root.

```
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.

Assembling internationalized applications

Use 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 966
- “Setting the internationalization type for enterprise beans” on page 967

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 968

Setting the internationalization type for servlets:

This task sets the internationalization type for a servlet within a Web module.

This article assumes that you have an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also 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.

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 article assumes that you have an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also 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.

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 article assumes that you have an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also 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.

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 article assumes that you have an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also 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.

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.

Object pools

Learn about object pools

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Enable the object pool service
Configure object pool managers

Conceptual overviews

Documentation

"Object pool managers" on page 970

See Chapter 14 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

No tutorials are available at this time.

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. These 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 Pools**.
3. Define the name of the object pool manager. This name can be up to 30 ASCII characters long.
4. Assign the object pool manager a Java Naming and Directory Interface (JNDI) name.

5. Provide a description of this object pool manager.
6. Categorize 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 Messaging 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:

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 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.

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 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 > 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:

The fully qualified class name of the objects that are stored in the object pool.

Data type String

Pool implementation class name:

The fully qualified class name of the CustomObjectPool implementation class for this 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	Selected
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: 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

Scheduler

Learn about scheduler

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Develop and schedule tasks

- Assigning schedulers
- Develop a task that calls a session bean
- Developing a task that sends a JMS message
- Receiving scheduler notifications
- Submit a task to a scheduler

Administer and secure tasks

- Administer tasks with a scheduler
- Secure scheduler tasks

Conceptual overviews

Documentation

“Using schedulers” on page 977

developerWorks articles

WebSphere Enterprise Scheduler planning and administration guide

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Scheduler
- Calendar

See Chapter 13 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

The Samples Gallery offers:

- **Greenhouse by WebSphere**

Using the Greenhouse by WebSphere online supplier, customers can open accounts, select items and amounts to order, and check their order status. The Greenhouse by WebSphere application uses Web services, the Java message service (JMS) API, scheduler, asynchronous beans, container-managed persistence (CMP), container-managed relationships (CMR), stateless session beans, message-driven beans (MDB), Java server pages (JSP) files, and the struts framework.

- **Scheduler - Account Report**

The Account Report application demonstrates use of the Scheduler APIs, in addition to the J2EE 1.4 programming model. This application consists of a servlet, container-managed persistence (CMP) beans, and session beans. Using the servlet, a report generation task can be scheduled. After each specified interval, account balances are totaled, and a new report is generated.

Using schedulers

Schedulers enable J2EE application tasks to run at a requested time. 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

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. 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:

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:

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."

```

Developing and scheduling tasks

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 that will call a method on a TaskHandler session bean

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 Javadoc. 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.

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 sched0 [$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 $sched0 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 (Javadoc) 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](#).

Receiving scheduler notifications:

Various notification events are generated by a scheduler when it performs an operation on a task. These events 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 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 $sched0 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:

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 $sched0 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 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 will deadlock.

All methods defined by the Scheduler API are described in Javadoc.

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:

Transactions and the scheduler daemon

Scheduled `BeanTaskInfo` and `MessageTaskInfo` tasks execute once. This 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`.

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. If the NotificationSink bean is set, a FIRING notification is fired.
6. The BeanTaskInfo or MessageTaskInfo object has started.
7. If the task fails and the NotificationSink bean is set, a FIRE_FAILED notification is fired on a separate transaction.
8. If the task's NotificationSink bean is set, then the various notifications are fired as required.
9. 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's 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 "Requires".
- 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.

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 setting the bean's transaction setting to "Requires new".

Transactions and the scheduler interface

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, the existing global transaction is used. Otherwise, a new global transaction begins.

If the method participates in the caller's global transaction and an unexpected error occurs, 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, the transaction is rolled back, and the exception is resubmitted to the caller.

Scheduler task user authorization: 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 global 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 global 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:

- Global 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 "Deprecated features in Version 6.0" in the information center.

Securing scheduler tasks:

Scheduled tasks are protected using application isolation and administrative roles. If a task is created using a 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 global 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. Configure global security.
2. Manage schedulers. Refer to the *Administering applications and their environment* PDF.

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 handles 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" in the information center), 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.

Updating 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:

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 Javadoc.

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 987, 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 Javadoc.

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 Javadoc .

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 Web 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 Javadoc.

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 Web 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 Javadoc 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 Web Developer:

```
com.ibm.websphere.scheduler.UserCalendarHome  
com.ibm.websphere.scheduler.UserCalendar
```

The bean itself needs to implement the `applyDelta()` and `validate()` methods defined in the remote interface. For details, see the Interface `UserCalendar` in the Javadoc.

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 Javadoc 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

Learn about startup beans

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Configure startup beans

Conceptual overviews

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Startup beans

See Chapter 12 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

Tutorials are not available at this time.

Samples

Samples are not available at this time.

Using startup beans

A startup bean is a session bean that is loaded when an application starts. 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. Use the home interface, `com.ibm.websphere.startupservice.AppStartUpHome`, to designate a bean as a startup bean.
2. Use the remote interface, `com.ibm.websphere.startupservice.AppStartUp`, to define `start()` and `stop()` methods on the bean.

The startup bean `start()` method is called when the application starts and contains business logic to be run at 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()` method is called when the application stops and contains business logic to be run at 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.

View the startup beans service settings.

Startup beans service settings:

Use this page to enable or disable startup beans on all applications within an Application Server. A startup bean is a special session bean with start() and stop() methods containing business logic that is run at application start time and application stop time. 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
Range

Cleared
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

Learn about work areas

This topic provides links to Web resources for learning, including conceptual overviews, tutorials, samples, and "How do I?..." topics, pending their availability.

How do I?...

Implement shared work areas

Conceptual overviews

Documentation

“Work area service - Overview” on page 997

Presentations

Education on Demand offers:

- WebSphere programming model extensions overview
- Work areas

See Chapter 15 of the IBM Redbook WebSphere Application Server Enterprise Version 5 and Programming Model Extensions

Note:

- Version 5.x Redbooks are cited for their conceptual material. Product technical details have changed in Version 6. Refer to the product documentation for current product and technical details. Links to Version 6 Redbooks will be added as they become available.
- Redbooks are supplemental rather than formal product documentation. Read their Notices carefully. For information about supported configurations, consult the product documentation.
- The product documentation is available in either information center format or in PDF book format.

Tutorials

No tutorials are available at this time.

Samples

The Samples Gallery offers:

- **WorkArea Company Context**

The WorkArea Company context Sample verifies the correct installation and configuration of the WorkArea service in the three supported run-time environments: J2EE client, servlet, and enterprise bean. It can also be used to verify the correct installation of the WorkArea partition service in the same three run-time environments.

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. Refer to the information center or to the *Administering applications and their environment* PDF for details.
2. Managing work areas. The work area service is managed using the administrative console. . Refer to the information center or to the *Administering applications and their environment* PDF for details.

Work area service - Overview: 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.

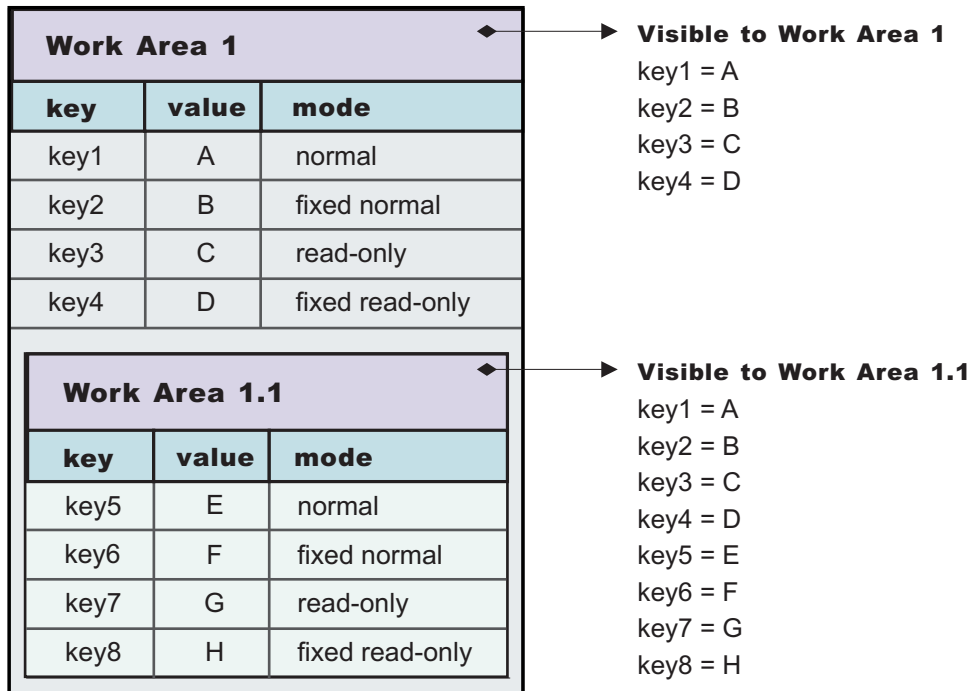


Figure 15. Defining new properties in nested work areas

Nesting can also affect the apparent settings of the properties. Properties can be deleted from or directly modified only within the work areas in which they were set, but nested work areas can also be used to temporarily override information in the property without having to modify the property. Depending on the modes associated with the properties in the enclosing work area, the modes and the values of keys in the enclosing work area can be overridden within the nested work area.

The mode associated with a property when it is created determines whether nested work areas can override the property. From the perspective of a nested work area, the property modes used in enclosing work areas can be grouped as follows:

- Modes that permit a nested work area to override the mode or the value of a key locally. The modes that permit overriding are:
 - Normal
 - Fixed normal
- Modes that do not permit a nested work area to override the mode or the value of a key locally. The modes that do not permit overriding are:
 - Read-only
 - Fixed read-only

If an enclosing work area defines a property with one of the modes that can be overridden, a nested work area can specify a new value for the key or a new mode for the property. The new value or mode becomes the value or mode seen by subsequently nested work areas. Changes to the mode are governed by the restrictions described in Changing modes. If an enclosing work area defines a property with one of the modes that cannot be overridden, no nested work area can specify a new value for the key.

A nested work area can delete properties from enclosing work areas, but the changes persist only for the duration of the nested work area. When the nested work area is completed, any properties that were added in the nested area vanish and any properties that were deleted from the nested area are restored.

The following figure illustrates the overriding of properties from an enclosing work area. The nested work area redefines two of the properties set in the enclosing work area. The other two cannot be overridden. The nested work area also defines two new properties. From the outermost work area, the properties set

or redefined in the nested work are not visible. From the nested work area, the properties in both work areas are visible, but the values seen for the redefined properties are those set in the nested work area.

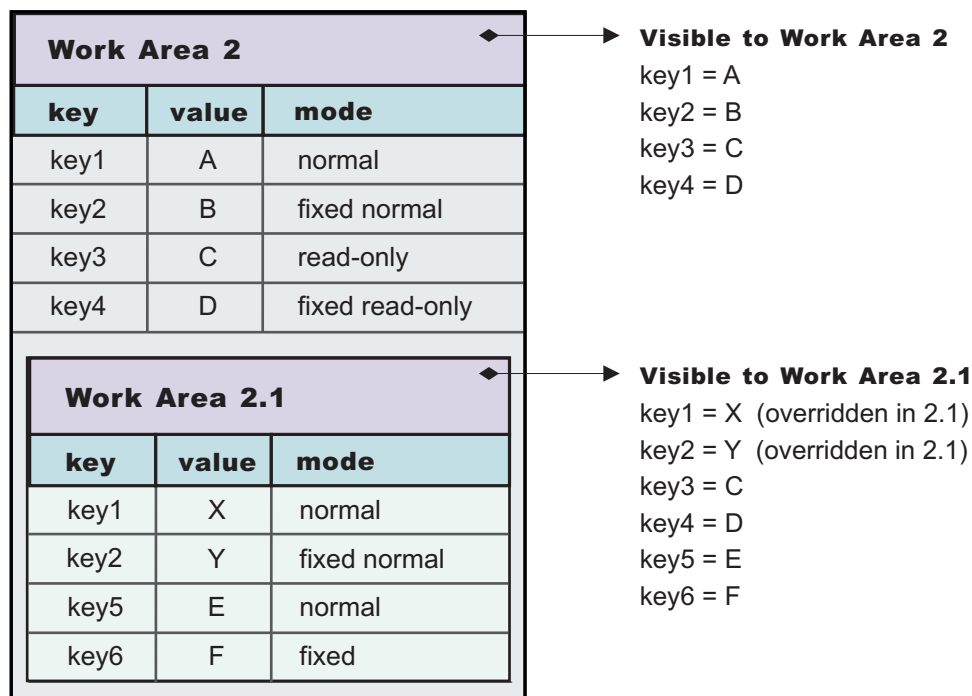


Figure 16. Redefining existing properties in nested work areas

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 WorkArea 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. 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, but 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. 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" in the information center 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.

Chapter 5. Debugging applications

To debug your application, you must use your application development tool (such as Rational Application Developer) to create a Java project or a project with a Java nature. 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.

There are two debugging styles 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 need not 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 then specify a name for the project (such as WASExamples).
 4. Press **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. Select the directory Browse pushbutton and go to the following directory:
installation_root\installedApps\node_name\DefaultApplication.ear\DefaultWebApplication.war.
 7. Select the checkbox next to DefaultWebApplication.war in the left side of the Import dialog and then click **Finish**. This will import 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. To do this, select **Properties** from the right-click menu. Choose the Java Build Path node and then select the Libraries tab. Use the Add External JARs pushbutton to add the following JAR files:
 - *installation_root\installedApps\node_name\DefaultApplication.ear\Increment.jar*.
Once you have added this JAR file, select it and use the **Attach Source** pushbutton to attach Increment.jar as the source - Increment.jar contains both the source and class files.
 - *installation_root\lib\j2ee.jar*
 - *installation_root\lib\pagelist.jar*
 - *installation_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. Once 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://localhost: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 IBM Support page.

Debugging with 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 6. 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 article 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
- 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 Web 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 article 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 articles on **Rapid deployment of J2EE applications** in this information center 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 Web 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 article “Assembling applications: Resources for learning” on page 1025 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 the AST or Rational Web 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.
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 1041 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.

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)

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), Rational Web Developer 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.

Starting an assembly tool

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) deployment descriptors.

Before you can assemble code artifacts into modules, install an assembly tool. Either or both assembly tools are available on CD-ROM in the WebSphere Application Server CD-ROM package. To install an assembly tool, follow the installation instructions for the tool that are on its CD-ROM.

Users on the z/OS platform: The assembly tools are not available on the z/OS platform. Install an assembly tool on an operating system that the tools support, such as a Linux Intel or Windows platform.

If you have installed an assembly tool previously and you install an assembly tool again, you must delete the workspace of the previous installation of the assembly tool before starting it. The default workspace directory is *my_directory/eclipse/workspace*. If you do not delete the workspace for a previous installation, you might encounter error messages such as the following when starting the assembly tool:

Problems during startup. Check the ".log" file in the ".metadata" directory of your workspace.

1. Run the `eclipse` executable. For information on parameters that you can specify when starting the `eclipse` executable, refer to the article "Running Eclipse" in the assembly tool online help. Click **Help > Help Contents > Workbench User Guide > Tasks > Running Eclipse**.
2. In the Workspace Launcher dialog, specify the workspace directory and click **OK** to launch the graphical interface.

The navigation tree displays a hierarchical structure used to build the contents of a new module, or to work with the contents of an existing module.

After you start an assembly tool, configure it. Next, consider whether you have any existing J2EE 1.2 or 1.3 application modules that you would like to migrate to J2EE 1.4.

You can import or create new modules of the following types, to assemble into an application module later:

- Assembling enterprise bean (EJB) modules
- Assembling Web modules
- Assembling application client modules
- Assembling resource adapter modules

Rather than import or create new modules to assemble an application, you can proceed directly to assembling a new application module. While assembling an application module, you can create any new modules that you need.

Configuring an assembly tool

When you first start an assembly tool such as Application Server Toolkit (AST) or Rational Web Developer, menu choices for the J2EE Perspective might not be enabled. To assemble code artifacts into J2EE modules that can be deployed onto an application server, you must work in the J2EE Perspective. This article explains how to configure your assembly tool for work on J2EE modules and specify a target server supported by WebSphere Application Server.

Before you can configure an assembly tool, you must install and start the assembly tool.

Configuring an assembly tool consists of ensuring that menu choices for the J2EE Perspective are enabled and specifying a target server supported by WebSphere Application Server. When you first start AST and Rational Web Developer, menu choices for the J2EE Perspective might not be enabled, meaning that you cannot assemble code artifacts into deployable J2EE modules. You perform the steps in this article when you cannot work in the J2EE Perspective or when you need to specify a new target server for your modules.

1. Enable menu choices for the J2EE Perspective.
 - a. Click **Window > Customize Perspective**.
 - b. In the Customize Perspective dialog, select **J2EE**. Also select **EJB**, **Web**, and any other categories that you might need. Then, click **OK**.
 - c. Click **Window > Open Perspective > Other > J2EE > OK**.
2. Select the Project Explorer view for your work.
 - a. Click **Window > Show View > Other > J2EE > Project Explorer > OK**.

The Project Explorer view is displayed in a panel of the assembly tool.

3. Define a target server for your modules.
 - a. Click **Window > Preferences > Server > Installed Runtimes**. The Installed Server Runtime Environments section of the Preferences dialog is displayed.
 - b. Optional: Click **Search**. The assembly tool searches the local machine for an installation of WebSphere Application Server. If a list of installations is displayed, select the desired target server and click **OK** in the displayed dialog; then click **OK** in the Preferences dialog. If no installations of WebSphere Application Server are found, continue completing the steps for this task.
 - c. Click **Add**. The New Server Runtime dialog is displayed.
 - d. Select the target server run time. For work on Version 6 modules, click **WebSphere Application Server v6.0 > Next**. Note that menu choices and deployment descriptor panels for application assembly services such as ActivitySession, Application profiling, Internationalization, and Last participant support are not available unless you specify a WebSphere Application Server target server. The *product_name* Runtime panel is displayed.
 - e. For **Installation directory** or **Location**, specify the location of the target server. 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 modules; however, you cannot deploy modules to the target server using the assembly tool.

Tip: If you specify a false location for a WebSphere Application Server v6.0 target server, use the assembly tool to assemble modules and then use WebSphere Application Server to deploy the modules.

If you selected for a target server **J2EE Runtime Library**, a generic J2EE container, specify a directory that contains `.jar` files such as `\bin\lib` directory for a Java development kit installation.

- f. Click **Finish**. The target server name and type is added to the table in the Installed Server Runtime Environments section.
- g. In the Preferences dialog, click **OK**.

When you configure J2EE modules, the target server that you selected is shown in the list of available target servers. If you selected **WebSphere Application Server v6.0** for the target server, menu choices and deployment descriptor panels for WebSphere Application Server services are enabled.

When you configure a J2EE module, select a target server for the module from the list of available target servers.

To change the list of available target servers, click **Window > Preferences > Server > Installed Runtimes** to go to the Installed Server Runtime Environments section of the Preferences dialog. Use the **Add, Edit, Remove** or **Search** options to change the list of target servers as needed.

To change the target server for a module, right-click on the module in the Project Explorer view and click **Properties > Server**. Change the value for **Target runtime** to select a different target server for the module. In the assembly tools, the terms *target server* and *target runtime* have the same meaning.

Archive support in Version 6.0

The following archives and Web components are supported in Version 6.0:

- Java 2 Platform, Enterprise Edition (J2EE) Version 1.3 and 1.4 enterprise archive (EAR) files
- Enterprise bean (EJB) 2.0 JAR files
- Servlet 2.3 Web archive (WAR) files
- Application client 1.3 and 1.4 JAR files
- Connector 1.0 resource adapter archive (RAR) files

These archive files and Web components are back-level and can be read but not created or changed:

- J2EE 1.2 EAR files
- EJB 1.1 JAR files
- Servlet 2.2 WAR files
- Application client 1.2 JAR files

Migrating code artifacts to an assembly tool

You can migrate enterprise archive (EAR), Web archive (WAR), enterprise bean (EJB) JAR, application client JAR, resource adapter archive (RAR) files created with the Application Assembly Tool (AAT), Assembly Toolkit or a different tool to the Application Server Toolkit or Rational Web Developer.

This article assumes you have previously assembled code artifacts that you want to work with in an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also assumes that you have started the assembly tool and have configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules.

You can migrate EAR, WAR, enterprise bean JAR, application client JAR, or RAR projects or files to an assembly tool using the J2EE Migration wizard or by importing the files.

1. **Optional:** Migrate a project from J2EE 1.2 to 1.3 or 1.4, or from J2EE 1.3 to 1.4, using the J2EE Migration wizard. As part of the migration, you can migrate CMP 1.x beans to CMP 2.x beans. The J2EE Migration wizard is similar to the earconvert batch utility or the **File > ConvertEar** option of the AAT.
 - a. In the Project Explorer view, right-click the enterprise application project (EAR file) you want to migrate.
 - b. Click **Migrate > J2EE Migration Wizard**.
 - c. Follow the instructions in the wizard.
2. Import an enterprise application.
3. Import a WAR file.
4. Import an application client file.
5. Import an enterprise bean JAR file.
6. Import a resource adapter RAR file. RAR files are also known as *connectors*.

After importing your previously assembled files into an assembly tool, do the following:

1. Verify the archive files.

2. Deploy the imported and assembled module or application.

Importing enterprise applications

Importing an enterprise archive (EAR) file migrates the EAR file to the assembly tool and defines a new enterprise application project using the tool.

