IBM WebSphere Application Server, Version 5.1

Applications

Note

Before using this information, be sure to read the general information under “Notices” on page 673.

Compilation date: December 15, 2003

© Copyright International Business Machines Corporation 2003. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2003

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.

© Copyright IBM Corp. 2003

Chapter 1. Welcome to Applications

The following items comprise the application programming model, including numerous services available to support deployed applications.

Web modules

Use Web components such as servlets and JavaServer Pages files to develop dynamic Web sites. Product extensions to the open source servlet and JSP APIs enhance standard features, and provide additional functionality.

Web modules consist of the following application components, each performing a different function:
- HTML and JSP pages provide the user interface and program logic
- Servlets coordinate work between other components of the application

HTTP sessions are a key area of product support for Web modules. By **managing HTTP sessions** for your Web applications, you can personalize a Web site for individual customers. A session is a series of requests to a servlet, originating from the same user at the same browser. Managing HTTP sessions allows servlets running in a Web container to keep track of individual users. For example, a servlet might use sessions to provide ″shopping carts″ to on-line shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she will 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 choices of Shopper 1 to the cart of Shopper 2.

EJB modules

IBM WebSphere Application Server provides broad support for enterprise beans, including the Enterprise JavaBeans (EJB) 2.0 specification. The EJB 2.0 specification introduces a container-managed persistence (CMP) 2.0 component model, which provides a number of improvements to aid developer productivity and application performance. In addition, this product continues to fully support enterprise beans written to the CMP 1.1 programming model and deployed in previous versions of this product; applications can use CMP 1.1 beans, CMP 2.0 beans, or a mixture of both. CMP 1.1 beans can be directly carried forward in an EJB 1.1 ejb-jar module or may be repackaged and combined with CMP 2.0 beans in an EJB 2.0 module.

For EJB 2.0 modules, a feature introduced in Version 5 of this product, called **access intent** policies, eases the management of interactions between CMP beans and their underlying data stores. Each policy sets such data access characteristics such as access type (read or update) and transaction isolation that affect the locking of resources, letting you choose the level of data integrity and performance for your application.

Several excellent trade books that cover EJB 2.0 and the CMP 2.0 persistence model are already available. A good way to locate some of these is to visit your favorite online bookstore and search on the term *Enterprise JavaBeans*. For a more basic orientation, see “Enterprise beans: Resources for learning” on page 91.

Your application development might include **asynchronous messaging**, which the product supports as a method of communication based on the Java Message Service (JMS) programming interface.

The base JMS support enables IBM WebSphere Application Server applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). An application can explicitly poll for messages on a destination.

The product also provides a message listener service that applications can use to automatically retrieve messages from JMS destinations for processing by message-driven beans, without the application having to explicitly poll JMS destinations.

© Copyright IBM Corp. 2003

Client modules

For an overview, refer to Welcome to Client modules.

Web services

The Web services development and implementation components included in this product version are based on Apache SOAP 2.3. This information is deprecated in newer product versions.

The Web services development and implementation components included with this product version are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE), Java for XML-based remote procedure call (JAX-RPC) and WS-I Basic Profile 1.0 specifications.

An open source implementation for a Web Services Invocation Framework (WSIF) is also supported.

Additional features, such as UDDI Registry and Web Services Gateway are described in Welcome to Servers.

WebSphere Application Server supports Web services security functionality that is based on standards included in the Web services security (WS-Security) specification.

Application services

IBM WebSphere Application Server provides essential services to ease the building of dynamic and flexible e-business applications. These services support and extend the open standards of J2EE and Web services, with a focus on application reuse and integration.

- **Class loading**

  The WebSphere Application Server product provides several class-loading modes, policies, and features to enable you to deploy and run your applications successfully. An application server provides an Application class-loader policy that enables you to control the isolation of applications in a server. If you want applications to share classes, choose the SINGLE policy; otherwise choose the MULTIPLE policy, which isolates the class loaders for each application.

  Similarly, at the application level, you can choose a WAR class-loader policy that configures the isolation of Web modules within an application. If you choose the policy APPLICATION, then each Web module in your application can see the classes of other Web modules. A policy of MODULE creates a separate class loader for each Web module, resulting in isolation for each of the classes of each Web module.

  The class-loader mode setting, which you can configure at the server, application, or Web module level depending on your class-loader policy, enables you to control whether application class loaders override classes contained in base run-time class loaders. By default, the WebSphere Application Server class loaders have a class-loader mode of PARENT_FIRST, which is the standard JDK mode and does not allow the application class loader to override classes. You must take care when using the PARENT_LAST class-loader mode to make all dependent classes available within the application or you might get LinkageErrors or other class-loader exceptions. For example, if you provide a newer version of the `Xerces.jar` file and your application is using XSLT, you must also provide a `xalan.jar` file within your application.
- **Shared library**

  Version 5.0 of WebSphere Application Server introduces the concept of a shared library. A shared library is a CLASSPATH and a symbolic name for that class path. You define shared libraries at the cell, node, or server level and then associate the shared libraries either with an application server (making the classes available to all applications in the server) or with individual applications (making the classes available only to the referencing application). This mechanism provides a convenient way to make libraries of classes available to your applications outside of a standard J2EE enterprise application (EAR) file for easier version management and space efficiency.
- **Internationalization support**

If your application component must support multiple locales, the **localizable-text** API can help both developers and administrators through central management of displayed strings. The developer separates strings into a message catalog, which is then translated into the other languages required. All message catalogs are then deployed with the application component. From then on, the administrator can add or update message catalogs centrally as required. See Chapter 11, "Internationalizing applications," on page 527.