This article assumes that you have an existing enterprise application that you want to work on in an assembly tool such as Application Server Toolkit (AST) or Rational Web Developer and that you want to deploy onto an application server.

You can import an EAR file and define a new enterprise application project 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. Click **File > Import > EAR file > Next**. Alternatively, you can right-click **Enterprise Applications** in a view such as the Project Explorer view and click **Import > EAR file**. Or, on Windows platforms, you can drag the EAR file and drop it on a view.
4. In the Import dialog, define the EAR file:
 - a. Specify the EAR file to import. Use **Browse** to locate the EAR file and specify its full path name.
 - b. Optional: Specify a new enterprise application project name. A project name is assigned automatically. The project name you specify must be unique within the directory.
 - c. Specify whether to import an EAR project.
 - d. Optional: Specify whether to overwrite and delete existing resources.
 - e. Select a target server. Select the **WebSphere Application Server v6.0** target server to use services such as the following:
 - ActivitySession
 - Application profiling
 - Internationalization
 - Last participant support
 - f. Optional: Click **Next**.
 - g. Optional: Specify whether to allow nested projects to be overwritten.
 - h. Optional: Specify whether to import a file for partial development of an EAR file.
 - i. Optional: Select JAR files to be imported.
 - j. Optional: Specify the project location. Use **Browse** to locate the directory for the project files.
 - k. Click **Finish**.

Files for the enterprise application are shown in the Project Explorer view under **Enterprise Application**.

Update the enterprise application as needed and deploy it to an application server.

Importing WAR files

Importing a Web application archive (WAR) file migrates the WAR file to an assembly tool.

This article assumes you have assembled a WAR file and want to work with it in an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also assumes that you have started the assembly tool and have configured the assembly tool for work on **Web** and Java 2 Platform, Enterprise Edition (J2EE) modules.

Importing an existing WAR file into an assembly tool migrates the WAR file to the assembly tool so you can further configure and assemble a deployable Web application.

1. Click **File > Import > WAR file > Next**. Alternatively, you can right-click **Dynamic Web Projects** in a view such as the Project Explorer view and click **Import > WAR file**. Or, on Windows platforms, you can drag the WAR file and drop it on a view.
2. In the Import dialog, define a Web project:
 - a. Specify a WAR file name. Use **Browse** to locate the WAR file and specify its full path name.
 - b. Specify a Web project name. For example, if you are importing the HelloWorld.war file, you might name the project HelloWorld. Click **New** and specify HelloWorld for the project name.
 - c. Optional: Specify whether to overwrite and delete existing resources without warning.
 - d. Select a target server. To use an application assembly service of WebSphere Application Server, select the **WebSphere Application Server v6.0** target server. Available assembly services include ActivitySession, Application profiling, Internationalization, and Last participant support.
 - e. If you want to add Web components to an enterprise application (EAR file), select **Add module to an EAR project**.
 - f. Optional: Specify a new or existing enterprise application (EAR) project to be associated with your Web project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project is created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
 - g. Click **Finish**.

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

After importing a WAR file, you can edit Web module deployment descriptors as needed and deploy the module or its application to an application server.

Importing client applications

You can migrate an existing application client JAR file to an assembly tool by importing the JAR file.

This article assumes you have assembled an application client JAR file and want to work with it in an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also assumes that you have started the assembly tool and have configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules.

Importing an existing application client JAR file into an assembly tool migrates the JAR file to the assembly tool so you can further configure and assemble an application client.

1. Click **File > Import > App Client JAR file > Next**. Alternatively, you can right-click **Application Client Projects** in a view such as the Project Explorer view and click **Import > App Client JAR file**. Or, on Windows platforms, you can drag the application client JAR file and drop it on a view.
2. In the Import dialog, specify the application client file and project name:
 - a. Specify the application client JAR file to be imported. Use **Browse** to locate the JAR file and specify its full path name.
 - b. Specify an application client project name. For example, if you are importing the HelloWorld.jar file, you might name the project HelloWorld. Click **New** and specify HelloWorld for the project name.
 - c. Select a target server. To use application assembly services of WebSphere Application Server, select the **WebSphere Application Server v6.0** target server.
 - d. If you want to add application client components to an enterprise application (EAR file), select **Add module to an EAR project**.

- e. Specify a new or existing enterprise application (EAR) project to be associated with your application client project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project is created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
- f. Optional: If you are creating a new enterprise application project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and JAR file dependencies. On the Module Dependencies page, select dependent JAR files or modules within the associated enterprise application project. This updates the runtime class-path and Java project build path with the appropriate JAR files. Application client modules, EJB modules, and Web modules can all have dependencies on EJB modules or utility JAR files. Modules cannot depend on WAR or application client JAR files.
- g. Click **Finish**.

A new application client project is created, reflecting the J2EE folder structure that specifies the location of application client content files, class files, class paths, the deployment descriptor, and supporting metadata. Files for the application client project are shown in the Project Explorer view under **Enterprise Applications** and **Application Client Projects**.

After importing an application client project, you can edit the application client deployment descriptor if default properties are not sufficient. In the Client Deployment Descriptor editor, you can add enterprise bean, resource, or resource environment references as well as 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 > Defining J2EE application clients**.

Importing EJB files

Importing an enterprise bean (EJB) Java archive (JAR) file migrates the EJB JAR file to an assembly tool.

This article assumes you have assembled an EJB JAR file and want to work with it in an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also assumes that you have started the assembly tool and have configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules.

Importing an existing EJB JAR file into an assembly tool migrates the JAR file to the assembly tool so you can further configure and assemble an application that contains the EJB module.

1. Click **File > Import > EJB JAR file > Next**. Alternatively, you can right-click **EJB Projects** in a view such as the Project Explorer view and click **Import > EJB JAR file**. Or, on Windows platforms, you can drag the enterprise bean JAR file and drop it on a view.
2. In the Import dialog, define the EJB JAR file and project:
 - a. Specify the enterprise bean JAR file to import. Use **Browse** to locate the JAR file and specify its full path name.
 - b. Specify an EJB project name. For example, if you are importing the HelloWorld.jar file, you might name the project HelloWorld. Click **New**, specify HelloWorld for the project name, specify whether you want to use the EJB 1.1, 2.0 or 2.1 specification, and click **Next**.
 - c. Optional: Specify whether to overwrite and delete existing resources without warning.
 - d. Select a target server. To use an application assembly service of WebSphere Application Server, select the **WebSphere Application Server v6.0** target server. Available assembly services include ActivitySession, Application profiling, Internationalization, and Last participant support.

- e. If you want to add EJB components to an enterprise application (EAR file), select **Add module to an EAR project**.
- f. Specify a new or existing enterprise application (EAR) project to be associated with your EJB project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project is created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
- g. Specify whether you want to add support for annotated Java classes.
- h. Click **Finish**.

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

After importing an EJB JAR file, you can edit EJB deployment descriptors as needed and deploy the module or its application to an application server.

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**.

For more information, refer to articles under **EJB assembly** in the **Application Server Toolkit** information center that accompanies this information center.

Importing RAR files or connectors

Importing a resource adapter archive (RAR) file, or *connector*, migrates the RAR file to an assembly tool.

This article assumes you have assembled a RAR file and want to work with it in an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer.

This article also assumes that you have started the assembly tool and have configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules.

Importing an existing RAR file into an assembly tool migrates the RAR file to the assembly tool so you can further configure and assemble define a connector module.

1. Click **File > Import > RAR file > Next**. Alternatively, you can right-click **Connector Projects** in a view such as the Project Explorer view and click **Import > RAR file**. Or, on Windows platforms, you can drag the connector Java archive (JAR) file and drop it on a view.
2. In the Import dialog, define a connector project:
 - a. Specify the name of the RAR file to import. Use **Browse** to locate the RAR file and specify its full path name.
 - b. Specify a connector project name. For example, if you are importing the HelloWorld.rar file, you might name the project HelloWorld. Click **New**, specify HelloWorld for the project name, and click **Next**.
 - c. Optional: Specify whether to overwrite and delete existing resources without warning.
 - d. Select a target server. To use an application assembly service of WebSphere Application Server, select the **WebSphere Application Server v6.0** target server. Available assembly services include ActivitySession, Application profiling, Internationalization, and Last participant support.
 - e. If you want to add connector components to an enterprise application (EAR file), select **Add module to an EAR project**. If you want the connector project to stand alone, do not add the module to an EAR project.
 - f. Specify a new or existing enterprise application (EAR) project to be associated with your connector project for purposes of deployment. Select an existing enterprise application project from the

drop-down list or type a new project name. Or, click **New** and create a new enterprise application. The name must be unique among EAR files in the directory.

- g. Click **Finish**.

A new connector project is created. Files for the connector project are shown in the Project Explorer view under **Enterprise Applications** and **Connector Projects**.

After importing a RAR file, you can edit connector deployment descriptors as needed and deploy the module or its application to an application server.

When deploying the RAR file onto a server, 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.

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 > Connector projects**.

Creating enterprise applications

Enterprise applications are Java 2 Platform, Enterprise Edition (J2EE) applications that can be deployed onto an application server. Enterprise applications are projects comprised of enterprise bean modules, Web modules, connector modules, application client modules, or JAR files containing dependent classes or other components required by the application. Enterprise applications are called *EAR files* because enterprise application projects are stored in enterprise archive (EAR) files.

This article assumes that you have created and unit tested code artifacts that you want to assemble in an enterprise application and deploy onto an application server. A deployable enterprise application can consist of the following code artifacts:

- Enterprise beans
- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (*connector*) implementations
- Application clients
- Other supporting classes and files

Before you can deploy your archive files onto an application server, you must assemble them in a J2EE enterprise application. This article describes how to create an enterprise application archive (EAR) file using the Application Server Toolkit or Rational Web Developer assembly tools.

Creating an enterprise application in an assembly tool consists of the following:

1. Creating an enterprise application project.
2. Adding (importing) archive files such as Web application archives (WAR), resource adapter archives (RAR), enterprise bean (EJB) JAR files, and application client archives (JAR) files.
 1. Start an assembly tool.
 2. If you have not done so already, configure the assembly tool for work on J2EE modules.
 3. Click **File > New > Project > J2EE > Enterprise Application Project > Next**. Or, if you have created an enterprise application project before, click **File > New > Enterprise Application Project**.
 4. In the New Enterprise Application Project dialog, create an enterprise application project:
 - a. Specify an EAR file name and location.

- b. To change the default project location, click **Browse** and specify a new location.
- c. Click **Show Advanced** to display hidden settings.
- d. Specify whether you want an EAR file that supports J2EE 1.2, 1.3 or 1.4.
- e. Select a target server. Select the **WebSphere Application Server v6.0** target server to use services such as the following:
 - ActivitySession
 - Application profiling
 - Internationalization
 - Last participant support
- f. Click **Next**.
- g. On the EAR Module Projects page, select the existing modules that you want to add to the new enterprise application project. To create new modules for this enterprise application:
 - 1) Click **New Module**.
 - 2) On the New Module Project page, select **Create default module projects** to create modules for application client, enterprise bean (EJB), Web or connector projects. You can use the default project names for the modules or specify different project names. If you clear the **Create default module projects** check box, you can select a single module type and proceed with the proper wizard for that project type.
 - 3) Click **Finish** to create the project modules and add their names to the list of available modules on the EAR Module Projects page.
- h. Click **Finish**.

Files for the enterprise application are shown in the Project Explorer view under **Enterprise Application**.

Verify the contents of the new enterprise application in either of the following ways:

- In the Project Explorer view, expand **Enterprise Application** and view the new EAR file and its contents.
- Click **Window > Show View > Navigator** to see the associated files for the enterprise application in a Navigator view.

Creating Web applications

Web applications are Java 2 Platform, Enterprise Edition (J2EE) applications that can be deployed onto an application server. Web applications are projects 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. Web applications are called *WAR files* because enterprise application projects are stored in Web application archive (WAR) files.

This article 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.

In the Application Server Toolkit (AST) and Rational Web Developer assembly tools, you create and maintain resources for Web applications in Web projects. There are two types of Web projects, dynamic and static. Dynamic web projects can contain dynamic J2EE resources such as servlets, JSP files, filters, and associated metadata, in addition to static resources such as images and HTML files. Static Web projects only contain static resources.

Dynamic Web projects are always embedded in enterprise application projects. Creating a Web project in an assembly tool requires that an enterprise application (EAR) project exist, or the assembly tool creates one for you. Creating a Web project updates the `application.xml` deployment descriptor of the specified enterprise application project to define the Web project as a module element. If you are importing a WAR file rather than creating a Web project anew, the WAR Import wizard requires that you specify a Web project, which already requires an EAR project.

This article describes how to create a dynamic Web project using an assembly tool. For instructions on how to create a static Web project, see the Application Server Toolkit or Rational Web Developer online help, or the **Application Server Toolkit** information center that accompanies this 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 **Web** capabilities are enabled.
3. Click **File > New > Project > Web > Dynamic Web Project > Next**. Or, if you have created a Web project before, click **File > New > Dynamic Web Project**.
4. On the Dynamic Web Project page:
 - a. Specify a Web project (WAR file) name.
 - b. Specify a location for the WAR file. To change the default WAR file location, click **Browse** and specify a new location.
 - c. Click **Show Advanced** to display hidden settings.
 - d. Specify whether you want a WAR file that supports 2.2, 2.3 or 2.4 servlet version. Select the appropriate Sun Microsystems Servlet and JSP specification level for the dynamic elements you plan to include in your Web project. Any new servlets and JSP files that you expect to create should adhere to the latest specification level available; previous specification levels are offered to accommodate any legacy dynamic elements that you expect to import into the project.
 - e. Select a target server. Select the **WebSphere Application Server v6.0** target server to use Version 6 WebSphere Application Server capabilities.
 - f. If you want to add Web components to an enterprise application (EAR file), select **Add module to an EAR project**.
 - g. If you selected **Add module to an EAR project**, customize the Web project:
 - 1) Specify a new or existing enterprise application (EAR) project to be associated with your new Web project for purposes of deployment.

Note: If you want to add a Web project as a module to another enterprise application project in the future, open the `application.xml` editor for the enterprise application project and select **Add** on the General page.
 - 2) Provide a **Context Root** value. The context root is the Web application root, the top-level directory of your application when it is deployed to a Web server. The default value is the name of your Web project.

Note: You can change the context root after you create a project using the project Properties dialog, which you access from the project context menu.
 - 3) Specify whether you want to add support for annotated Java classes to the Web application.
 - h. Optional: If you are creating a new Web project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and file dependencies.
5. Click **Finish**.

A new Web project is created, reflecting the J2EE folder structure that specifies the location of Web content files, class files, class paths, the deployment descriptor, and supporting metadata. Files for the Web project are shown in the Project Explorer view under **Enterprise Applications** and **Dynamic Web Modules**.

You can now begin creating or importing content for your Web project using the New File wizards or the Import wizards available from the **File** menu or from the popup menu for the Web module. For example, right-click the Web module, click **Import > WAR file** and specify the WAR file to import.

Creating 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 article 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.

In an Application Server Toolkit (AST) or Rational Web Developer assembly tool, you can create and test enterprise beans that conform to the distributed component architecture defined in the Sun Microsystems Enterprise JavaBeans (EJB) specification and that support extended functionality for WebSphere Application Server.

You can create enterprise beans (either with or without inheritance) such as session beans, container-managed persistence (CMP) entity beans, bean-managed persistence (BMP) entity beans, or message-driven beans. Using the EJB deployment descriptor editor of an assembly tool, you can set deployment descriptor and assembly properties for enterprise beans.

This article describes how to create an EJB project (or EJB module) 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. Ensure that the **J2EE** capability is enabled.
3. Click **File > New > Project > EJB > EJB Project > Next**. Or, if you have created a J2EE project before, click **File > New > EJB Project**.
4. In the New EJB Project dialog:
 - a. Name the EJB project and specify its location. To change the default project location, click **Browse** and specify a new location. If you specify a non-default project location that is already being used by another project, you cannot create the project.
 - b. Select the EJB specification version to which you want your EJB project to adhere. If you plan on using EJB 2.1 enterprise beans, you must specify an EJB 2.1 project. You can add EJB 1.1 enterprise beans to an EJB 2.1 project. An EJB 2.1 project must exist in a J2EE 1.4 enterprise application project. An EJB 2.0 project can exist in a J2EE 1.4 or 1.3 enterprise application project. Your available options can differ, depending on the J2EE preferences defined.
 - c. Select a target server. Select the **WebSphere Application Server v6.0** target server to use Version 6 WebSphere Application Server capabilities.
 - d. If you want to add EJB components to an enterprise application (EAR file), select **Add module to an EAR project**.
 - e. Specify a new or existing enterprise application (EAR) project to be associated with your new EJB project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project is created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
 - f. Specify whether you want to add support for annotated Java classes to the EJB module.
 - g. Specify whether you want to create a default stateless session bean.
 - h. Optional: If you are creating a new enterprise application project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to create an EJB client JAR file from your existing enterprise bean. On the EJB Client JAR Creation page, specify a URI, name and project location for the EJB client JAR file. If you have no module dependencies to specify, skip the rest of this step; otherwise, click **Next**. On the Module Dependencies page, select dependent JAR files or modules within the associated enterprise application project. This updates the runtime class-path and Java project build path with the appropriate JAR files. Application client modules, EJB modules, and Web modules can all have dependencies on EJB modules or utility JAR files. Modules cannot depend on WAR or application client JAR files.
 - i. Click **Finish** to create the EJB project.

A new EJB project is 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 project are shown in the Project Explorer view under **Enterprise Applications** and **EJB Projects**.

After you have an EJB project to hold enterprise beans, you can do the following:

- Create or import enterprise beans to your EJB project.
- Add methods to the home and remote interfaces.
- Add custom finders.
- Add and define additional CMP fields.
- Add relationships.
- Edit the EJB deployment descriptor if default properties are not sufficient.
- Create EJB access beans and use them to create your client application.
- Map enterprise beans to RDB tables.

For detailed instructions on creating CMP fields or CMP finder methods for entity beans, relating CMP fields, adding methods to interfaces, or managing enterprise beans, refer to articles under **EJB assembly** in the **Application Server Toolkit** information center that accompanies this information center.

Creating application clients

Application clients are the client applications that use a servlet to communicate with an enterprise bean, with the servlet residing on the same machine as an application server.

This article assumes that you have created and unit tested an application client program that you want to assemble in an enterprise application and deploy onto an application server.

Application clients consist of several models. This article applies to application clients based on the Java 2 Platform, Enterprise Edition (J2EE) model.

A *J2EE application client* is a Java application program that accesses enterprise beans, Java DataBase Connectivity (JDBC) application programming interfaces, and Java Message Service (JMS) message queues. The J2EE application client program runs on networked client systems. The 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.

You can use a J2EE application client to develop an application program, assemble the program into an application client project, deploy the project as a client application (JAR file), and launch the client application.

To create a deployable client application, use an Application Server Toolkit (AST) or Rational Web Developer assembly tool to create and add an application client project to a new or existing enterprise application project.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that the **J2EE** capability is enabled.
3. Click **File > New > Project > J2EE > Application Client Project > Next**. Or, if you have created a J2EE project before, click **File > New > Application Client Project**.
4. In the New Application Client Project dialog:
 - a. Name the application client project and specify its location. To change the default project location, click **Browse** and specify a new location. If you specify a non-default project location that is already being used by another project, you cannot create the project.
 - b. Click **Show Advanced** to display hidden settings.
 - c. Select the J2EE specification version to which you want your project to adhere.

- d. Select a target server. Select the **WebSphere Application Server v6.0** target server to use Version 6 WebSphere Application Server capabilities.
- e. If you want to add application client components to an enterprise application (EAR file), select **Add module to an EAR project**.
- f. Specify a new or existing enterprise application (EAR) project to be associated with your new application client project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project is created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
- g. Specify whether you want to create a default Main class.
- h. Optional: If you are creating a new enterprise application project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and JAR file dependencies. On the Module Dependencies page, select dependent JAR files or modules within the associated enterprise application project. This updates the runtime class-path and Java project build path with the appropriate JAR files. Application client modules, EJB modules, and Web modules can all have dependencies on EJB modules or utility JAR files. Modules cannot depend on WAR or application client JAR files.
- i. Click **Finish**.

A new application client project is created, reflecting the J2EE folder structure that specifies the location of application client content files, class files, class paths, the deployment descriptor, and supporting metadata. Files for the application client project are shown in the Project Explorer view under **Enterprise Applications** and **Application Client Projects**.

After creating an application client project, you can edit the application client deployment descriptor if default properties are not sufficient. In the Client Deployment Descriptor editor, you can add enterprise bean, resource, or resource environment references as well as 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 > Defining J2EE application clients**.

Creating connector modules

A *resource adapter archive* (RAR file) is a Java archive (JAR file) used to package a resource adapter for the J2EE Connector Architecture (JCA) for WebSphere Application Server.

This article 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 Web 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 *Enterprise Information System (EIS)* is a set of related classes that lets an application access a resource such as data, or an application on a remote server, often called a resource adapter.

This article describes how to create a connector module 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. Ensure that the **J2EE** capability is enabled.

3. Click **File > New > Project > J2EE > Connector Project > Next**. Or, if you have created a J2EE project before, click **File > New > Connector Project**.
4. In the New Connector Project dialog:
 - a. Name the connector project and specify its location. To change the default project location, click **Browse** and specify a new location.
 - b. Click **Show Advanced** to display hidden settings.
 - c. Select the JCA specification version to which you want your project to adhere.
 - d. Select a target server. Select the **WebSphere Application Server v6.0** target server to use Version 6 WebSphere Application Server capabilities.
 - e. If you want to add resource adapter components to an enterprise application (EAR file), select **Add module to an EAR project**.
 - f. Specify a new or existing enterprise application (EAR) project to be associated with your new resource adapter project for purposes of deployment. Select an existing project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project will be created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the resource adapter, you must click **New** and create a new adapter.
 - g. Optional: If you are creating a new project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and JAR file dependencies. On the Module Dependencies page, select dependent JAR files or modules within the associated project.
 - h. Click **Finish** to create the connector project.

A new connector project is 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 > Connector projects**.

Editing deployment descriptors

A deployment descriptor is an extensible markup language (XML) file that describes how to deploy a module or application by specifying configuration and container options.

This article assumes that you have assembled code artifacts into a J2EE module that you want to deploy onto an application server.

When you create a module in an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer, the assembly tool creates deployment descriptor files for the 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.

Deployment descriptor editor	Resources modified in the editor
Application deployment descriptor editor	<ul style="list-style-type: none"> • application.xml • ibm-application-bnd.xmi • ibm-application-ext.xmi

Web deployment descriptor editor	<ul style="list-style-type: none"> • WEB-INF/web.xml • Binding information • IBM binding and extensions information such as ibm-web-bnd.xml and ibm-web-ext.xml files
Enterprise bean (EJB) deployment descriptor editor	<ul style="list-style-type: none"> • ejb-jar.xml • ibm-ejb-jar-bnd.xml • ibm-ejb-jar-ext.xml • ibm-ejb-access-bean.xml
Client deployment descriptor editor	<ul style="list-style-type: none"> • application-client.xml • ibm-application-client-bnd.xml • ibm-application-client-ext.xml
Web services editor	<ul style="list-style-type: none"> • webservices.xml • ibm-webservices-bnd.xml • ibm-webservices-ext.xml
Resource adapter deployment descriptor editor	<ul style="list-style-type: none"> • ra.xml • j2c_plugin.xml

This article describes how to edit a deployment descriptor using an assembly tool deployment descriptor editor.

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE modules.
3. In a Project Explorer view (**Window > Show View > Project Explorer**), right-click the module with deployment descriptor values that you want to browse or edit, and click **Open With > Deployment Descriptor Editor**. A deployment descriptor editor for the module is displayed in a view. You can click tabs such as **Overview**, **Module**, **Security**, and **Source** at the bottom of the view to browse or edit specific deployment descriptor values. Clicking **Source** displays editable source code; it is preferable to edit values in fields on or accessible from the other tabs rather than edit the source code manually.

If you selected the **WebSphere Application Server v6.0** target server for a module, tabs and deployment descriptor panels for application assembly services such as the following are also available:

 - ActivitySession
 - Application profiling
 - Internationalization
 - Last participant support
4. Edit the deployment descriptor values as desired. For information on fields in the deployment descriptor editor, press **F1** and click a topic. For deployment of your application onto a server supported by WebSphere Application Server, consider specifying bindings such as Java Naming and Directory Interface (JNDI) names in the **WebSphere Bindings** section of module deployment descriptor editors. If you do not specify bindings during application assembly, you can specify them during installation of the application onto a server. If you do not specify bindings during assembly or installation, WebSphere Application Server can generate default bindings.
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.

You also can save changes to deployment descriptors at any time by pressing **Ctrl+S**.

Files for the enterprise application are shown in the Project Explorer view.

Verify the contents of the updated module and deploy the module to an application server.

Mapping enterprise beans to database tables

Mapping enterprise bean JAR files (EJB modules) to relational database (RDB) tables enables the EJB modules to access database resources.

This article assumes that you have created an enterprise application that has an EJB module and that you want this application to access one or more resources.

This article also assumes that you have started an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer and have configured the assembly tool for work on J2EE modules.

You can map enterprise bean JAR files (EJB modules) to relational database (RDB) tables using the EJB to RDB Mapping wizard of an assembly tool. The wizard creates EJB to RDB mappings for the following situations:

Existing enterprise bean but no database schema

Top Down mapping generates a default database schema and a mapping from one or more existing enterprise beans.

Existing database schema but no enterprise bean

Bottom Up mapping generates one or more enterprise beans and mappings from an existing database schema.

Existing enterprise bean and database schema

Meet In the Middle mapping matches existing enterprise beans with existing database tables. You can match by name, by name and type, or by neither.

Top-down and meet-in-the-middle mapping support multiple backends, making multiple deployments inside a single EJB module configurable at run time. Bottom-up mapping only supports a single backend. A *backend* can represent different database vendors, or simply alternative mappings and table qualifiers. If multiple backends exist, then current BackendID needs to be set in the EJB deployment descriptor editor (when working with EJB 2.0 beans). This mapping is used at run time when the JAR is installed on WebSphere Application Server. When deploying EJB 1.1 beans inside an EJB 2.0 project, the EJB 1.1 beans are deployed only once, using the first declared database and type. You specify a **Backend ID** in an EJB deployment descriptor editor under **WebSphere Bindings**. The **Backend ID** determines the persist classes that get loaded at deployment.

1. In the Project Explorer view, right-click the EJB module.
2. Click **EJB to RDB Mapping > Generate Map**.
3. After the wizard opens, press **F1** and select a type of mapping. The online help provides detailed information on generating a mapping.
4. For EJB 2.0 projects, on the EJB to RGB Mapping page specify whether you want to create a new backend (*Top Down*) or use an existing backend (*Bottom Up* or *Meet In the Middle*) where the schema exists in the backend but without a mapping file. If you previously generated a mapping, you can create and map unmapped elements or open the mapping editor to manually make changes. In EJB 2.0, your mapping and schema files make up a *backend* for EJB 2.0 projects. You can have multiple backend folders for each project; for example, one DB2 and one Oracle backend. The wizard uses one database backend only as the default, but you can define as many as you need.
5. Follow the instructions in the wizard and in the online help.
6. Click **Finish** to generate the mapping.

Files for the updated module are shown in the Project Explorer view.

After testing your module, you can deploy your module to an application server.

For EJB modules, 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**.

Mapping constraints for databases

Each of your Java 2 platform, Enterprise Edition (J2EE) applications (in an EAR file) can have more than one enterprise bean Container Managed Persistence (CMP) module in it. Each EJB CMP module (as a JAR file) can contain more than one CMP bean. You can configure each bean to connect to its own data source by using the different Java Naming and Directory Interface (JNDI) names of each data source.

However, there are some constraints:

- The deployment tool only enables you to map EJB CMP beans to different databases at the EJB JAR level. There are different backend ids for different database schema mapping. The product uses the attribute *backendId* to distinguish database specific code. For example, the code that accesses an Oracle database is different from that accessing a DB2 database. Even for the same vendor, different versions of their products can result in different generated code.
- At run time, WebSphere Application Server uses the *CurrentBackendId* attribute to access the correct back end related code. You can only configure the *CurrentBackendId* at the EJB JAR level. This means that **all** of the EJB CMP beans within a specific EJB JAR file (which contains all the CMR beans) must connect to the same type of data source (that is, the same vendor, same version).
- Even though you can connect EJB CMP beans to different databases of the same type, the process fails if your connections result in a *table join* between tables from different databases. Various actions such as EJB Query , Inheritance support, *readAhead..*, and so on, can result in a table join situation.
- The datasource needs to be an XA datasource to support multiple databases' connection within a transaction.

Verifying archive files

Verifying an archive file consists of validating code in the files.

This article assumes you have assembled an archive file into a module using an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer. Thus, this article assumes that you have done the following:

1. Started the assembly tool;
2. Configured the assembly tool for work on Java 2 Platform, Enterprise Edition (J2EE) modules; and
3. Migrated an existing module to the assembly tool or created a new module.

An assembly tool validates code when you request code validation manually, automatically during a resource change, and automatically during a build.

As part of validating the code, the validation checks for the following:

- Required deployment properties contain values.
 - Values specified for environment entries match their associated Java types.
 - In both enterprise archive (EAR) and Web archive (WAR) files:
 - The target enterprise bean of the link exists for enterprise bean (EJB) references.
 - The target role exists for security role references.
 - Security roles are unique.
 - Each module listed in the deployment descriptor exists in the archive for EAR files.
 - Files for icons, servlets, error and welcome pages listed in the deployment descriptor have corresponding files in the archive for WAR files.
 - For EJB modules:
 - All class files referenced in the deployment descriptor exist in the JAR file.
 - Method signatures for enterprise bean home, remote and implementation classes comply with the EJB 2.0 or 2.1 specification.
1. **Optional:** Specify whether you want automatic code validation during a resource change or during a build. The default is for automatic code validation.
 - a. In the Project Explorer view, right-click on a project.

- b. Click **Properties > Validation**.
 - c. Ensure that the **Run validation** options for builds and for automatic validation are selected. Select **Override validation preferences** to disable automatic code validation.
 - d. If you changed the **Validation** settings, click **Apply** or **OK**.
2. **Optional:** Specify validation options for a project. The default is to check all validators for a project during code validation. For an enterprise application project, the validators might be for DTD, EAR, Web services, XML, XML schema, or XSL files.
 - a. In the Project Explorer view, right-click the project containing the code that you want to validate.
 - b. Click **Properties > Validation**.
 - c. Select **Override validation preferences**.
 - d. Select the validators you want checked during code validation.
 - e. If you changed the **Validation** settings, click **Apply** or **OK**.
3. Right-click the project containing the code that you want to validate and click **Run Validation** to manually validate the code.

The results of the code validation are shown in a Tasks view. For information on the results, select an entry in the Tasks view, press F1, and click **Tasks view**.

If your module uses Web services, refer to *Generating code for Web service deployment*. Otherwise, you are ready to deploy the module onto the application server.

For EJB modules, 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**.

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.

This article also assumes that you have started an assembly tool such as the Application Server Toolkit (AST) or Rational Web Developer and have configured the assembly tool for work on Web services.

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

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 the information center 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

Programming instructions and examples

- The J2EE™ Tutorial
- Rational developer community
- Developing and testing a complete J2EE "Hello World" application with WebSphere Studio V5
- Getting to know WebSphere Studio Application Developer: Its capabilities, technologies, and relationship to the open-source Eclipse IDE
- Developing and Deploying an End-to-end J2EE Application to JBoss Application Server using WebSphere Studio V5
- JMS Applications with WebSphere Studio V5 -- Part 1: Developing a JMS Point-to-Point Application
- Java 2 Enterprise Edition: Books index

Programming specifications

- J2EE 1.4 specification
- EJB specifications
- Servlet specifications

Administration

- Application Client files
- Connector RAR files

Chapter 7. 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 run-time behavior of packaged applications of deployed applications.

This article 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 article “Class loading: Resources for learning” on page 1036 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.
6. **Optional:** Configure class preloading.

Class loaders

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.

The run-time environment of WebSphere Application Server uses the following class loaders to find and load new classes for an application in the following order:

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 contains the Java 2 Platform, Enterprise Edition (J2EE) application programming interfaces (APIs) of the WebSphere Application Server product in the `j2ee.jar` file.

Because the J2EE APIs are in this class loader, 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-specific extensions class loader

The WebSphere Application Server extensions class loader loads the WebSphere Application Server run-time and J2EE 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 Application Server 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 adapters 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. The WebSphere Application Server run time enables you to associate a shared library class path with an application.

Each class loader is a child of the previous class loader. That is, the application module class loaders are children of the WebSphere-specific 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 Application Server 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 preloading

The first time that a WebSphere Application Server process starts, the name of each run-time class that is loaded and the name of the JAR file that contains the class are written to a preload file. The names of non-runtime classes such as custom services, resource classes such as `db2java.zip`, classes on the JVM class path, and application classes are not written to the preload file. Subsequent startups of the process use the preload file to start the process more quickly.

Preload files have the `.preload` extension. WebSphere Application Server processes that have preload files include the following:

Process	Preload file name
Application server	<code>cell_name.node_name.server_name.preload</code>
startserver	<code>WsServerLauncher.preload</code>
launchClient	<code>launchClient.preload</code>

Running the `startserver server1` command causes the `startserver` command to use a `WsServerLauncher.preload` file and the server to use a `cell_name.node_name.server1.preload` file. Later, running a command such as `startserver server1 -script`, where the `-script` option creates a new script, uses the `cell_name.node_name.server1.preload` file only.

Preload files, by default, are installed in the `install_root` directory.

New classes required during the startup of a process are added to the preload file. Any classes that are removed from a process are ignored during subsequent startups. Although it is not necessary, an administrator can delete the preload file and force a refresh that removes the ignored classes from the file.

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:

Application class-loader policy

Application class loaders consist of EJB modules, dependency JAR files, resource adapters, and 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 class-loader policy

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.

Note: WebSphere Application Server class loaders never load application client modules.

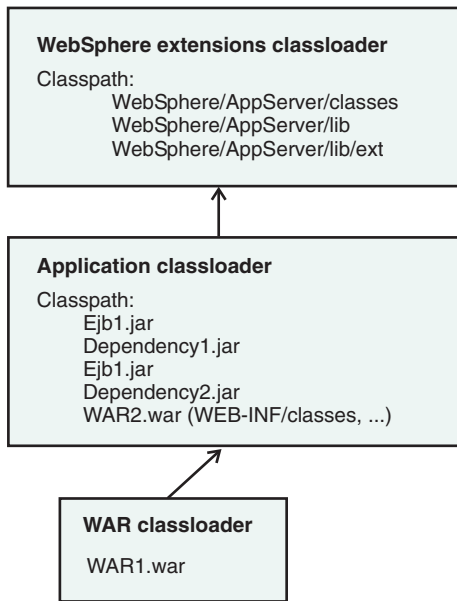
For each application server in the system, you can set the application class-loader policy to `Single` or `Multiple`. When the application class-loader policy is set to `Single`, then a single application class loader loads all EJB modules, dependency JAR files, and shared libraries in the system. When 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.

This application class loader can load the Web modules for each application if the class-loader policy of that WAR module is also set to `Application`. If the class-loader policy of the WAR module is set to `Application`, then the application loader loads the WAR module classes. If the 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.

Application class-loader policy: `Single`

```
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
```

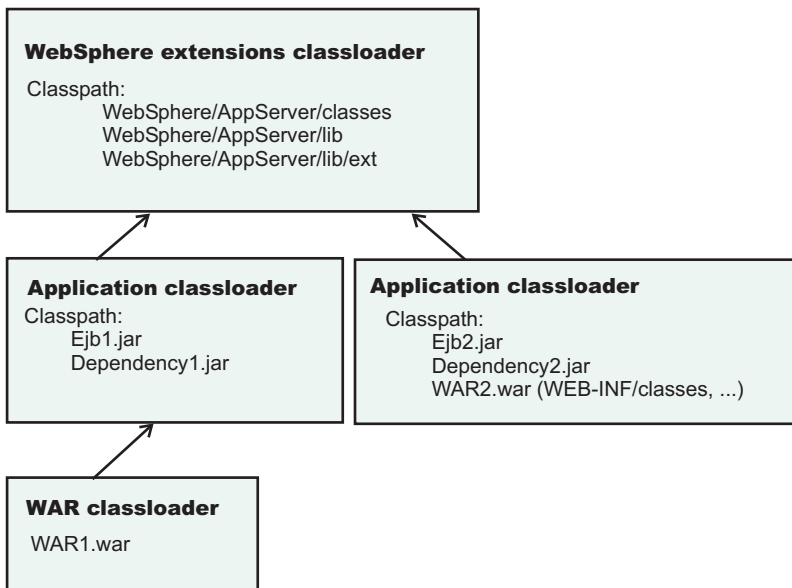


The following example shows that when the application class-loader policy of an application server is set to Multiple, each application on the server has its own class loader. An application class loader also loads its Web modules if the application WAR class-loader policy is set to Application. If the policy is set to Module, then a Web module uses its own class loader.

Application class-loader policy: Multiple

```

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

Two values for a class-loader mode are supported:

Parent First

The Parent 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

The Parent Last 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 application class-loader policy of an application server defines the mode for an application class loader.
- If the application class-loader policy of an application server is `Multiple`, the class-loader mode of an application defines the mode for an application class loader.
- If the WAR class-loader policy of an application is `Module`, the WAR class-loader policy of a Web module 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 article 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 mode. The Parent 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 Parent Last 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 mode

Specifies the class-loader mode when the application class-loader policy for the application server is Single.

You specify a policy of Single on the settings page for the application server, accessed by clicking **Servers > Application Servers > *server_name***. When the policy is Single, applications are not isolated from each other. A single application class loader contains all of the EJB modules, dependency JAR files, and shared libraries in the system.

The Parent 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 Parent Last value causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Specifying the Parent Last value enables an application class loader to override and provide its own version of a class that exists in the parent class loader.

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 mode

Specifies the class-loader mode when the application class-loader policy for the application server is `Single`.

You specify a policy of `Single` on the settings page for the application server, accessed by clicking **Servers > Application Servers > *server_name***. When the policy is `Single`, applications are not isolated from each other. A single application class loader contains all of the EJB modules, dependency JAR files, and shared libraries in the system.

The `Parent 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 `Parent Last` value causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Specifying the `Parent Last` value enables an application class loader to override and provide its own version of a class that exists in the parent class loader.

Data type String
Default Parent First

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 article 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*** to access the settings page for an enterprise application.

- Specify the class-loader mode for the application. The application class-loader mode 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 mode**:

Option	Description
Parent 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.
Parent Last	Causes the class loader to search in the application class loader first to load a class. By specifying Parent Last, your application can override classes contained in the parent class loader. Tip: Specifying the Parent Last value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes.

- 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 Module, which uses a separate class loader to load each WAR file. Setting the value to 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
Module	Uses a different class loader for each WAR file.
Application	Uses a single class loader to load all of the WAR files in your application.

- Specify whether to enable class reloading when application files are updated. By default, class reloading is not enabled. See the description for **Enable class reloading** in “Enterprise application settings” on page 1057 for details on enabling and disabling reloading. You might specify different values for EJB modules and for Web modules such as servlets and JavaServer page (JSP) files.
- Specify the number of seconds to scan the application’s file system for updated files. The value specified for **Reloading interval** 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.xml) 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).

- 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 article assumes that you installed a Web module on an application server.

Configure the class-loader mode 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 `Module`. If the policy is set to `Module`, then each Web module receives its own class loader whose parent is the application class loader. If the policy is set to `Application` on the settings page of an enterprise application, 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 **WAR class loader policy**, **Enable class reloading**, and **Reloading interval** can affect Web module class loading. If **WAR class loader policy** is set to `Module`, 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 `Application`, then the application class loader loads the Web module contents.
2. Click **Applications > Enterprise Application > *application_name* > Web modules > *Web_module_name*** to access the settings page for a deployed Web module.
3. Specify the class-loader mode 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 mode**:

Option	Description
Parent 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 <code>Parent first</code> . 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 <code>Parent first</code> is specified, the WAR class loader never receives a request to load a class.
Parent last	Causes the class loader to search in the WAR class loader first to load a class. By specifying <code>Parent last</code> , your WAR class loader can override classes contained in the parent application class loader. Tip: Specifying the <code>Parent last</code> value might result in <code>LinkageErrors</code> or <code>ClassCastException</code> messages if you have mixed use of overridden classes and non-overridden classes.

4. Click **OK**.

Save the changes to the administrative configuration.

Configuring class preloading

Class preloading affects how quickly a WebSphere Application Server process starts.

The first time that a WebSphere Application Server process starts up, the name of each class that is loaded and the name of the JAR file that contains the class are written to a preload file. Subsequent startups of the process use the preload file to start the process more quickly.

No configuring of class preloading is necessary.

However, an administrator can disable or enable preloading explicitly. By default, class preloading is enabled for WebSphere Application Server processes. To change the configuration for class preloading, an administrator sets new values for system properties.

- Disable class preloading. Set the Java virtual machine (JVM) system property `ibm.websphere.preload.classes` to `false`.
 1. In the administrative console, click **Servers > Application Servers > *server_name* > Java and Process Management > Process Definition > Java Virtual Machine** to access the Java Virtual Machine page.
 2. On the Java Virtual Machine page, specify `-Dibm.websphere.preload.classes=false` for **Generic JVM arguments**.
 3. Click **OK**.
 4. Save your administrative configuration.
 5. Stop the application server and then restart the application server.
 - Enable class preloading again. If you disabled class preloading, you can enable it again by doing either of the following:
 - Set the JVM system property to `true`. On the Java Virtual Machine page, specify `-Dibm.websphere.preload.classes=true` for **Generic JVM arguments**.
 - Remove the JVM system property that was created to disable class preloading. On the Java Virtual Machine page, remove the value `-Dibm.websphere.preload.classes=false` specified for **Generic JVM arguments**.
- After you change the JVM system property, click **OK**, save your administrative configuration, stop the application server, and then restart the application server.
- Regenerate a class preload file. Delete the `.preload` file for the WebSphere Application Server process. When the process next starts up, a new class preload file is generated for the process.

Class loading: Resources for learning

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 the information center 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

- J2EE Class Loading Demystified
- Understanding J2EE Application Server Class Loading Architectures

Programming instructions and examples

- Developing and Deploying Modular J2EE Applications with WebSphere Studio Application Developer and WebSphere Application Server
- IBM WebSphere Application Server Programming

Programming specifications

- Sun's J2EE™ Platform Specification
- Sun's J2EE™ Extension Mechanism Architecture

Chapter 8. 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 and assemble the module and configure the target server. Before choosing a server as a target for the module, ensure that the node version for the server 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.

- Install application files on an application server.
- Edit the administrative configuration for an application.
- 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.

After making changes to administrative configurations of your applications in the administrative console, ensure that you save the changes.

System applications

A *system application* is a J2EE enterprise application that is central to a WebSphere Application Server product.

Examples of system applications include *adminconsole* 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. Users cannot change the metadata for a system application such as its J2EE bindings or J2EE extensions, unless the metadata assigns users and groups for security purposes. Non-security related metadata requiring a change 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), and resource adapter (connector or RAR) files. 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.

Next, 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.

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 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 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, calls a 6.x API or uses a 6.x feature, then you must install the module on a 6.x deployment target.

Selecting options such as **Pre-compile JSP**, **Use Binary Configuration**, **Deploy Web services** or **Deploy enterprise beans** during application installation to a 6.x server or a 6.x deployment manager indicates that the application uses 6.x product features. You cannot deploy such applications on a 5.x deployment target. You must deploy such applications on a 6.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 on a WebSphere Application Server Version 5 product.

A *6.x deployment target* is a server on a WebSphere Application Server Version 6 product.

Table 13. 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
Application, EJB, 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
Resource adapter	JCA 1.5	Yes or No	6.x	6.x

Ways to install applications or modules

WebSphere Application Server provides several ways to install application files on a server.

Installable files include enterprise archive (EAR), enterprise bean (EJB), Web archive (WAR), resource adapter (connector or RAR), and application client modules.

Table 14. 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 1042.	Click Applications > Install New Application in the console navigation tree and follow instructions in the wizard.	All EAR, EJB, WAR, RAR, and application client files	Provides one of the easier ways to install application files. See “Preparing for application installation settings” on page 1048 for guidance. For applications that do not require changes to the default bindings, select the Generate Default Bindings option and then, on the Summary panel, click Finish .	Click Start on the Enterprise Applications page accessed by clicking Applications > Enterprise Applications in the console navigation tree.

Table 14. Ways to install application files (continued)

wsadmin scripts	Invoke AdminApp object <i>install</i> commands in a script or at a command prompt.	All EAR, EJB, WAR, RAR, and application client files	"Getting started with scripting" in the information center provides an overview of wsadmin.	<ul style="list-style-type: none"> Invoke the AdminApp <i>startApplication</i> command. Invoke the <i>startApplication</i> method on an ApplicationManager MBean using AdminControl.
Java application programming interfaces	Install programs by completing the steps in "Managing applications through programming" in the information center.	All EAR files	Use MBeans to install the application.	Start the application by calling the <i>startApplication</i> method on a proxy.
WebSphere rapid deployment Refer to articles under Rapid deployment of J2EE applications in the information center.	Briefly, do the following: <ol style="list-style-type: none"> 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, RAR, and application client files	WebSphere rapid deployment offers the following advantages: <ul style="list-style-type: none"> • 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.
Java programs	Code programs that use J2EE DeploymentManager (JSR-88) methods.	All J2EE modules, including EAR files and standalone EJB, WAR, RAR, and application client files	<ul style="list-style-type: none"> • Uses J2EE Application Deployment Specification (JSR-88). • Can customize modules using DConfigBeans. 	Call the J2EE DeploymentManager (JSR-88) method <i>start</i> 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, or Java programs that call J2EE DeploymentManager (JSR-88) methods. This article describes how to use the administrative console to install an application, EJB component, 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. The first of two Preparing for application installation pages is displayed.
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) or Java archive (JAR) file for installation.
 - b. If you are installing a standalone WAR file, specify the context root.
 - c. Click **Next**.
3. On 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 1048 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 (last step below). "Example: Installing an EAR file using the default bindings" on page 1053 provides sample steps.
4. On the **Step: Select installation options** panel, provide values for the following settings specific to WebSphere Application Server. Default values are used if you do not specify a value.
 - a. For **Pre-compile JSP**, specify whether to precompile JavaServer page (JSP) files as a part of installation. The default is not to precompile JSP files. Install onto a 6.x deployment target. If you select **Pre-compile JSP** and try installing your application onto a 5.x deployment target, the installation is rejected. For this option, install only onto a 6.x deployment target.
 - b. For **Directory to install application**, specify the directory to which the 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 *install_root/installedApps*; for example, C:\WebSphere\AppServer\profiles*profile_name*\installedApps*cell_name*.
 - c. For **Distribute application**, specify whether WebSphere Application Server expands or deletes application binaries in the installation destination. The default is to enable application distribution. As a result, when you save changes in the console, application binaries for newly installed applications are expanded to the directory specified. The binaries are also deleted when you uninstall and save changes to the configuration. 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 is expected to run.
 - d. For **Use Binary Configuration**, specify 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 EAR file. The default is not to use the binary

configuration. If you select **Use Binary Configuration**, your application files must be installed onto a 6.x deployment target. The files cannot be installed onto a 5.x deployment target.

- e. For **Deploy enterprise beans**, specify whether the EJBDeploy tool runs during application installation. The tool generates code needed to run EJB files. You must enable this setting in the following situations:
 - The EAR file was assembled using an assembly tool such as 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.
 - The EAR file was assembled using versions of the Application Assembly Tool (AAT) previous to Version 5.

Enabling this setting might cause the installation program to run for several minutes. Also, install onto a 6.x deployment target. If you select **Deploy enterprise beans** and try installing your application onto a 5.x deployment target, the installation is rejected. For this option, install only onto a 6.x deployment target.

- f. For **Application name**, name the application. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.
- g. For **Create MBeans for resources**, specify whether to create MBeans for various resources (such as servlets or JSP files) within an application when the application is started. The default is to create MBean instances.
- h. For **Enable class reloading**, specify whether to enable class reloading when application files are updated. The default is not to enable class reloading. For EJB modules or any non-Web modules, enabling class reloading sets `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 Web modules such as servlets and 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`. You can set `reloadingEnabled` to `true` when editing the extended deployment descriptors of your Web module in an assembly tool.

To disable reloading of a Web module, set the IBM extension `reloadingEnabled` in the `ibm-web-ext.xmi` file to `false`. Or, if the Web module has the IBM extension `reloadingEnabled` in the `ibm-web-ext.xmi` file set to `true`, enable class loading, and set the **Reload interval** property to zero (0).

- i. For **Reload interval in seconds**, specify the number of seconds to scan the application's file system for updated files. The default is the value of the reload 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 reload interval specified here takes effect only if class reloading is enabled.
- j. For **Deploy Web services**, specify 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. Note that if you select **Deploy** and try installing your application onto a 5.x deployment target, the installation is rejected. For this option, install only onto a 6.x deployment target.
- k. For **Validate Input off/warn/fail**, specify 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.

- l. For **Process embedded configuration**, specify 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`.
5. On the **Step: Map modules to servers** panel, for every module select a target server or cluster from the **Clusters and Servers** list. Select the check box beside **Module** to select all of the application modules or select individual modules. Ensure that you are installing your application onto an appropriate deployment target. You can specify Web servers as targets that route requests to the application. The plug-in configuration file `plugin-cfg.xml` for each Web server is generated based on the applications which are routed through it. 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.
6. If your application uses EJB modules, on the **Step: 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`.
7. If your application uses EJB modules that contain Container Managed Persistence (CMP) beans that are based on the EJB 1.x specification, for **Step: Provide default datasource mapping 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.
8. If your application has CMP beans that are based on the EJB 1.x specification, for **Step: Map datasources for all 1.x CMP**, 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.
9. If your application defines EJB references, for **Step: 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.
10. If your application defines resource references, for **Step: 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.
11. If your application uses Web modules, for **Step: 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.
12. If the application has security roles defined in its deployment descriptor then, for **Step: Map security roles to users/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 if predefined users such as **Everyone** or **All authenticated users** are mapped to it. To select specific users or groups from the user registry:
 - a. Select a role and click **Lookup users** or **Lookup groups**.

- b. On the Lookup users/groups panel displayed, enter search criteria to extract a list of users or groups from the user registry.
 - c. Select individual users or groups from the results displayed.
 - d. Click **OK** to map the selected users or groups to the role selected on the **Step: Map security roles to users/groups** panel.
13. If the application has Run As roles defined in its deployment descriptor, for **Step: Map RunAs roles to user**, 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**.
14. If your application contains EJB 1.x CMP beans that do not have method permissions defined for some of the EJB methods, for **Step: 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.
15. If your application contains message driven enterprise beans, for **Step: Provide Listener Ports or activation specification JNDI name for messaging beans**, provide a listener port name or an activation specification JNDI name for every message driven bean. 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**.
16. If your application uses EJB modules that contain CMP beans that are based on the EJB 2.x specification, for **Step: Provide default datasource mapping 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.
17. If your application has CMP beans that are based on the EJB 2.x specification, on the **Step: Map datasources for all 2.x CMP** 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.
18. 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 **Step: 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.
19. If the **Deploy enterprise beans** setting is enabled on the **Select installation options** panel, then you can specify options for the EJBDeploy tool on the **Step: 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 EJBDeploy tool. The tool is run on the EAR file during installation after you click **Finish**.
20. If your application contains resource environment references, for **Step: Map resource environment 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.

21. If your application defines **Run-As Identity** as *System Identity*, for **Step: Replace RunAs System to RunAs Roles**, 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.
22. If your application has resource references that map to resources that have an Oracle database doing backend processing, for **Step: Specify the isolation level for Oracle type provider**, 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.
23. If your application uses message driven beans, for **Step: Build message destination to administered objects**, specify the JNDI name of the J2C administered object to bind the message destination reference to the message driven beans.
24. If your application contains an embedded .rar file, for **Step: Map JCA resources to resources**, specify the name and JNDI name of each J2C connection factory, J2C administered object and J2C activation specification.
25. If your application contains an embedded .rar file, its activationSpec property has the value *Destination*, and its introspected type is *javax.jms.Destination*, for **Step: Bind J2CActivationSpec to Destination Jndi name**, specify the *jndiName* value for each activation bound to it.
26. If your application has EJB modules for which deployment code has been generated for multiple backend databases using an assembly tool, for **Step: Select a backend ID**, specify the backend ID representing the backend database to be used when the EJB module runs.
27. 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. Click **Finish**.

Several messages are displayed, indicating whether your application file is installing successfully.

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 `FileNotFoundException` 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. Associate any shared libraries that the application needs to the application.
2. 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

`install_root/installedApps/cell_name` by default or the directory that you designate. For a single-server installation, application files are copied to the destination directory when the changes are saved.

3. Start the application.
4. 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, update the application.

Preparing for application installation settings

Use this page to install an application (EAR file) or module (JAR 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, or Web module.

Path:

Specifies the fully qualified path to the .ear, .jar, 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. Only .ear, .jar, 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, 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 on a Windows machine might be `C:\WebSphere\AppServer\installableApps\test.ear`. If you are installing a stand-alone 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 server machine.

Context root:

Specifies the context root of the Web application (WAR).

This field is used only to install a stand-alone WAR 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`.

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.

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-linkName`. Otherwise the JNDI name is set to `eis/message-destination-refName`.
- 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.

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.

Connection Factory Bindings:

Specifies the default data source JNDI name.

If **Default connection factory bindings** is selected, specify the JNDI name for the default data source to be used with the bindings. Also specify the resource authorization.

Virtual Host:

Specifies the virtual host for the Web module.

Specific bindings file:

Specifies a bindings file that overrides the default binding.

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>
```

```

        </connection-factory>
    </ejb-binding>
</ejb-bindings>
</ejb-jar-binding>
</module-bindings>
</df1tbndngs>

```

Note: 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 df1tbndngs SYSTEM "df1tbndngs.dtd">
<df1tbndngs>
  <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/somA0</jndi-name>
            </message-destination-ref-binding>
          </message-destination-ref-bindings>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</df1tbndngs>

```

Note: 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 df1tbndngs SYSTEM "df1tbndngs.dtd">
<df1tbndngs>
  <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>
</df1tbndngs>

```

Note: 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 version="1.0"?>
<!DOCTYPE dfldbndngs SYSTEM "dfldbndngs.dtd">
<dfldbndngs>
<module-bindings>
<ejb-jar-binding>
<jar-name>YourEjbJar.jar</jar-name>
<ejb-bindings>
<ejb-binding>
<ejb-name>YourMDB</ejb-name>
<activation-spec-jndi-name>activationSpecJNDI</activation-spec-jndi-name>
</ejb-binding>
</ejb-bindings>
</ejb-jar-binding>
</module-bindings>
</dfldbndngs>
```

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 version="1.0"?>
<!DOCTYPE dfldbndngs SYSTEM "dfldbndngs.dtd">
<dfldbndngs>
<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>
</dfldbndngs>
```

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 version="1.0"?>
<!DOCTYPE dfldbndngs SYSTEM "dfldbndngs.dtd">
<dfldbndngs>
<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>
</dfldbndngs>
```

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_app1.ear and the file resides on a server at C:\sample_apps.
3. Now that a value is given for **Specify path**, on the first Preparing for application installation page, click **Next**.
4. 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.
5. If application security warnings are displayed, read the warnings and click **Continue**.
6. On the Install New Application page, click on **Summary**, the last step.
7. On the Summary panel, verify the cell, node, and server onto which the application files will install.
 - a. Beside the **Cell/Node/Server** option, click **Click here**.
 - b. On the **Map modules to servers** 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**.

Note that on the **Map modules to servers** 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.

Because my_app1.ear does not require any additional settings to complete an installation, the Summary panel is displayed again.

8. 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 `install_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 stand-alone modules during deployment using the DConfigBean class in the Java 2 Platform, Enterprise Edition (J2EE) Deployment API Specification (JSR-88).

This article 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.getIndexReadMethod().invoke
            (dcBean, new Object[]{new Integer(++index)});
    }
}

```

Enterprise application collection

Use this page to view and manage enterprise applications.

This page lists installed J2EE 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 *adminconsole* 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 J2EE 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 <i>Started</i> if the application starts up on all deployment targets, else the state changes to <i>Partial Started</i> .
Stop	Attempts to stop the processing of the application. After the application stops successfully, the state of the application changes to <i>Stopped</i> if the application stops on all deployment targets, else the state changes to <i>Partial Stopped</i> .
Install	Opens a wizard that helps you deploy an application or a module such as a .jar, .war or .rar file onto a server.
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.
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.
Remove File	Deletes a file of the deployed application or module. Remove File deletes a file from the WebSphere Application Server configuration repository and from the file system of all nodes where the file is installed.
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.

Name

Specifies the name of the installed (or deployed) application. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.

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 <i>Stopped</i> state to a <i>Started</i> state. Application is starting to run but is not fully running yet.
	Stopped	Application is not running.
	Partial Stop	Application is in the process of changing from a <i>Started</i> state to a <i>Stopped</i> state. Application has not stopped running yet.
	Unavailable	Status cannot be determined. An application with an unavailable status might, in fact, be running but have an unavailable status because the server running the administrative console cannot communicate with the server running the application.
	Not applicable	Application does not provide information as to whether it is running.

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. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.

Data type String

Application binaries

Specifies the directory to which the 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 `install_root/installedApps`; for example, `C:\WebSphere\AppServer\profiles\profile_name\installedApps\cell_name`.

You can specify an absolute path or use a pathmap variable such as `${MY_APPS}`. You can use a pathmap variable in any installation. A WebSphere Application Server variable `${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 page of the administrative console, accessed by clicking **Environment > WebSphere Variables**.

Data type String
Units Full path name

Use metadata from binaries

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 metadata from binaries** 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 distribution

Specifies whether WebSphere Application Server expands or deletes application binaries in the installation destination. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified. The binaries are also deleted when you uninstall and save changes to the configuration. 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.

This **Enable distribution** setting is the same as the **Distribute application** field on the application installation and update wizards.

Data type Boolean

Default true

Validation

Specifies whether WebSphere Application Server examines the application references specified during application installation or updating and, if validation is enabled, warns the users 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 **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 **Validation** setting is the same as the **Validate Input off/warn/fail** field on the application installation and update wizards.

Data type String

Default warn

Class loader mode

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 Parent First. By specifying Parent Last, 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 Parent First and Parent Last. The default is to search in the parent class loader before searching in the application class loader to load a class.

Data type String

Default Parent First

WAR class loader policy

Specifies whether to use a single class loader to load all WAR files of this application or to use a different class loader for each WAR file.

The options are `Application` and `Module`. The default is to use a separate class loader to load each WAR file.

Data type	String
Default	Module

Enable class reloading

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

For EJB modules or any non-Web modules, selecting **Enable class reloading** sets `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 Web modules such as servlets and JavaServer page (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`. You can set `reloadingEnabled` to `true` when editing your Web module's extended deployment descriptors in an assembly tool.

To enable reloading of a Web module, where you also want reloading of EJB and non-Web modules enabled:

1. Set the IBM extension `reloadingEnabled` in the `ibm-web-ext.xmi` file to `true`.
2. Select this **Enable class reloading** property.
3. Set the **Reloading interval** property to a value greater than zero (for example, 1 to 2147483647).

To enable reloading of a Web module only, and not enable reloading of EJB or non-Web modules:

1. Set the IBM extension `reloadingEnabled` in the `ibm-web-ext.xmi` file to `true`.
2. Set the IBM extension `reload interval` attribute in the `ibm-web-ext.xmi` file to a value greater than zero (for example, 1 to 2147483647).
3. Do not select this **Enable class reloading** property.

To disable reloading of a Web module, set the IBM extension `reloadingEnabled` in the `ibm-web-ext.xmi` file to `false`. Or, if the Web module has the IBM extension `reloadingEnabled` in the `ibm-web-ext.xmi` file set to `true`, to disable reloading using the administrative console:

1. Select this **Enable class reloading** property.
2. Set the **Reloading interval** property to zero (0).

Data type	Boolean
Default	false

Reloading interval

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 **Reloading interval** 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 reloading interval specified here overrides the value specified in the IBM extensions for each non-Web module in the EAR file (which in turn overrides the reload interval specified in the IBM extensions for the application in the EAR file). The reloading interval attribute takes effect only if class reloading is enabled.

The range is from 0 to 2147483647.

Data type	Integer
Units	Seconds
Default	3

Starting weight

Specifies the order in which applications are started when the server starts. The application with the lowest starting weight is started first.

Data type	Integer
Default	1
Range	0 to 2147483647

Background application

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 WebSphere Application Server 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 MBean files for various resources (such as servlets or JSP files) within an application when the application starts. The default is to create MBean files.

Data type	Boolean
Default	true

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, this article assumes 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 article describes how to change the settings of an application or module using the administrative console.

- Change the settings of the application or module on the settings page for the enterprise application.
 1. Click **Applications > Enterprise Applications > *application_name*** in the console navigation tree.

2. Change the values for settings as needed. The settings page help provides detailed information on the settings and allowed values. When you installed the application or module, you specified most, if not all, of the settings values. After installation, the settings on this page that you are likely to change include the following:

Enable class reloading and Reloading interval	These settings control 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 page (JSP) files, a Web container reloads a Web module only when the IBM extension reloadingEnabled in the <code>ibm-web-ext.xml</code> file is set to <code>true</code> . Refer to the settings page help for detailed information on enabling or disabling class reloading.
Starting weight	If your application starts automatically when its server starts, this value controls how quickly the application starts. Starting weight specifies the order in which applications are started when the server starts. The application with the lowest starting weight is started first.
Background application	If your application starts automatically when its server starts, Background application 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.

3. Click **OK**.

- Map each module of your application to a target server. Specify the application servers or Web servers onto which to install modules of your application.
- 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.
 - Stateful session bean failover
 - Session management
 - Application profiles
 - Libraries or library references
 - Target mappings
 - Last participant support extension
 - Deployment descriptors
 - Publish WSDL files
 - Provide JMS and EJB endpoint URL information
 - Provide HTTP endpoint URL information
 - Map security roles to users/groups
 - Provide JNDI Names for Beans. For more information, refer to “Task overview: Using enterprise beans in applications” on page 88.
 - Map resource references to resources
 - Map EJB references to beans. For more information, refer to “Task overview: Using enterprise beans in applications” on page 88.
 - Map data sources for all 2.x CMP beans
 - Provide default data source mapping for modules containing 2.x entity beans. For more information, refer to “Creating and configuring a data source using the administrative console” on page 531.
 - Map virtual hosts for web modules. For more information, refer to “Configuring virtual hosts” in the information center.
 - Map modules to servers
 - Web modules
 - EJB modules
 - Connector 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.

Save changes to your administrative configuration.

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. 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`.

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 a deployer. When assembling the application and then installing it 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 assembling an application. After defining the bindings, the deployer can install the application onto a server supported by WebSphere Application Server without selecting to 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.

Note: 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* > Provide JNDI Names for Beans** 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.

During and after application installation, data sources can be mapped to 2.x entity beans on the Map data sources for all 2.x CMP beans panel and on the Provide default data source mapping for modules containing 2.x entity beans panel. After installation, click **Applications > Enterprise Applications > *application_name*** in the administrative console, then select **Map data sources for all 2.x CMP beans** or **Provide default data source mapping for modules containing 2.x entity beans**. 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.

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 > Map EJB references to beans** in the administrative console.

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	<code>java:comp/env/jdbc</code>
JMS connection factory	<code>java:comp/env/jms</code>
JavaMail connection factory	<code>java:comp/env/mail</code>
Uniform Resource Locator (URL) connection factory	<code>java:comp/env/url</code>

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* > Map resource references to resources** in the administrative console to access the Map resource references to 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.xml`.

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* > Map virtual hosts for Web modules** 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 spec 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.xml`.

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 **Provide Listener Ports or activation specification JNDI name for messaging 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" in the information center.

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 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.

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 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 article 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 Map modules to servers 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 Map modules to servers 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* > Map modules to servers** in the console navigation tree. The Selecting servers - Map modules to servers 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 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.

Save changes to your administrative configuration.

Starting and 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 article 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 article 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 <i>Started</i> . The status is changed to <i>partially started</i> 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 <i>Stopped</i> .

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**.
2. Specify a different value for **Starting weight**.

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 **Background application**.

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. Save the changes to the application configuration.

Disabling automatic starting of applications

By default, an installed application starts automatically when the server on which the application resides starts. You can disable the automatic starting of the application, and later enable the automatic starting again.

This article assumes that the enterprise application is installed on an application server and that the application starts automatically when the server starts.

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 Mapping settings page for your application. Click **Applications > Enterprise Applications > *application_name* > Target Mappings > *target_name***. The *target_name* is the server on which the application resides. You use the Target Mapping settings page to map an installed application or module to a server.
2. Clear the **Enabled** check box.
3. Click **OK**.
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, select the **Enabled** check box on the Target Mappings settings page for the application, click **OK**, and then save changes to the configuration.

Target mapping collection

Use this page to view mappings of deployed applications or modules to servers.

To view this administrative console page, click **Applications > Enterprise Applications > *application_name* > Target Mappings**.

Target

States the name of the target server to which the application or module maps. You specify the target on the Map modules to servers 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 on a WebSphere Application Server Version 5 product.

A *6.x deployment target* is a server on a WebSphere Application Server Version 6 product.

An application, enterprise bean (EJB) module, Web module or application client module developed for a WebSphere Application Server 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 run-time 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 run-time 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.

Status

Indicates whether the status of the target server is started, stopped or unavailable.

Target mapping settings

Use this page to map a deployed application or module to a server.

To view this administrative console page, click **Applications > Enterprise Applications > *application_name* > Target Mappings > *target_name***.

Target:

States the name of the target server to which the application or module maps. You specify the target on the Map modules to servers page accessed from the settings for an application.

Data type String

Enabled:

Indicates whether the application modules installed on the target server are started (or enabled) when the server starts. This sets the initial state of application modules. A `true` value indicates that the corresponding modules are enabled and thus are accessible when the server starts. A `false` value indicates that the corresponding modules are not enabled and thus are not accessible when the server starts.

Data type Boolean
Default true

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.

Refer to “Ways to update application files” on page 1073 and decide how to update your application files. You can update enterprise applications or modules using the administrative console or a wsadmin tool. Both ways provide identical updating capabilities. Further, in some situations, you can update applications or modules without restarting the application server.

Note that Version 6 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. The administrative console Server collection pages show the versions for servers. You can deploy J2EE 1.4 modules to Version 6.x servers only. You cannot deploy J2EE 1.4 modules to servers on Version 5.x nodes. Refer to “Installable module versions” on page 1040 for details.

This article describes how to update deployed applications or modules using the administrative console.

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. 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.
2. 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.
3. With the application selected on the Enterprise Applications page, click **Update**. The Preparing for application update page is displayed.
4. Under **Specify the EAR, WAR or JAR module to upload and install**:
 - a. Ensure that **Application name** refers to the application to be updated.
 - b. Under **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. Briefly, the options are as follows:

Full application

Replaces the installed (old) application with the updated (new) application on the server. If you select **Full application**, specify the path for the new .ear file. The path provides the location of the new .ear file before installation.

Single module

Adds a new module to, or replaces a module in, the installed application. Specify the path for the new Web module (.war), EJB module (.jar), or resource adapter module (.rar). The path provides the location of the new module before installation.

To replace a module, the value for **Relative path to module** (or 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 value for **Relative path to module** 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 module, specify a value for **Context root**.

Single file

Adds a new file to, or replaces a file in, the installed application. Specify the path for the new file. The path provides the location of the new file before installation.

To replace a file, the value for **Relative path to file** must match the path of the file to be updated in the installed application.

To add a new file to the installed application, the value for **Relative path to file** must *not* match the path of a file in the installed application. The value specifies the desired path for the new file.

The relative path to a file from the root of the application is the concatenation of the module path and the file path within the module. For example, if the file is `com/mycompany/abc.class` within the module `foo.jar`, then the relative file path is `foo.jar/com/mycompany/abc.class`.

Partial application

Updates 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.

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.

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.

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. Refer to *Preparing for application update settings* for more information.

After you select an option, specify a path. 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. Only .ear, .jar, or .war files are shown during the browsing.

During application updating, 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, 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 on a Windows machine might be C:\WebSphere\AppServer\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 server machine.

5. If you selected the **Full application** or **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.
6. Click **Finish**.
7. If you did not use the Map modules to servers page of the update wizard, after updating the application, map the installed application or module to servers. Use the Map modules to servers page accessed from the Enterprise Applications page.
 - a. Go to the Map modules to servers page. Click **Applications > Enterprise Applications > *application_name* > Map modules to servers**.
 - 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 only.

After the application file or module installs successfully, do the following:

1. Save the changes to your configuration.

When you update a full application in the single server (base) product, after you save the changes, the old version of the application is uninstalled and the new version is installed into the configuration. The application binaries for the old version are deleted from the destination directory and the new binaries are copied to the directory.
2. Examine the values specified for **Reload Enabled** and **Reload Interval** on the settings page for your enterprise application.

If reloading of application files is enabled and the reload interval is greater than zero (0), the application's files are reloaded after the application is updated. For Web modules such as servlets and JavaServer page (JSP) files, a Web container reloads a Web module only when the IBM extension reloadingEnabled in the `ibm-web-ext.xmi` file is also set to `true`. You can set `reloadingEnabled` to `true` when editing your Web module's extended deployment descriptors in an assembly tool.
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.
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.

Ways to update application files

You can update application files deployed on a server in several ways.

Table 15. Ways to update application files

Option	Method	Comments	Starting after update
Administrative console update wizard See "Updating applications" on page 1070.	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: <ul style="list-style-type: none">Use Full application to update an .ear file.Use Single module to update a .war, enterprise bean .jar, or connector .rar file.Use Single file to update a file other than an .ear, .war, 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	Invoke AdminApp object <i>install</i> commands with the <i>-update</i> option in a script or at a command prompt.	"Getting started with scripting" in the information center provides an overview of wsadmin.	Invoke the wsadmin <i>startApplication</i> command.
Java application programming interfaces. See "Using administrative programs (JMX)" in the information center.	Update deployed applications by completing the steps in "Managing applications through programming" in the information center.	Update an application in the following ways: <ul style="list-style-type: none">Update the entire applicationAdd to, update or delete multiple files in an applicationAdd a module to an applicationUpdate a module in an applicationDelete a module in an applicationAdd a file to an applicationUpdate a file in an applicationDelete a file in an application	Start the application by either of the following methods: <ul style="list-style-type: none">On the Enterprise Applications page, select the updated application and click Start.Invoke the wsadmin <i>startApplication</i> command.
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.	WebSphere rapid deployment offers the following advantages: <ul style="list-style-type: none">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, 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.

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 name

Specifies the name of the installed (or deployed) application that you selected on the Enterprise Applications page.

Full application

Under **Update options**, specifies to replace the application already installed on the server with a new (updated) enterprise application .ear file.

After selecting this option, 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. Only .ear, .jar, 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 on a Windows machine might be C:\WebSphere\AppServer\installableApps\test.ear.

Note: 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, 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.

After specifying the required information on the .ear file, 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, 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.

Single module

Under **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), 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 **Full application** above.

To replace a module, the value for **Relative path to module** (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 value for **Relative path to module** 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 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`.

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.

Single file

Under **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, .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 **Full application** option. To update a .war file, .rar file, or .jar file that is defined as a module in the application, use the **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 **Full application** above.

Next, specify a value for **Relative path to file**. The relative path of the file must start 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 value for **Relative path to file** must match the path of the file to be updated in the installed application.

To add a new file to the installed application, the value for **Relative path to file** must *not* match the path of a file in the installed application. The value specifies the desired path for the new file.

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:

- For files added or updated at 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.
- For files 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.

Partial application

Under **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, .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	<p>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.</p> <p>For example, to delete a file named utils/config.xml from the root of the my.ear file, include the line utils/config.xml in the META-INF/ibm-partialapp-delete.props file.</p>
Module	<p>Include <code>module_uri</code>/META-INF/ibm-partialapp-delete.props in the compressed file.</p> <p>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.</p> <p>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 <code>.*jsp</code> in the my.war/META-INF/ibm-partialapp-delete.props file. The line uses a regular expression, <code>.*jsp</code>, to identify all .jsp files in my.war.</p>

You can use a single partial application file to add, delete and update multiple files.

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.xml` 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.xml` 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 article assumes that your application files are deployed on a server and you want to upgrade the files.

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 1070, 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, *install_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. **Optional:** Examine the values specified for **Enable class reloading** and **Reloading interval** on the settings page for your enterprise application. If reloading of application files is enabled and the reload interval is greater than zero (0), the application files are reloaded after the application is updated. For Web modules such as servlets and JavaServer page (JSP) files, a Web container reloads a Web module only when the `IBM extension reloadingEnabled` in the `ibm-web-ext.xmi` file is also set to `true`. You can set `reloadingEnabled` to `true` when editing your Web module’s extended deployment descriptors in an assembly tool.
4. Change or add the following components or modules as needed:
 - Application files
 - WAR files

- EJB Jar files
 - HTTP plug-in configuration files
5. For changes to take effect, you might need to start, stop, or restart an application. “Starting and stopping applications” on page 1067 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.

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. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server. This article 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 wsadmin `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. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server. This article 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. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server. This article 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.xml` or `ibm-ejb-jar-bnd.xml` 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.xml` or `ibm-ejb-jar-bnd.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

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. 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 for information on how to set this property.) You can also run the `GenPluginCfg.bat/sh` script, or issue a `wasadmin` 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 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 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.

In the single-server product, application binaries are deleted after you save the changes.

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 from an application page is displayed
 4. Select the URI of the file that you want removed from the application.
 5. Select **Export before removing file** to back up the application.
 6. Specify the location to which you want the file exported.
 7. Click **Back** to return to the Enterprise Applications page.
- Remove a file from a module.
 1. Go to the settings page for the application. Click **Applications > Enterprise Applications > application_name** in the console navigation tree.
 2. Under **Related Items**, click **Web modules**, **EJB Modules**, or **Connector Modules**.
 3. Select the module from which you want to delete a file.
 4. Click **Remove File**. The Remove a file from a module page is displayed.
 5. Select the URI of the file that you want removed from the module.
 6. Optional: Back up the application. Select the application name and then specify the location to which you want the file exported.
 7. 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.

Save the changes to your administrative configuration. In the single-server product, application binaries are deleted after you save the changes.

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 the information center 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
- Programming instructions and examples

Programming model and decisions

- The J2EE™ Tutorial: The Duke's Bookstore Application
- Best Practices in WebSphere Application: Separating the developers from the administrators
- Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, Second Edition
- Designing Enterprise Applications, Second Edition

- Building Java™ Enterprise Applications Volume I: Architecture

Programming instructions and examples

- WebSphere Application Server education
- Developing and Testing a Complete 'Hello World' J2EE Application with IBM WebSphere Studio Application Developer for Linux
- Writing Enterprise Applications with Java™ 2 Platform, Enterprise Edition

Administration

- Listing of all IBM WebSphere Application Server Redbooks

Chapter 9. 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 and databases

- If you did not solve the problem, prepare to contact IBM support.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see "Obtaining help from IBM" in the information center.

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.

Errors or problems deploying, installing, or promoting applications

This article 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.
- I get a `java.lang.RuntimeException: Failed_saving_bytes_to_wor_ERROR_` in the assembly tool, administrative console or wsadmin tool
- I get a WASX7015E error running the wsadmin command `$AdminApp installInteractive` or `$AdminApp install..`
- A data definition language (DDL) generated by an assembly tool throws an SQL error on the target platform.
- The error ADMA0004E: Validation error in task Specifying the Default Datasource for EJB Modules occurs when installing application in administrative console or the wsadmin tool.
- The error No valid target is specified in ObjectName *object* for module *module* occurs from installation.
- The `addNode -includeapps` option does not appear to upload all applications to the deployment manager.
- "Timeout!!!" error displays when attempting to install an enterprise application in the administrative console.
- I get a `NameNotFoundException` message when deploying an application that contains an EJB module
- During application installation, the call to EJB deploy throws an exception
- I get compilation errors and EJB deploy fails when installing an EJB JAR file generated for Version 5.x or earlier
- While uploading documents, `addNode -includeapps` fails with an `OutOfMemoryError` exception

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 "Obtaining help from IBM" in the information center.

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 *install_dir*/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 \$AdminConfigsave command in the wsadmin tool before exiting.

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 EJBDploy 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 UNIX-style "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 IDs 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 Windows operating system on a UNIX platform, 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 UNIX platforms 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 Version 5, 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, either:

- 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" in the information center.

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 EJB deploy 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:
-Xmxsize
- 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 "Java virtual machine settings" in the information center 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" in the information center.
- "The application does not start or starts with errors" on page 1098.
- "A web resource does not display" on page 1099.
- "Cannot access a data source" in the information center.
- "Cannot access an enterprise bean from a servlet, a JSP file, a stand-alone program, or another client" in the information center.
- "Cannot look up an object hosted by WebSphere Application Server from a servlet, JSP file, or other client" in the information center.
- "Access problems after enabling security" in the information center.
- "Errors after enabling security" in the information center.
- "Errors after configuring or enabling Secure Sockets Layer" in the information center.
- "Errors in messaging" in the information center.
- "Errors returned to a client sending a SOAP request" in the information center.
- "A client program does not work" in the information center.
- "Errors connecting to WebSphere MQ and creating WebSphere MQ queue connection factory" in the information center.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see "Obtaining help from IBM" in the information center.

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.

Errors 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
- Graphics do not appear on the JavaServer Pages (JSP) file or servlet output
- SRVE0026E: [Servlet Error]-[Unable to compile class for JSP error on JSP]
- After modifying and saving a JSP file, the change does not show up in the browser (the old JSP file displays)
- Message similar to "Message: /jspname.jsp(9,0) Include: Mandatory attribute page missing" displays when trying to access JSP file
- The Java source generated from a JSP file is not retained in the temp directory (only the class file is found)
- The JSP batch compiler fails with the message "Enterprise Application [application name you typed in] not found"
- There is a translation problem with non-English browser input
- Scroll bars do not appear around items in the browser window
- A "Page cannot be displayed... server not found or DNS error" error displays when attempting to browse a JavaServer Pages (JSP) file using Internet Explorer

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 *install_dir/config/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. You might need 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 and 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 *install_dir/config/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.
5. For further details on troubleshooting plug-in-related problems, see the topic "Troubleshooting the HTTP plug-in component" in the information center.

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 *installation_root/installedApps/nodename/appname.ear* directory or *installation_root/installedApps/nodename/appnameNetwork.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.xml* configuration file, typically found in the *install_root/config/cells/nodename* or *nodenameNetwork/applications/application name/deployments/application name/Webmodule 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 *installation_root/installedApps/nodename/appname.ear* directory or *installation_root/installedApps/nodename/appnameNetwork.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: ``, not ``.

- 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.xml* configuration file, typically found in the *install_root/config/cells/nodename* or *nodenameNetwork/applications/application name/deployments/application name/Webmodule name/web-inf* directory.
 3. After following the previous steps:
 - 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 completing the following steps:

- 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. Turn Reloading on, or if it is already on, 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 JSPdirective, 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 *install_root*\config\cells*node_name*Network\applications directory is expected as an argument to the JspBatchCompiler tool, not just the display name. 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() 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 > DeploymentManager > Administration Services > Custom Properties**
2. Select ConnectionIOTimeout
3. Increase the ConnectionIOTimeout value
4. Click OK.

The application does not start or starts with errors

What kind of error do you see when you start an application?

- A `java.lang.ClassNotFoundException: classname Bean_AdderServiceHome_04f0e027Bean` error occurs
- A `ConnectionFactory E J2CA0102E: Invalid EJB component: Cannot use an EJB module with version 1.1 using The Relational Resource Adapter` error occurs
- `NMSV0605E: A Reference object looked up from the context...` error when starting an application.
- Other name server ("NMSV...") errors.

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 "Obtaining help from IBM" in the information center.

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 v6.0**. 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.
 4. Use the administrative console to install the deployed version created by the assembly tool.
- If you are using 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 Assembly Toolkit fulfill the Enterprise JavaBeans 1.1 specification. To run on WebSphere Application Server V5, 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 UriContextHel 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.

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 "Obtaining help from IBM" in the information center.

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 "Obtaining help from IBM" in the information center

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`.

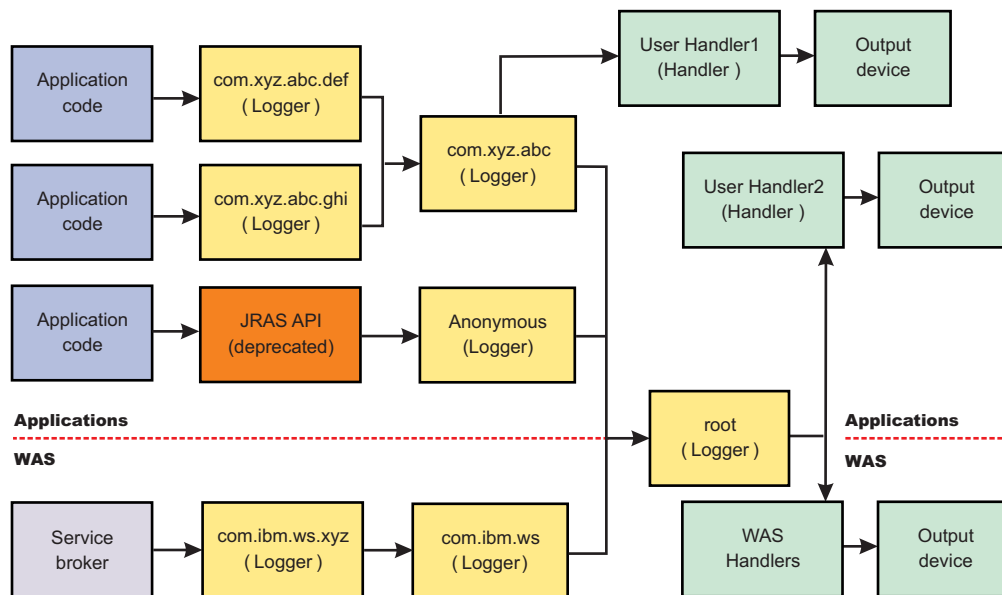
Chapter 10. Adding logging and tracing to your application

Deprecation: The JRes 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 simply adding `System.out.println` statements to your code:

- Your messages are displayed in the WebSphere Application Server standard log files, using 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 Analyzer tool.



Logging and tracing with Java logging

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 has 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*.

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.

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, meaning they are displayed in the national language of the end user. Although the destination and lifetime of messages might be configurable, some level of message logging should be enabled in normal system operation. Message logging should be used 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 the following topics:

- “Configuring logging properties using the administrative console” on page 1107.

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 has completed startup) or because a potential problem has been detected (such as a timeout waiting for a resource) a message will be 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 logged to it. If the resource bundle is not specified, a logger will use the same resource bundle as its parent.

Loggers can be configured with a level. If specified the level is compared by the logger to incoming events -- events less severe than the level set for the logger are ignored by the logger. If the level is not specified, a logger will take on the level used by its parent. The default level for loggers is `Level.INFO`.

Loggers can have zero or more attached handlers. If supplied, all events logged to the logger will be passed to the attached handlers (for more details see “Log handlers” on page 1105). Handlers write events to output destinations such as log files or network sockets. When a logger has finished passing a logged event to all of the handlers attached to that logger, the logger then passes the event to the handlers attached to the parents of the logger. This process will stop if a parent logger has been 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 up 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 logger "com.ibm.websphere.ras" has a parent logger named "com.ibm.websphere", which has a parent named "com.ibm", and so on. 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 logger "com".

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 log (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 ResourceBundle) methods add the capability of specifying a ResourceBundle 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 a complete list of methods, see the java.util.logging documentation at <http://java.sun.com/j2se/>.

Log handlers

Handlers write log record objects to an output device. Some examples of an output device are log files, sockets, and notification mechanisms.

Loggers can have zero or more attached handlers. All objects logged to the logger will be passed to the attached handlers, if handlers are supplied.

Handlers can be configured with a level. The handler compares the level specified in the logged object to the level 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 could decide to first include the timestamp, followed by a string representation of the level, followed by the message included in the logged object. The handler writes this formatted representation to the output device.

Both loggers and handlers can have levels and filters, and an logged object must pass all of these in order to be output. For example, you can set the logger level to FINE, but if the handler level is set at WARNING, then only WARNING level messages will appear 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, then 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
- 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 1107

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 (see “Configuring logging properties using the administrative console” on page 1107). 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 has been 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
Finest	Trace information - A more detailed trace - Includes all the detail needed to debug problems
All	All events are logged. If you have created custom levels, "All" would include your custom levels, and could 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 1107

Note: Trace information, which are events at levels Fine, Finer and Finest, can only be written 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.

Log filters

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. For a complete list of methods, see the `java.util.logging` documentation at <http://java.sun.com/j2se/>

Log formatters

This article describes what a log formatter is.

Handlers may be configured with a formatter, which knows how to format log records. The event (represented by the log record 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.

The formatter is responsible for rendering the event for output. This usually means the formatter uses the `ResourceBundle` specified in the event to look up the message in the appropriate language.

Formatters are attached to handlers using the `setFormatter` method.

WebSphere Application Server allows you to configure the formatter to be used with `trace`, `SystemOut.log`, and `SystemErr.log` log files:

- **Basic (Compatible)** - Preserves only basic trace information. The option allows you to minimize the amount of space taken up by the trace output.
- **Advanced** - Preserves more specific trace information. **Advanced** allows you to see detailed trace information for use in troubleshooting and problem determination.
- **Log Analyzer** - Preserves trace information in a format that is compatible with the Log Analyzer tool, so that you can use the trace output as input to the Log Analyzer 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/>.

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, and specify a format for log output. You can also 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 propagate the change to their children, as necessary.

To configure loggers and log handlers for use by 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. See “Log level settings” for a description of each 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 page 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

Specifies tracing details.

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 below. You can enter the log detail level string directly, or generate it using the graphical trace interface.

If you select the configuration tab, a 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 groups is displayed will all such components currently registered on the running server. This list is static; you will not see your application logger names on the runtime tab.

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 * as a wildcard to indicate components that include all classes in all packages contained by the specified component. For example:

- * Specifies all traceable code running in the application server, including WebSphere Application Server system code and customer code.

com.ibm.ws.*

specifies all classes whose package name begins with com.ibm.ws.

com.ibm.ws.classloader.JarClassLoader

Specifies only the JarClassLoader class.

Note: 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 desired 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 below lists the valid levels for application servers at WebSphere Application Server Version 6 and higher, and the valid logging and trace levels for earlier versions:

Version 6 Logging Level	Logging Level pre-Version 6	Trace Level pre-Version 6	Content / Significance
Off	Off	All disabled*	Logging is turned off. * In Version 6, a trace level of All disabled will turn off trace, but will not turn off logging. Logging will be enabled from the Info level.
Fatal	Fatal	-	Task cannot continue and component/ application/ server cannot function.
Severe	Error	-	Task cannot continue but component/ application/ 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 / 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" would include those levels, and could provide a more detailed trace than "finest".
-----	--	-------------	--

When you enable a logging level in version 6, you are also enabling all of the levels above it. For example, if you set the logging level to warning on your version 6 application server, then warning, severe and fatal events will be processed.

Note: Trace information, which are events at levels Fine, Finer and Finest, can only be written 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

To view this administrative console page, click **Application servers** > > *server name* > **HTTP error and NCSA access logging** .

Use this panel to configure an HTTP error log and NCSA access logs for an HTTP transport channel.

The HTTP error log logs HTTP errors. The level of error logging that occurs is dependent on the value selected for the Error log level field.

The NCSA access log contains a record of all inbound client requests that the HTTP transport channel handles. All of the messages contained in these logs are in NCSA format.

After you have configured the HTTP error log and the NCSA access logs, make sure the Enable NCSA access logging field is selected for the HTTP channels for which you want logging to take place. 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 will be initialized when the server is started.

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: Specifies 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: Specifies the maximum size, in megabytes, of the NCSA access log file. When this size is reached, an archive log named *logfile_name.1* is created. However, every subsequent time that the original log file overflows this archive file is overwritten with the most current version of the original log file.

NCSA access log format: Specifies 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: Specifies the directory path and 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: Specifies the maximum size, in megabytes, of the HTTP error log file. When this size is reached, an archive log named *logfile_name.1* is created. However, every subsequent time that the original log file overflows this archive file is overwritten with the most current version of the original log file.

Error log level:

Specifies 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 Errors in responses to clients are logged. These errors require Application Server administrator intervention if they are caused by 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 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.

Configuring logging properties for an application

The `logger.properties` file allows you to set logger attributes for specific loggers. The properties file is loaded the first time the method `Logger.getLogger(loggername)` is called within an application.

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 JAR file for the application
- directories included in the class path of 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 - minimum `LogRecord` level for which localization will be attempted
 - group - logical group that this component belongs to
 - eventfactory - The Common Base Event template file to use with the event factory. Note that 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 package `com.ibm.compXYZ` would be called `com.ibm.compXYZ.event.xml`
- Logger data information
 - product name
 - organization name
 - component name
 - extensions – additional properties

Sample logger.properties file

In the following sample, event factory com.ibm.xyz.MyEventFactory will be used by any loggers in the com.ibm.websphere.abc package or any sub-packages which do not override this value in their own configuration file.

```
com.ibm.websphere.abc.eventfactory=com.ibm.xyz.MyEventFactory
```

Sample security

Purpose

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 to allow the application to programmatically alter controlled properties of loggers and handlers.

```
////////////////////////////////////  
//  
// 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 article describes how to use Java logging within an application.

To instrument 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 will be used by the application, create a resource bundle as described in “Creating log resource bundles and message files” on page 1117.
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

There are various ways in which to log messages or trace using Java logging. The following guidelines will help you to use Java logging to log messages and add tracing:

1. Use level `WsLevel` 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`

2. Use the `logp` method instead of `log` or `logrb` since `logp` 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.
3. Avoid using the `logrb` method since this leads to inefficient caching of resource bundles which results in poor performance.
4. Use the `isLoggable` method to avoid creating data for a logging call that will 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
// - can't specify message substitution parameters
// - can't specify class/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 - not generally recommended due to lack of class / method
// names
// - enable use of WebSphere Application Server specific levels
// - enable use of message substitution parameters
// - can't specify class/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 - the recommended way to log
// - enable use of WebSphere Application Server specific levels
// - enable use of message substitution parameters
// - enable use of class/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 - not generally recommended due to diminished performance
of switching resource bundles on the fly
// - enable use of WebSphere Application Server specific levels
// - enable use of message substitution parameters
// - enable use of class/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 - recommended 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 - not generally recommended due to lack of message - use
logp with a throwable parameter instead
if (logger.isLoggable(Level.FINER))
    logger.throwing(className, methodName, throwable);

// "Convenience" methods - not generally recommended due to lack of class
/ method names
// - can't specify message substitution parameters
// - can't specify class/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 - not generally recommended due to lack of class /
method names
// - enable use of WebSphere Application Server specific levels
// - enable use of message substitution parameters
// - can't specify class/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 - the recommended way to log
// - enable use of WebSphere Application Server specific levels
// - enable use of message substitution parameters
// - enable use of class/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 - not applicable for trace logging since no localization
is involved

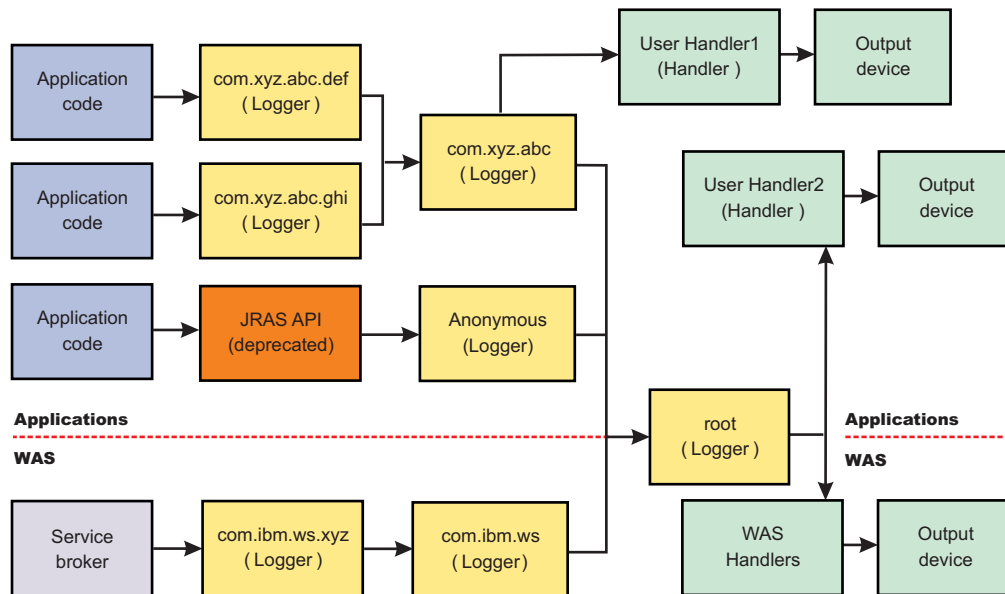
```

Understanding the logger hierarchy

WebSphere Application Server handlers are attached to the Java root logger in the logger hierarchy. As a result, any request from anywhere in the logger tree can be processed by WebSphere Application Server handlers. WebSphere Application Server handlers will process both standard log records and WebSphere Application Server log records. You can configure the system to:

- Forward all application logging requests to the WebSphere Application Server handlers. This is the default behavior.
- Forward all application logging requests to your own custom handlers. To do this, set **useParentHandlers** 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. To do this, set **useParentHandlers** to true on one of your non-root custom loggers (true is the default setting) , and then attach your handlers to that logger.
- 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 example below shows how these requirements can be met using the Java logging infrastructure.



Creating log resource bundles and message files

Every method that accepts messages will localize those messages. The mechanism for providing localized messages is the Resource Bundle support provided by the IBM Developer Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer's Kit, you can get more information from various texts, or by reading the Javadoc for the `java.util.ResourceBundle`, `java.util.ListResourceBundle` and `java.util.PropertyResourceBundle` classes, as well as the `java.text.MessageFormat` class.

The `PropertyResourceBundle` 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 displayed on the administrator 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. This means that 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 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 are logged. 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's 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 only white space, 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 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 doing this is not recommended, because escaping characters is error prone and confusing. It is instead recommended that you use a valid separator character that does not appear 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 construction of properties files.

2. The file can then be translated 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's classpath.
4. When a message logger is obtained from the log manager, it can be configured to use a particular resource bundle. Messages logged via the `Logger()` API will 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 specified by a message key, thus 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's location relative to any directory in the classpath. For instance, if the property resource bundle named `DefaultMessages.properties` 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.

Developing log resource bundles: **Resource bundle sample**

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a `PropertiesResourceBundle`. This sample shows how to create such a properties file.

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs 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 imposed on the values by the Java `MessageFormat` class. For example, apostrophes must be "escaped" or they will cause a problem. Also avoid use of non-portable characters. WebSphere Application Server does not support usage 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 will be in the classpath. Assume that the properties file is stored in a subdirectory of `baseDir` that is not in the classpath (e.g. `baseDir/subDir1/subDir2/resources`). In order to allow the messages file to be resolved, the name `subDir1.subDir2.resources.DefaultMessage` is used to identify the `PropertyResourceBundle` and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`.

```

# Contents of 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}

```

Once the file `DefaultMessages.properties` is created, the file can be sent to a translation center where the localized versions will be generated.

Creating a custom log handler

There may be occasions when you only 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 provided by the IBM Developer Java Technology Edition. If you are not familiar with handlers as implemented by the Developer's Kit, you can get more information from various texts, or by reading the Javadoc for `java.util.logging`.

The following is a sample of 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-Javadoc)
     * @see java.util.logging.Handler#publish(java.util.logging.LogRecord)
     */
    public void publish(LogRecord record) {
        // ensure that this LogRecord should be logged by this Handler
        if (!isLoggable(record))
            return;

        // Output the formatted data to the file
        printWriter.println(getFormatter().format(record));
    }

    /* (non-Javadoc)

```

```

    * @see java.util.logging.Handler#flush()
    */
    public void flush() {
        printWriter.flush();
    }

    /* (non-Javadoc)
    * @see java.util.logging.Handler#close()
    */
    public void close() throws SecurityException {
        printWriter.close();
    }
}

```

Creating a custom filter

A Filter provides optional, secondary control over what is logged, beyond the control provided by the level.

The mechanism for creating a customer filter is the Filter interface support provided by the IBM Developer Java Technology Edition. If you are not familiar with filters as implemented by the Developer's Kit, you can get more information from various texts, or by reading the Javadoc for `java.util.logging`.

The following is an example of a custom filter:

```

import java.util.Vector;
import java.util.logging.Filter;
import java.util.logging.LogRecord;

/**
 * MyCustomFilter rejects any LogRecords 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-Javadoc)
    * @see java.util.logging.Filter#isLoggable(java.util.logging.LogRecord)
    */
    public boolean isLoggable(LogRecord record) {
        return (acceptableLevels.contains(record.getLevel()));
    }

}

```

Creating a custom formatter

A formatter formats events. Handlers are associated with one or more formatters.

The mechanism for creating a customer formatter is the Formatter class support provided by the IBM Developer Java Technology Edition. If you are not familiar with formatters as implemented by the Developer's Kit, you can get more information from various texts, or by reading the Javadoc for `java.util.logging`.

The following is an example of 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();
    }
}

```

Using custom handlers, filters, and formatters

In some cases you may wish 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 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 “Understanding the logger hierarchy” on page 1116). 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 which we wish 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 executed, 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 has been 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

```

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 should be recorded for later analysis or which may require additional work to be triggered. An *event* is a structured notification that reports information related to a situation. An event reports three kinds of information:

- The situation itself (what has happened)
- The identity of the affected component (for example, the server that has shut down)
- The identity of the component that is reporting the situation (which might be the same as the affected component)

The application creating 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 a XML document 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 defining a common representation of events that is intended for use by enterprise management and business applications. This standard, developed by the IBM Autonomic Computing Architecture Board, supports encoding of logging, tracing, management, and

business events using a common XML-based format, making 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 should contain all of the information needed by the consumers to understand the event. This includes information 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 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

This topic describes types of problem determination events.

Problem determination involves using 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 may 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 operations. 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.

Common Base Events are primarily used to represent log events.

The structure of the Common Base Event

A Common Base Event contains several structural elements. These are:

- 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 fields in the Common Base Event and their usage requirements for problem determination events. It 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

Note:

1. 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.
2. Some of the elements are optional, but when included, they include sub-elements and attributes that are required. For example, reporterComponentId is of type ComponentIdentification. The component attribute in ComponentIdentification is required. Therefore, the reporterComponentId.component attribute is required, but only when the parent element (reporterComponentId) is included.
3. 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.
4. The contextDataElements element can be included multiple times to supply context data information.
5. The associatedEvents element can be included multiple times to supply correlation data. There are no recommended uses of this element for the producers of problem determination data, and it is in fact discouraged.

Common Header Information

The common header information in the Common Base Event includes the following information about an event:

- the version of this Common Base Event (version).
- the date and time when the event was generated (creationTime).
- the severity of the condition (situation) identified by the event (severity and priority).
- the type of event that was captured (extensionName).
- identifiers that can be used to quickly identify a specific event within a set of events (localInstanceId and globalInstanceId).
- information that allows a system to efficiently report multiple events of the same type, by consolidating those events into a single event (repeatCount and elapsedTime).
- sequence information that allows a system to order a set of events in other ways than time of capture (sequenceNumber).

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 article will provide additional information about how to format and use these fields for problem determination events, which should be used to clarify and extend the information provided in the other documents.

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 is more restrictive than the base specification for the Common Base Event, which lists this as an optional field because effective and efficient problem determination requires the ability to quickly identify the information needed to resolve a problem as well as prioritize the problems that need to be addressed. Typically the following values are used for problem determination events:

10	Information	Log information events (normal conditions, events 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 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 was able to correct. These events may require administrator action or intervention.
40	Minor	Minor errors describe events that represent an unrecoverable error within a component. The failure affects the component's ability to service some requests. The business application is able to continue to perform its normal functions, but its overall operation may 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's ability to service most requests. The business application is able to continue most (but not all) of its normal functions and its overall operation may 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 may be able to continue some normal functions, but its overall operation may be degraded. These events require administrator action or intervention to address the condition.
----	-------	---

msg

Refer to “Message Data” on page 1130 for information on this attribute.

priority

The usage 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. When the priority field is used, it should only be used to enhance the information provided in severity field, i.e. prioritize events of the same severity.

extensionName

The extensionName field is used to communicate the 'type' of event being reported, for example, what general class of events is being reported. In many cases this field provides an indication of what additional data should be expected to be supplied 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/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/consolidation by event management and analysis systems.

sequenceNumber

The sequenceNumber field is valid for problem determination events. It is typically only used by event producers when the granularity of the event time stamp (the creationTime field) is not sufficient in ordering events. In other words, the sequenceNumber field is typically used to sequence events that have the same time stamp value.

Note: Event management and analysis systems may use the sequenceNumber field for a number of reasons, including providing alternative sequencing, not necessarily based on time stamp. The recommendations here are provided primarily for event producers.

Component Identification (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 described by the event (the sourceComponentId) and which component emitted the event (the reporterComponentId). Typically, these are the same component, in which case only the sourceComponentId is supplied. Some notes and scenarios on when these two elements in the Common Base Event should be used:

- the sourceComponentId is always used to identify the component experiencing the condition described by the event.
- the reporterComponentId is used to identify the component that actually produced and emitted the event. This element is typically only used within events 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 captured by a component into Common Base Event format. The event conversion function is considered an extension of the component and should NOT be identified separately.

The information 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 (i.e. subcomponent) of a component, e.g. 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, i.e. 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, i.e. the 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 (e.g. the operating system name for a software application, the application server name for a 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 will provide additional information about how to format and use some of these fields for problem determination events, which should be used to clarify and extend the information provided in the other documents.

Component

The component field in a problem determination event is used to identify the manageable asset 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#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 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, i.e. 'What part failed?'. Examples of typical subcomponents and their names are:

- Intel Pentium processor within a server system (“Intel Pentium IV Processor”)
- the EJB container within a web application server (“EJB 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 it is recommended that the convention shown above for naming a Java class or the combination of a Java class and method is followed. Subcomponent is a required field in the Common Base Event.

componentIdType

The componentIdType field is required by the base Common Base Event specification, but provides minimal value for problem determination events. The only recommendation regarding this field for problem determination events is to discourage the use of the “application” value. 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 it should be provided within problem determination events whenever it is available. The only reason this is not a required field for problem determination events is there are instances where the issuing component may 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 it should be provided within problem determination events whenever it is available.

The instanceId should always be provided when a software component is being identified and should identify the operational instance of the component (for example, which operation instance of an installed software image is actually associated with the event). This value should also be provided for hardware components, but not all hardware components support the concept of operational instances.

The format of the supplied value is defined by the component, but must be a value that can be used by an analysis system (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 EJB
- **serial number** for a hardware processor

processId

The processId field is listed as an optional value within the Common Base Event specification, but it should be provided for problem determination events whenever it is available and applicable. It should always be provided for software-generated events, and should identify the operating system process associated with the component identified in the event. The format of the thread ID should match the format of the operating system (or other execution environment, such as a Java Virtual Machine). This field is typically not applicable or used for events emitted by hardware (for example, firmware).

threadId

The threadId field is listed as an optional value within the Common Base Event specification, but it

should be provided for problem determination events whenever it is available and applicable. It should always be provided for software-generated events, and should identify the operating system thread that was active when the event was detected or issued. A notable exception to this recommendation is some operating systems or execution environments do not support threads. The format of the thread ID should match the format of the operating system (or other execution environment, such as a Java Virtual Machine). This field is typically not applicable or used for events emitted by hardware (for example, firmware).

executionEnvironment

The `executionEnvironment` field, when used, should identify the immediate execution environment 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 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 EJB.

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 being 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, along with the values and formats used to describe these situations. The Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

There are a few recommendations for situation information for problem determination events, such as:

- Whenever possible, use the situation categorizations and qualifiers 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 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 (i.e. an event with `severity=warning`) typically indicates a recoverable failure, but the situation settings can be interpreted as unrecoverable failures (e.g. `ConnectSituation, successDisposition=UNSUCCESSFUL`). The appropriate situation categorization should always be used and the severity setting will indicate the severity of the situation, i.e. 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 describing the specific event being reported within the `msg` field of the Common Base Event. The text associated with events representing actual messages or log entries is expected to be internationalized (in other words, translated and localized). We strongly recommend including 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 was created and how it should be interpreted, and is particularly invaluable when trying to interpret the event programmatically or when trying to interpret the message independent of the locale or language used to format the message text.

Note: Readers of this section of the document should be familiar with the concepts associated with creating internationalized messages (in other words, messages that are translated and localized). A

good source of education on these concepts is provided by the documentation 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 text (i.e. the value of the `msg` field) provided with an event:

- The locale of the supplied message text, which identifies how the locale-independent fields within the message were formatted, as well as the language of the message (`msgLocale`).
- A locale-independent identifier associated with the message that can be used to interpret the message independent of the message language, message locale, and how the message was formatted (`msgId` and `msgIdType`).
- Information on how a translated message was created, including:
 - The identifier used to retrieve the message template (`msgCatalogId`).
 - The name and type of message catalog used to retrieve the message template (`msgCatalog` and `msgCatalogType`).
 - Any locale-independent information that was 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 will provide additional information about how to format and use these fields for problem determination events, which should be used to clarify and extend the information provided in the other documents.

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 is more restrictive than the base specification for the Common Base Event, which lists this as an optional field; because effective and efficient problem determination requires the ability to quickly identify the condition being reported by the event. The format and usage of this message is component-specific, but the following general guidelines should be followed:

- The message text supplied with messages and log events is expected to be internationalized.
- The locale of the supplied message text should be provided using the `msgLocale` field in the `msgDataElement` element of the Common Base Event.
- Additional information regarding the format and construction of internationalized messages should be provided whenever possible, using the `msgDataElement` element of the Common Base Event.

msgLocale

The message locale should be provided 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 it should be provided within problem determination events whenever it is available. The only reason this is not a required field for problem determination events is there are instances where the locale information is not provided or available when formatting the Common Base Event.

msgId and msgIdType

Several companies include within internationalized message text a locale-independent identifier that can be used to interpret the condition being described by the message text, independent of the language 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 the location (and language) where it was issued, meaning it 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 msgld. These fields should not be supplied when the message text has not been 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 it should be provided whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events representing issued messages or log events. It is not a required field for problem determination events because not all problem determination events include translated message text, and there are cases where the value is not provided or available when formatting the Common Base Event. This field should not be supplied when the message text has not been 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 it should be provided whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events representing issued messages or log events. It is not a required field for problem determination events because not all problem determination events include translated message text, and there are cases where the value is not provided or available when formatting the Common Base Event. This value contains the list of locale independent values (message tokens) inserted into the localized message text when creating a translated message. It is very difficult to extract these values from a translated message without having knowledge of the translated message template used to create the message, meaning having these values separate from the message text is invaluable for locale-independent and programmatic analysis of the data supplied in the message text. This field should not be supplied when the message text has not been translated or localized, e.g. for trace events. Note: 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. The msgCatalogTokens field should always be used to supply the list of message tokens included in the message text associated with an event. These values can also be supplied in other parts of the Common Base Event as well, 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 they should be provided whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events representing issued messages or log events. They are not required fields for problem determination events because not all problem determination events include translated message text, and there are cases where the values are not provided or available when formatting the Common Base Event. These fields should not be supplied when the message text has not been translated or localized, for example, for trace events.

Extended Data

The base information included in a Common Base Event may not be sufficient to represent all of the information captured by a component when creating a problem determination event. 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. We recommend using the latter approach, ExtendedDataElement elements.

An ExtendedDataElement element is used to represent a single data item, and 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 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

The XML document shown below is an example of a Common Base Event (CBE) instance generated by a WebSphere Application Server application.

```
<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>
```

Note that there are a number of extendedDataElement elements in the actual XML which are used by WebSphere Application Server, but which are not for use by applications as they may change without notice in future releases.

The CommonBaseEvent element defines the CBE instance. This element has a set of attributes that are common for all CBEs. This includes the extensionName attribute that defines the type or class of the CBE instance, the creation time, severity and priority.

Nested within the CommonBaseEvent element are elements giving more detail about the situation. The first of these is the situation element. This is a standardized classification of the situation.

The CommonBaseEvent element also includes the sourceComponentId and the (optional) reporterComponentId elements. The sourceComponentId describes where the situation occurred; the reporterComponentId describes where the situation was detected. If the sourceComponentId and reporterComponentId are the same, the reporterComponentId is omitted.

The attributes of both the sourceComponentId and the reporterComponentId elements are the same. They identify the component's type, name, operating system and network location. The content of these attributes provides vertical correlation of the stack of IT resources active when the when the CBE was created.

Also included in the CommonBaseEvent element are contextDataElements that describe the context in which the situation occurred. This context correlates CBE instances that were part of the same piece of

work. This is called horizontal correlation since an instance of a particular context type correlates events at the same level of abstraction (for example at the business level, or at the application level, or at the middleware level.)

Finally, there are the extended data elements. These contain additional data used to describe to situation. In this example, there is an extended data element added by WebSphere Application Server to describe the J2EE component that generated the CBE instance and some application data.

Sample Common Base Event template

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. Template information is used by the content handler to fill in blanks in the Common Base Event when the Common Base Event complete method is called. Information that is already supplied in the event will not be overridden if the same field is supplied in the template.

The following is an example of 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. The following definitions and examples should be used when creating Common Base Events for problem determination.

Business Application

A business application is the business logic and business data 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 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, developed either by the enterprise or a vendor, managed by the enterprise. A components may be created by the enterprise, typically for usage within a specific business application. For example, the ACME Corporation may create a set of enterprise beans to represent the business logic required by their Payroll Application. A component may also be an asset 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, may 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 may 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.

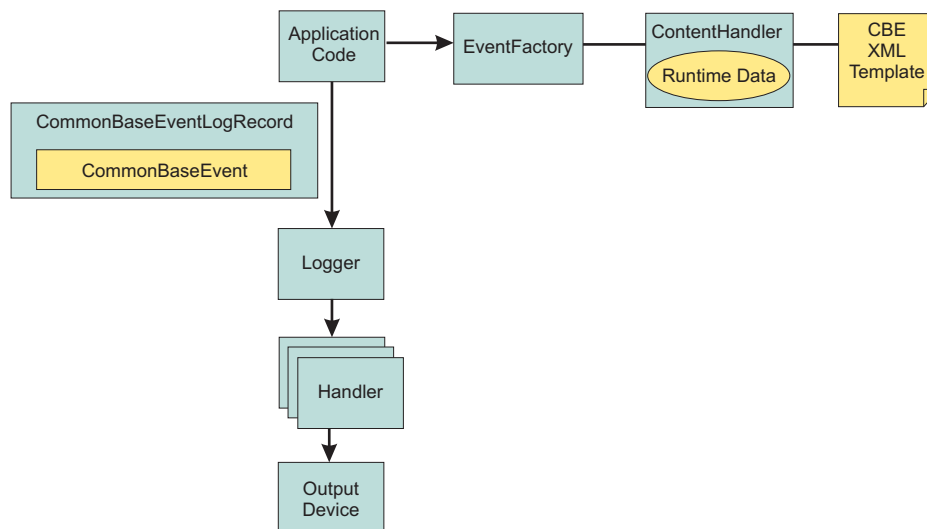
Logging Common Base Events in WebSphere Application Server

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 via 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.

Using the Common Base Event API with the Java Logging API to log Common Base Events

In cases where the events generated by the Java logging API are insufficient to describe the event that needs to be captured, Common Base Events can be created using 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 CommonBaseEvents:



The steps for generating a Common Base Event are as follows:

1. Application code invokes createCommonBaseEvent method on EventFactory to create a CommonBaseEvent.
2. Application code wraps CommonBaseEvent in a CommonBaseEventLogRecord, and adds event specific data.
3. Application code calls CommonBaseEvent's complete() method.

4. `CommonBaseEvent` invokes `ContentHandler`'s `completeEvent()` method.
5. `ContentHandler` adds XML template data to `CommonBaseEvent` (including for example, the component name). Note that not all `ContentHandlers` support templates.
6. `ContentHandler` adds runtime data to `CommonBaseEvent` (including for example, the current thread name).
7. Application code passes `CommonBaseEventLogRecord` to `Logger` using `Logger.log` method.
8. `Logger` passes `CommonBaseEventLogRecord` to `Handlers`.
9. `Handlers` format data and write to output device.

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 ensures consistency of Common Base Event content across events. However, other Common Base Event factories can be create and used. See “Configuring Common Base Events for an application” on page 1138 for details on how to create and use custom event factories.

Common Base Event factory context:

The event factory context provides a service to look up event factory homes. The event factory context can be retrieved using a call to `EventFactoryContext.getInstance()`. 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` must be located on the classpath to be found. The `EventFactoryContext` also stores an `EventFactoryHome` as a default, which can be obtained with a call to `EventFactoryContext.getInstance().getEventFactoryHome()`.

In WebSphere Application Server, the `EventFactoryContext` is configured with a default `EventFactoryHome` which is associated to a `ContentHandler` capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the javadoc for `org.eclipse.hyades.logging.events.cbe.EventFactory`.

Common Base Event factory home:

Event Factory homes provide Event Factory instantiation based on a unique factory name. Event Factory home implementations are tightly coupled with content handlers which 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 based on the type of the Event Factory home implementation.

In the WebSphere Application Server, the default Event Factory home obtained with a call to `EventFactoryContext.getInstance().getEventFactoryHome()` is associated with a `ContentHandler` capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the javadoc for `org.eclipse.hyades.logging.events.cbe.EventFactoryHome` at www.eclipse.org/hyades.

Common Base Event factory:

Event factories allow you to create Common Base Events and complete event properties using 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 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 only be retrieved from their associated event factory home. Event factory instances are retrieved and maintained based on unique names. Event factory names are hierarchal; they are represented using the standard Java dot-delimited name-space naming conventions.

More details can be found in the javadoc for `org.eclipse.hyades.logging.events.cbe.EventFactory` at www.eclipse.org/hyades.

Common Base Event content handler:

Content handlers populate data into Common Base Events when the Common Base Event `complete()` method is invoked. Content handlers can be associated with Common Base Event templates which provide default information to transfer into each Common Base Event. Content handlers may also provide any other information that is relevant to completing the population of the Common Base Event, such as runtime defaults deemed to be appropriate.

The use of content handlers is recommended to ensure consistency of field usage in the Common Base Event within a component or within a set of components sharing the same runtime. For example, some content handlers allow a template to be specified. If used consistently across a component, this would ensure that all events for that component would have the same template info filled in. Similarly, some content handlers can also supply runtime information to their associated Common Base Events. If consistently used throughout the entire runtime, this would ensure that all events use runtime data in a similar fashion.

The event factory home used in the WebSphere Application Server runtime is associated with a content handler that both reads from a template, and supplies runtime data. It is recommended that components use Event Factories obtained from this event factory home with their own templates, giving consistency between application events and server events.

More details can be found in the javadoc for `org.eclipse.hyades.logging.events.cbe.ContentHandler` at www.eclipse.org/hyades.

Using the Java Logging API to Generate and Log Common Base Events

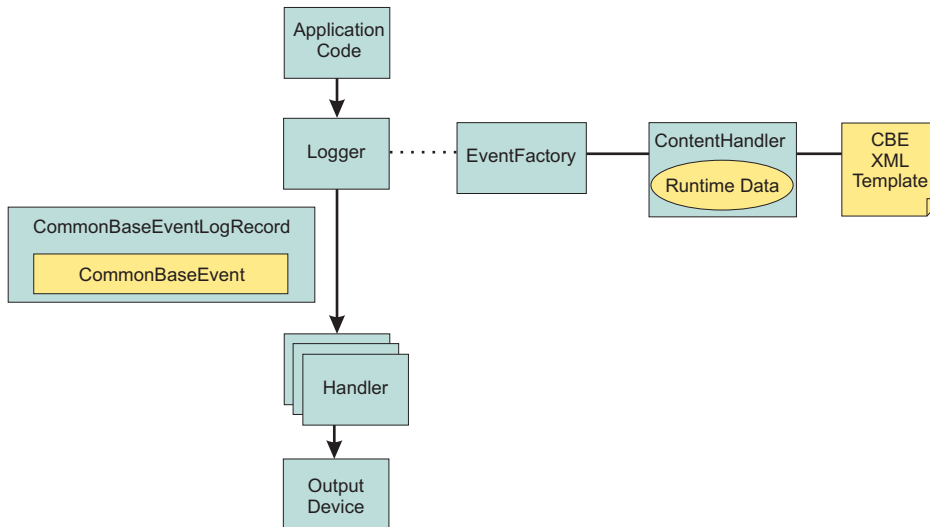
In the WebSphere Application Server, the Java logging API (`java.util.logging`) automatically creates Common Base Events for events logged at level `WsLevel.DETAIL` or above (including `WsLevel.DETAIL`, `Level.CONFIG`, `Level.INFO`, `WsLevel.AUDIT`, `Level.WARNING`, `Level.SEVERE`, and `WsLevel.FATAL`). These Common Base Events are created using the event factory associated with the Logger to which the message was logged. If no event factory is specified, WebSphere Application Server uses a default event factory which will automatically fill in WebSphere Application Server specific information.

The `java.util.logging.Logger` class provides a variety of methods with which data can be logged. The WebSphere Application Server uses a special implementation of the 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 `WsLevel.DETAIL` or more severe levels
- `logp` - when used with `WsLevel.DETAIL` or more severe levels
- `logrb` - when used with `WsLevel.DETAIL` or more severe levels

WebSphere Application Server logger implementation is only used for named loggers (for example, loggers instantiated with calls such as `Logger.getLogger("com.xyz.SomeLoggerName")`). Loggers instantiated with calls to `Logger.getAnonymousLogger()` and `Logger.getLogger("")`, or `Logger.global` do not use the WebSphere implementation, and do not automatically create Common Base Events for logging requests made to them. LogRecords logged directly with `Logger.log(LogRecord)` are not automatically converted by WebSphere's Loggers into Common Base Events.

The following diagram illustrates how application code can create log CommonBaseEvents:



The Java logging API processing of named Loggers and message level events proceeds as follows:

1. Application code invokes named Logger (WsLevel.DETAIL or above) with event specific data.
2. Logger creates a CommonBaseEvent using createCommonBaseEvent method on EventFactory associated to the Logger (see Configuring Common Base Events for an application).
3. Logger creates a CommonBaseEvent using EventFactory associated to the Logger
4. Logger wraps CommonBaseEvent in a CommonBaseEventLogRecord, and adds event specific data.
5. Logger calls CommonBaseEvent's complete() method.
6. CommonBaseEvent invokes ContentHandler's completeEvent() method.
7. ContentHandler adds XML template data to CommonBaseEvent (including for example, the component name). Note that not all ContentHandlers support templates.
8. ContentHandler adds runtime data to CommonBaseEvent (including for example, the current thread name).
9. Logger passes CommonBaseEventLogRecord to Handlers.
10. Handlers format data and write to Output Device.

Configuring Common Base Events for an application

The `logger.properties` file allows you to set logger attributes for your component. The properties file is loaded the first time the method `Logger.getLogger(loggername)` is called within an application. The `logger.properties` file must be either on the WebSphere Application Server class path, or the context class path. See "Configuring logging properties for an application" on page 1111 for details on the `logger.properties` file

To use the Common Base Events standard for events generated by your application, include the `eventfactory` property in your `logger.properties` file. The event factory property specifies the name of the Common Base Event template to use with the event factory. See "Sample Common Base Event template" on page 1134 for details on the Common Base Event template. Note that the naming convention for this is

the fully qualified component name with the extension “.event.xml”. For example, a template that applies to package com.ibm.compXYZ would be called “com.ibm.compXYZ.event.xml”

Sample logger.properties file

In the following sample, event factory com.ibm.xyz.MyEventFactory will be used by any loggers in the com.ibm.websphere.abc package or any sub-packages which do not have their own configuration file.

```
com.ibm.websphere.abc.eventfactory=com.ibm.xyz.MyEventFactory
```

Common Base Event content generated when using the WebSphere Application Server default event factory

When no other event factory is configured for a logger, the WebSphere Application Server uses its default event factory for creation of Common Base Events. The content handler associated with the default event factory populates fields as follows:

Field	Value	Notes
CommonBaseEvent.global-Instanceld	unique record id	only set if CommonBaseEvent.globalInstanceld was null before completeEvent() was called
CommonBaseEvent.msg	localized message based on MsgDataElement	only set if CommonBaseEvent.msg was null before completeEvent() was called
CommonBaseEvent.severity	set based on value of Level set on CommonBaseEventLogRecord if level >= Level.SEVERE set to 50 else if level >= Level.WARNING set to 30 default set to 10	only set if CommonBaseEvent.severity was null before completeEvent() was called
CommonBaseEvent.Component-Identification.component	set based on value of LoggerName set on CommonBaseEventLogRecord	only set if CommonBaseEvent.Component-Identification.component was null before completeEvent() was called
CommonBaseEvent.Component-Identification.componentIdType	"Unknown"	only set if CommonBaseEvent.Component-Identification.componentIdType was null before completeEvent() was called
CommonBaseEvent.Component-Identification.executionEnvironment	OSname[OSarch]#OSversion	only set if CommonBaseEvent.Component-Identification.executionEnvironment was null before completeEvent() was called
CommonBaseEvent.Component-Identification.instanceld	cellName\nodeName\serverName	only set if CommonBaseEvent.Component-Identification.instanceld was null before completeEvent() was called only set in a server environment (ignored in a client application)
CommonBaseEvent.Component-Identification.location	hostname	only set if both CommonBaseEvent.Component-Identification.location and CommonBaseEvent.Component-Identification.locationType were null before completeEvent() was called

Field	Value	Notes
CommonBaseEvent.Component-Identification.locationType	"Hostname"	only set if both CommonBaseEvent.Component-Identification.location and CommonBaseEvent.Component-Identification.locationType were null before completeEvent() was called
CommonBaseEvent.Component-Identification.processId	internally generated representation of process number	only set if CommonBaseEvent.Component-Identification.processId was null before completeEvent() was called
CommonBaseEvent.Component-Identification.subComponent	set based on values of sourceClassName and sourceMethodName set on CommonBaseEventLogRecord sourceClassName. sourceMethodName	CommonBaseEvent.Component-Identification.subComponent was null before completeEvent() was called and both sourceClassName and sourceMethodName were set
CommonBaseEvent.Component-Identification.threadId	set to value of JVM thread name	only set if CommonBaseEvent.Component-Identification.threadId was null before completeEvent() was called
CommonBaseEvent.Component-Identification.componentType	"http://www.ibm.com/namespaces /autonomic /WebSphereApplicationServer"	only set if CommonBaseEvent.Component-Identification.componentType was null before completeEvent() was called
CommonBaseEvent.MsgData-Element.msgLocale	set based on default locale of JVM	only set if CommonBaseEvent.msg was null before completeEvent() was called
CommonBaseEvent.Situation.categoryName	"ReportSituation"	only set if CommonBaseEvent.Situation was null before completeEvent() was called
CommonBaseEvent.Situation.situationType.type	"ReportSituation"	only set if CommonBaseEvent.Situation was null before completeEvent() was called
CommonBaseEvent.Situation.situationType.reasoningScope	"EXTERNAL"	only set if CommonBaseEvent.Situation was null before completeEvent() was called
CommonBaseEvent.Situation.situationType.reportCategory	"LOG"	only set if CommonBaseEvent.Situation was null before completeEvent() was called

Note that the sourceComponentIdentification is populated if no reporterComponentIdentification exists when completeEvent is invoked on the ContentHandler, otherwise the reporterComponentIdentification will be populated instead.

The same content handler that WebSphere Application Server uses for generating its events can be used in your applications. Event factory instances can be obtained as follows:

```
EventFactory eventFactory =
    EventFactoryContext.getInstance().getEventFactoryHome().getEventFactory(factoryName);
```

where factoryName is the name of the CommonBaseEvent template you wish to use with the factory.

The factoryName can also be specified as a configuration parameter for a logger. See "Configuring Common Base Events for an application" on page 1138 for more details.

Best Practices for Logging Common Base Events in WebSphere Application Server

The following practices will ensure consistent use of Common Base Events within your components and between your components and WebSphere Application Server components:

- Use a different Logger for each component. Sharing loggers across components gets in the way of being able to associate Loggers with component specific information
- Associate Loggers with event templates that specify source component identification. This 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 `CommonBaseEvents` until you are finished adding data to the `CommonBaseEvent` and are ready to log it. This ensures that any decisions made by the `ContentHandler` based on data already in the event will be made using the final data.

The following sample `logger.properties` file entry demonstrates how to associate Logger `com.ibm.componentX` with event factory `com.ibm.componentX`:

```
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>
```

Programming with the JRas framework

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server applications. The JRas extensions 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.

Understanding the JRas facility

Note: The JRas framework 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 has failed and give information as to why. This enables the administrator to take the proper corrective action. Those who develop or maintain applications might need to gather detailed information relating to the execution path of a running application in order 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 in order 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 intended to be viewed by end users, systems administrators and support personnel. The text of the message must be clear, concise and interpretable by an end user. Messages are typically localized, meaning they are displayed 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 to be used by service engineers or developers. As such a trace record may be considerably more complex, verbose and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries may 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 may be enabled as needed to gather diagnostic information.

WebSphere Application Server provides a message logging and diagnostic trace API that can be used by applications. 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 (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 required for integration into the WebSphere Application Server runtime or for usage in a 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 introduced by the stand-alone JRas logging toolkit, but simply provide the appropriate implementation classes. The conceptual structure introduced by the stand-alone JRas logging toolkit is described below. It is equally applicable to the JRas extensions.

JRas Concepts

The following is a basic overview of important concepts and constructs introduced by the stand-alone JRas logging toolkit. It is not meant to be an exhaustive overview of the capabilities of this logging toolkit, nor is it intended to be 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 javadoc 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. These are message loggers and trace loggers. The set of methods on message loggers and trace loggers are different, since they provide different functionality. Message loggers create only message records and trace loggers create only trace records. Both types of loggers contain masks that indicates which categories of events the logger should process and which it should ignore. Although every JRas logger is defined to contain both a message and trace mask, the message logger only uses the message mask and the trace logger only uses the trace mask. For example, by setting a message logger's message mask to the appropriate state, it can be configured to process only Error messages and ignore Informational and Warning messages. Changing the state of a message logger's trace mask 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 will compare the event type specified by the caller to its current mask value. If the specified type passes the mask check, the logger will create an event object to capture the information relating to the event that was passed to the logger method. This information may include information such as the names of the class and method which is logging the event, a message and parameters to log, among others. Once the logger has created the event object, it forwards the event to all handlers currently registered with the logger.

Methods that are used within the logging infrastructure itself should not make calls to the logger method. When an application uses an object that extends a thread class, implements the `hashCode()`, 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 will process. For example, a message logger may be configured to pass both warning and error events, but a handler attached to the message logger may be configured to only pass error events. 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 may 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

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

JRas extensions

The stand-alone JRas logging toolkit defines interfaces and provides a variety of concrete classes that implement these interfaces. Since the stand-alone JRas logging toolkit was developed as a general purpose toolkit, the implementation classes do not contain the configuration interfaces and methods necessary for use in the WebSphere Application Server product. In addition, many of the implementation classes are not written appropriately for use in a J2EE environment. To overcome these shortcomings, WebSphere Application Server provides the appropriate implementation classes that allow 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 defined and maintained by the WebSphere Application Server runtime. This is the default mode of operation and is equivalent to the WebSphere Application Server 4.0 mode of operation.

stand-alone

In this mode, message and trace records are written solely to stand-alone logs 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 there 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 has been defined that allow the JRas extensions to be used in either a stand-alone or combined mode of operations.

JRas extension classes

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

WebSphere Application Server provides a base set of implementation classes that collectively are 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. As previously noted, this collection of 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 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 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, described below. You cannot directly instantiate message and trace loggers. Obtaining loggers in any manner other than directly from the Manager is not allowed. Doing so is a direct violation of the programming model.

The message and trace loggers instances obtained from the WebSphere Application Server Manager class are subclasses of the `RASMessageLogger()` and `RASTraceLogger()` classes provided by the stand-alone JRas logging toolkit. The `RASMessageLogger()` and `RASTraceLogger()` classes define the set of methods that are directly available. Public methods introduced by the JRas extensions logger subclasses cannot be called directly by user code. Doing so 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 will ever be created within the lifetime of a process. The first call to the Manager with a particular name will result in the logger being created and configured by the Manager. The Manager will cache a reference to the instance, then return it to the caller. Subsequent calls to the Manager that specify the same name will result in a reference to the cached logger being returned. Separate namespaces are maintained for message and trace loggers. This means a single name can be used to 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 could be used to instrument an entire application. Or users may determine that having a logger per class is more desirable. Or the appropriate granularity may lie somewhere in between. Partitioning an application into logging domains is rightfully determined by the application writer.

The WebSphere Application Server logger classes obtained from the Manager are thread-safe. Although the loggers provided as part of the stand-alone JRas logging toolkit implement the serializable interface, in fact 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.

Please note that there is no provision for allowing users to provide their own logger subclasses for use 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 not available to user code. Every logger 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. Once you have removed it, you cannot add the WebSphere Application Server handler again to the logger from which it was 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 located in the `com.ibm.websphere.ras` package. All message and trace loggers must be obtained from this Manager. A reference to the Manager is obtained by calling the static `Manager.getManager()` method. Message loggers are obtained by calling the `createRASMessageLogger()` method on the Manager. 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 allows multiple, unrelated trace loggers to be registered 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.

For example, suppose component A consist of 10 classes. Suppose each class is configured to use a separate trace logger. Suppose all 10 trace loggers in the component are registered as members of the same group (for example `Component_A_Group`). You can then 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.

Extending the JRas framework

Since the JRas extensions classes do not provide the flexibility and behavior required for many scenarios, a variety of extension points have been defined. You are allowed to write your own implementation classes to obtain the required behavior.

Note: The JRas framework described in this task and its sub-tasks 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 to allow 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 `RASHandler` interface. All handlers must implement this interface. You can write your own handler classes that implement the `RASHandler` interface. You should directly create instances of user-defined handlers and add them to the loggers obtained from the Manager.

The stand-alone JRas logging toolkit provides several handler implementation classes. These handler classes are inappropriate for usage in the J2EE environment. You cannot directly use or subclass any of the Handler classes 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 `RASFormatter` interface. All formatters must implement this interface. You can write your own formatter classes that implement the `RASFormatter` interface. You can only add these classes to a user-defined handler. 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 usage of these formatter classes is not supported.

Message event types

The stand-alone JRas toolkit defines message event types in the `RASMessageEvent` interface. In addition, the WebSphere Application Server reserves a range of message event types for future use. The `RASMessageEvent` 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 retrieved from the Manager have their message masks set to *pass* or process all message event types defined in the `RASMessageEvent` interface. In order 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 has been obtained from the Manager. WebSphere Application Server does not provide any built-in systems management support for managing any 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 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. Once 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 defined in the `RASITraceEvent` interface.

Trace loggers retrieved from the Manager 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, the state of the trace mask for a logger can be changed at run-time using WebSphere Application Server systems management facilities.

In order 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 handlers 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. Once 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 will process 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.

Writing User Extensions:

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. Since 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, it is recommended that you locate user written extensions in a separate

package. It is your responsibility to restrict access to the user written extensions appropriately.

Writing a handler

User written handlers must implement the `RASHandler` interface. The `RASHandler` interface extends the `RASMaskChangeGenerator` interface, which extends the `RASObject` interface. A short discussion of the methods 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 javadoc.

RASObject interface

The `RASObject` 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 toolkit 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 methods `public Hashtable getConfig()` and `public void setConfig(Hashtable ht)` are used to get and set the configuration state. The JRas extensions do not support properties based configuration and it is recommended that these methods be implemented 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 methods `public String getName()` and `public void setName(String name)` 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 methods `public String getDescription()` and `public void setDescription(String desc)` 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 method `public String getGroup()` is provided for usage by the `RASManager`. Since the JRas extensions provide their own `Manager` class, this method is never called. It is recommended you implement this as a no-operation.

RASMaskChangeGenerator interface

The `RASMaskChangeGenerator` interface is the interface that defines the implementation methods for filtering of events based on a mask state. This means that 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 since the message logger never generates trace events. Handlers however can actively use both mask values. For example a single handler could handle both message and trace events.

- The methods `public long getMessageMask()` and `public void setMessageMask(long mask)` are used to get or set the value of the message mask. The methods `public long getTraceMask()` and `public void setTraceMask(long mask)` 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 `RASMaskChangeListener` interface.

- The methods `public void addMaskChangeListener(RASMaskChangeListener listener)` and `public void removeMaskChangeListener(RASMaskChangeListener listener)` are used to add or remove listeners to the handler. The method `public Enumeration getMaskChangeListeners()` returns an Enumeration over the list of currently registered listeners. The method `public void fireMaskChangedEvent(RASMaskChangeEvent mc)` 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 `RASMaskChangeListener` interface. The logger implementations maintain a "composite mask" in addition to

the logger's own mask. The logger's 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's own mask. For example, the message logger's composite mask is formed by *or'ing* the message masks of all handlers registered with that logger, then *and'ing* the result with the logger's own message mask.

This means that all handlers are required to properly implement these methods. In addition, when a user handler is instantiated, the logger it is to be added to should be registered with the handler using 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. This allows the logger to dynamically maintain the composite mask.

The `RASMaskChangedEvent` class is defined by the stand-alone JRas logging toolkit. Direct usage of that class by user code is allowed in this context.

In addition the `RASIMaskChangeGenerator` introduces the concept of caching the names of all message and trace event classes that the implementing object will process. The intent of these methods is to allow a management program such as a GUI 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, if desired.

- The methods `public void addMessageEventClass(String name)` and `public void removeMessageEventClass(String name)` can be called to add or remove a message event class name from the list. The method `public Enumeration getMessageEventClasses()` will return an enumeration over the list of message event class names. Similarly, the `public void addTraceEventClass(String name)` and `public void removeTraceEventClass(String name)` can be called to add or remove a trace event class name from the list. The method `public Enumeration getTraceEventClasses()` will return an enumeration over the list of trace event class names.

RASHandler interface

The `RASHandler` interface introduces the methods that are specific to the behavior of a handler.

The `RASHandler` 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. Since spawning of threads is not allowed in the WebSphere Application Server environment, it is expected that handlers will not queue or batch events, although this is not expressly prohibited.

- The methods `public int getMaximumQueueSize()` and `public void setMaximumQueueSize(int size)` throw `IllegalStateException` are provided to manage the maximum queue size. The method `public int getQueueSize()` is provided to query the actual queue size.
- The methods `public int getRetryInterval()` and `public void setRetryInterval(int interval)` support the notion of error retry, which again implies some type of queueing.
- The methods `public void addFormatter(RASFormatter formatter)`, `public void removeFormatter(RASFormatter formatter)` and `public Enumeration getFormatters()` 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 methods `public void openDevice()`, `public void closeDevice()` and `public void stop()` are provided to manage the underlying device that the handler abstracts.
- The methods `public void logEvent(RASIEvent event)` and `public void writeEvent(RASIEvent event)` are provided to actually pass events to the handler for processing.

Writing a formatter

User written formatters must implement the `RASFormatter` interface. The `RASFormatter` interface extends the `RASIObjct` interface. The implementation of the `RASIObjct` interface is the same for both handlers and formatters. A short discussion of the methods introduced by the `RASFormatter` interface follows. For

more in depth information on the methods introduced by this interface, see the corresponding product API documentation.

RASIFormatter interface

- The methods *public void setDefault(boolean flag)* and *public boolean isDefault()* are used by the concrete RASHandler classes provided by the stand-alone JRas logging toolkit to determine if a particular formatter is the default formatter. Since 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 methods *public void addEventClass(String name)*, *public void removeEventClass(String name)* and *public Enumeration getEventClasses()* are provided to determine which event classes a formatter can be used to format. You can provide the appropriate implementations as you see fit.
- The method *public String format(RASIEvent event)* is called by handler objects and returns a formatted String representation of the event.

Programming model summary

The programming model 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.

Note: The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

As described previously, 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 provided by the stand-alone JRas logging toolkit directly, unless specifically noted otherwise. Direct usage of those classes is not supported. IBM Support will provide no diagnostic aid or bug fixes relating to direct usage of classes 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.
- There is no provision that allows you to replace the WebSphere Application Server message and trace logger classes.
- You must guarantee that the logger names passed to the Manager are unique, and follow the naming constraints documented below. Once a logger is obtained from the Manager, 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 results in the Manager creating a logger that is associated with that name. Subsequent calls to the Manager that specify the same name result in a reference to the existing logger being returned.
- The Manager maintains a hierarchical namespace for loggers. It is recommended but not required that a dot-separated, fully qualified class name be used to identify any given logger. Other than dots or periods, logger names cannot contain any punctuation characters, such as asterisk (*), comma(.), equals sign(=), colon(:), or quotes.
- Group names must comply with the same naming restrictions as logger names.
- The loggers returned from the Manager are subclasses of the *RASMessageLogger* and *RASTraceLogger* provided by the stand-alone JRas logging toolkit. You are allowed to call any public method defined by the *RASMessageLogger* and *RASTraceLogger* classes. You are not allowed to call any public method 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 are not allowed to use the Handler and Formatter classes provided by the stand-alone JRas logging toolkit. User written handlers and formatters must conform to the documented guidelines.

- Loggers obtained from the manager come with a WebSphere Application Server handler installed. This handler will write message and trace records to logs defined by the WebSphere Application Server runtime. 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 allowed. You are responsible for creating an instance of the handler to add, configuring the handler by setting the handler's 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 may get a reference to the handler 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 obtained in this fashion. You are allowed to remove the WebSphere Application Server handler from the logger by calling the logger's `removeHandler()` method, passing in the reference to the WebSphere Application Server handler. Once removed, the WebSphere Application Server handler cannot be re-added to the logger.
- You are allowed to 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 are allowed to define your own message event classes. The usage of user-defined message event classes is discussed in Extending the JRas framework.
- You are allowed to 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 are allowed to define your own trace event classes. The usage 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 being used to manage the predefined types, these updates must not modify the state of any of the bits corresponding 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 section describes JRas message and trace event types.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Event types

The base message and trace event types defined by the stand-alone JRas logging toolkit are not the same as the "native" types recognized by the WebSphere Application Server run-time. Instead the basic JRas types are mapped onto the native types. This mapping may vary by platform or edition. The mapping is discussed below.

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 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 recognized and processed by the WebSphere Application Server runtime are defined in the `RASITraceEvent` interface provided by the stand-alone JRas logging toolkit. The `RASITraceEvent` interface provides a rich and overly 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, `RASITraceEvent` provides `TYPE_LEVEL1`, `TYPE_LEVEL2`, and `TYPE_LEVEL3`. The implementations provide support for this set of levels. The levels are hierarchical (that is, enabling level 2 will also enable 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, `RASITraceEvent` 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 JRas `RASITraceEvent` "Level" types.

WebSphere Application Server native type	JRas <code>RASITraceEvent</code> level type
Event	<code>TYPE_LEVEL1</code>
EntryExit	<code>TYPE_LEVEL2</code>
Debug	<code>TYPE_LEVEL3</code>

Mapping WebSphere Application Server trace types to JRas `RASITraceEvent` enumerated types.

WebSphere Application Server native type	JRas <code>RASITraceEvent</code> enumerated types
Event	<code>TYPE_ERROR_EXC</code> , <code>TYPE_SVC</code> , <code>TYPE_OBJ_CREATE</code> , <code>TYPE_OBJ_DELETE</code>
EntryExit	<code>TYPE_ENTRY_EXIT</code> , <code>TYPE_API</code> , <code>TYPE_CALLBACK</code> , <code>TYPE_PRIVATE</code> , <code>TYPE_PUBLIC</code> , <code>TYPE_STATIC</code>
Debug	<code>TYPE_MISC_DATA</code>

For simplicity, it is recommended that one or the other of the tracing type methodologies is used consistently throughout the application. For users who decide to use the non-level types, it is further recommended that you choose one type from each category and use those consistently throughout the application to avoid confusion.

Message and Trace parameters

The various message logging and trace method signatures accept parameter types of `Object`, `Object[]` and `Throwable`. WebSphere Application Server will process and format the various parameter types as follows.

Primitives

Primitives, such as `int` and `long` are not recognized as subclasses of `Object` and cannot be directly passed to one of these methods. A primitive value must be transformed to a proper `Object` type (`Integer`, `Long`) before being passed as a parameter.

Object `toString()` is called on the object and the resulting `String` is displayed. The `toString()` method should be implemented appropriately for any object 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[]` is provided for the case when more than one parameter is passed to a message logging or trace method. `toString()` is called on each `Object` in the array. Nested arrays are not handled. (i.e. none of the elements in the `Object` array should be an array).

Throwable

The stack trace of the Throwable is retrieved and displayed.

Array of Primitives

An array of primitive (e.g. byte[], int[] is recognized as an Object, but is treated somewhat as a second cousin of Object by Java code. In general, arrays of primitives should be avoided, if possible. If arrays of primitives are passed, the results are indeterminate and may 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 should 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 may result.

- [B@924586a0b - This is deciphered as "a byte array at location X". This is typically returned when an array is passed as a member of an Object[]. It is the result of calling toString() on the byte[].
- Illegal trace argument : array of long. This 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 may be seen when a byte array is passed to a method taking a single Object. This behavior is subject to change and should not be relied on.
- "1" "2" ... : The String representation of the members of an int[] formed by converting each element to an Integer and calling toString on the Integers. This behavior is subject to change and should not be relied on.
- [Ljava.lang.Object;@9136fa0b : An array of objects. Typically this 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 defined in the RASIMessageEvent interface.
2. Logging of that message event type must be enabled by the state of the message logger's mask.
3. The message event type must pass any filtering criteria established by the WebSphere Application Server run-time itself.

When a WebSphere Application Server logger is obtained from the Manager, the initial setting of the mask is to forward 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's mask.

Some editions of the product allow the user to specify a message filter level for a server process. When such a filter level is set, only messages at the specified severity levels are written to WebSphere Application Server logs. This means that messages types that pass the message logger's mask check may be filtered out by the WebSphere Application Server itself.

Controlling Tracing

Each edition of the product provides a mechanism for enabling or disabling trace. The various editions may 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 defined in the RASITraceEvent interface.
2. Logging of that trace event type must be enabled by the state of the trace logger's mask.
3. The trace event type must pass any filtering criteria established by the WebSphere Application Server run-time itself.

When a logger is obtained from the Manager, 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 has been specified. Unlike message loggers, the WebSphere Application Server may dynamically modify the state of a trace loggers trace mask. WebSphere Application Server will only modify the portion of the trace logger's mask corresponding to the values defined in the `RASITraceEvent` interface. WebSphere Application Server will not modify undefined bits of the mask that may be in use for user defined types.

When the dynamic trace enablement feature available on some platforms is used, the trace state change is reflected both in the Application Server run-time and the trace loggers trace mask. If user code programmatically changes the bits in the trace mask corresponding to the values defined by in the `RASITraceEvent` interface, the trace logger's mask state and the run-time state will become unsynchronized and unexpected results will occur. Therefore, programmatically changing the bits of the mask corresponding to the values defined in the `RASITraceEvent` interface is not allowed.

Instrumenting an application with JRas extensions

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

To instrument an application using the WebSphere Application Server JRas extensions, perform the following steps:

1. Determine the mode the extensions will be used in: integrated, stand-alone or combined.
2. If the extensions will be used in either stand-alone or combined mode, create the necessary handler and formatter classes.
3. If localized messages will be used by the application, create a resource bundle as described in [Creating JRas resource bundles and message files](#).
4. In the application code, get a reference to the Manager class and create the manager and logger instances as described in [Creating JRas manager and logger instances](#).
5. Insert the appropriate message and trace logging statements in the application as described in [Creating JRas manager and logger instances](#).

Creating JRas resource bundles and message files

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The WebSphere Application Server message logger provides the `message()` and `msg()` methods to allow the user to log localized messages. In addition, it provides the `textMessage()` method for logging of messages that are not localized. Applications can use either or both, as appropriate.

The mechanism for providing localized messages is the Resource Bundle support provided by the IBM Developer Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer's Kit, you can get more information from various texts, or by reading the Javadoc for the `java.util.ResourceBundle`, `java.util.ListResourceBundle` and `java.util.PropertyResourceBundle` classes, as well as the `java.text.MessageFormat` class.

The `PropertyResourceBundle` 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 displayed on the administrator 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. This means that 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's resource bundles with the application.

Late binding

The application can choose to have the WebSphere Application Server runtime localize the message in the process where it is displayed. 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 administrator's console or log viewing program might be run. You must install the `.jar` file in a directory that is part of the extensions classpath. In addition, if you forward logs to IBM service, you must also forward the `.jar` file containing 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 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 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 doing this is not recommended, because escaping characters is error prone and confusing. It is instead recommended that you use a valid separator character that does not appear 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 construction of properties files.
2. The file can then be translated 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, write them to a system-managed persistent storage medium. Resource bundles are then used to convert the messages into the requested national language and locale.
 4. When a message logger is obtained from the JRes manager, it can be configured to use a particular resource bundle. Messages logged via the `message()` API will 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 specified by a message key, thus 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's location relative to any directory in the classpath. For instance, if the property resource bundle named `DefaultMessages.properties` 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.

Developing JRas resource bundles:

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Resource bundle sample

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a `PropertiesResourceBundle`. This sample shows how to create such a properties file.

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs 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 imposed on the values by the Java `MessageFormat` class. For example, apostrophes must be "escaped" or they will cause a problem. Also avoid use of non-portable characters. WebSphere Application Server does not support usage 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 will be in the classpath. Assume that the properties file is stored in a subdirectory of `baseDir` that is not in the classpath (e.g. `baseDir/subDir1/subDir2/resources`). In order to allow the messages file to be resolved, the name `subDir1.subDir2.resources.DefaultMessage` is used to identify the `PropertyResourceBundle` and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`.

```
# Contents of 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}
```

Once the file `DefaultMessages.properties` is created, the file can be sent to a translation center where the localized versions will be generated.

Creating JRas manager and logger instances

You can use the JRas extensions in integrated, stand-alone, or combined mode. Configuration of the application will vary depending on the mode of operation, but usage of the loggers to log message or trace entries is identical in all modes of operation.

Note: 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. See [Setting up for integrated JRas operation](#) for information on configuring for this mode of operation.

In the combined mode, message and trace events are logged to both WebSphere Application Server and user-defined logs. See [Setting up for combined JRas operation](#) for more information on configuring for this mode of operation.

In the stand-alone mode, message and trace events are logged only to user-defined logs. See [Setting up for stand-alone JRas operation](#) for more information on configuring for this mode of operation.

Using the message and trace loggers

Regardless of the mode of operation, the use of message and trace loggers is the same. See [Creating JRas resource bundles and message files](#) for more information on using message and trace loggers.

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

Since trace is normally disabled, trace methods should be guarded 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 we waste cycles gathering parameters to be traced.
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT) {
        // since I want to trace 3 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. Here an important state change
    has been 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

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 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 and create the loggers. Since loggers are named singletons, you can do this in a variety of places. One logical candidate for enterprise beans is the `ejbCreate()` method. For example, for the enterprise bean named "myTestBean", 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 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

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 have written a user defined handler named `SimpleFileHandler` and a user defined formatter named `SimpleFormatter`. It also assumes that 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, create the loggers and add the user handlers. Since 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 handle this. 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.
```



```

RASHandler handler = new SimpleFileHandler("myHandler", "FileName");
RASFormatter 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(RASMessageEvent.DEFAULT_MESSAGE_MASK);

```

Setting up for stand-alone JRas operation

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`. It also assumes that no user-defined types or events are being used.

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, create the loggers and add the user handlers. Since 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 handle this. 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");

// Get a reference to the Handler and remove it from the logger.
RASHandler aHandler = null;
Enumeration enum = msgLogger.getHandlers();
while (enum.hasMoreElements()) {
    aHandler = (RASHandler)enum.nextElement();
    if (aHandler instanceof WsHandler)
        msgLogger.removeHandler(wsHandler);
}

// Create the user handler and formatter. Configure the formatter,
// then add it to the handler.
RASHandler handler = new SimpleFileHandler("myHandler", "FileName");
RASFormatter 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);
```

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 Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA

Trademarks and service marks

For trademark attribution, visit the IBM Terms of Use Web site (<http://www.ibm.com/legal/us/>).