- **Transactions**

  IBM WebSphere Application Server applications can use transactions to coordinate multiple updates to resources as atomic units (as indivisible units of work) such that all or none of the updates are made permanent. 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 (where the bean manages transactions itself)
  - Entity beans use container-managed transactions
  - Web components (servlets) use bean-managed transactions

  The product is a transaction manager that supports the coordination of resource managers through their XAResource interface and participates in distributed global transactions with other OTS 1.2 compliant transaction managers (for example, J2EE 1.3 application servers). Applications can also be configured 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 IBM 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. With the Last Participant Support of Enterprise Extensions, you can coordinate the use of a single one-phase commit (1PC) capable resource with any number of two-phase commit (2PC) 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.

  The ActivitySession service of Enterprise Extensions 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 product EJB container and deployment tooling 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 application can then interact with a resource manager through its LocalTransaction interface for the period of a client-scoped ActivitySession rather than just the duration of an EJB method.
- **Naming**

  Naming clients use **Naming Services** primarily to access objects, such as EJB homes, associated with applications installed on IBM WebSphere Application Server. Objects are made available to clients by being bound into a name space. A name space is under the control of a name server. In this product, there are potentially many name servers, and the name spaces controlled by the various name servers are federated together to form the view of a single name space. Each name server presents the same logical view of the federated name spaces.

  Name servers provided by this product are a CORBA CosNaming implementation. IBM WebSphere Application Server provides a CosNaming JNDI plug-in which enables clients to access the name servers through the JNDI interface. Clients to EJB applications typically use JNDI to perform Naming operations. Clients may access the name servers directly through the CORBA programming model. The

CosNaming interface is part of the CORBA programming model. CORBA clients which need to access EJB homes or some other objects bound to the name space would typically use the CORBA CosNaming interface to perform Naming operations.

- **Dynamic cache**

  Dynamic cache improves application performance by caching outputs and contents of outputs of servlets, JavaServer Pages (JSP) files, Web services, and commands. On subsequent client requests to the same applications, dynamic cache intercepts these calls and responds by serving the output or the contents of output from the cache.

  Dynamic cache in this product version includes:

  Servlet/JSP files caching
  This caches output of dynamic servlets and JSP files by working with Java virtual machine of the application server by intercepting calls to service methods and serving Web pages from the cache. This improves server response time, throughput and scalability.

  Command caching
  Commands that are written to the Command Architecture encapsulate business logic tasks and provide a standard way to invoke the business logic request. Command objects need to implement CacheableCommand interface instead of TargetableCommand interface to cache. Like in servlets and JSP caching, requests to execute business logic in the command is intercepted by the cache. If a command with the same request attributes are available in cache, output properties are copied from the cached instance to the requested instance and returned without executing the business logic again.

  Web Services caching
  Web service responses can be cached just like servlet and JSP results. These requests are intercepted and cache ID computed based on how the cache ID rules are specified in the cache policy. Hash of the whole SOAPEnvelope can be used as a cache ID or it can be parsed and service name, operation name and parameter names to these operations used as cache ID. If a cache entry is not found for the computed cache ID, the request is forwarded to the SOAP engine and the result is cached.

  Edge Side Include caching
  This provides the ability to cache, assemble and deliver dynamic web pages at the edge of the enterprise network. Edge Side Includes (ESI) is a simple markup language which enables dynamic web pages (which by themselves are not so cache efficient) to be broken down into cacheable fragments. These fragments are then cached on the edge of the network and assembled into a single page upon user requests.

  Distributed caching
  Cache contents can be shared and replicated among servers by dynamic caching using an underlying JMS based message broker system, DRS (Data Replication Service). Sharing characteristics of individual cache entry is configured using the cache policy specification.

- **User profiles**

  Managing **user profiles** allows a company to maintain database tables containing fields for demographic data of individual customers or other users on the company system. For example, when a user repeatedly logs onto a Web site that supports user profiles, the Web site can display headlines and advertising tailored to the shopping preferences of that user. The site can address the user by his or her logon name. User profile API is deprecated in the current release.

Assembly tools

Assembling is an activity in which you package code components into ″modules″ that comply with the J2EE specification. You define configurations for the modules, in the form of XML documents known as deployment descriptors.

5.1 + See ″Chapter 16, “Assembling applications with the Assembly Toolkit,” on page 637.″

Enterprise archive (EAR) files are comprised of the following archives:
- Enterprise bean (JAR) files (known as “EJB modules” on page 83)

- Web archive (WAR) files (known as “Web modules” on page 40)
- Application client (JAR) files (known as “Application clients” on page 117)
- Resource adapter archive (RAR) files (known as resource adapter modules)
- Additional JAR files containing dependent classes or other components required by the application

The standard file extension of an enterprise application file is .ear.

For a discussion of archives and Web components supported by the Assembly Toolkit, see ″“Archive support in Version 5.0” on page 639.″

Deployment

Deployment involves placing applications onto application servers and running the applications. The main tasks include:
1. Installing application files onto an application server.
2. Configuring the application for the particular operational environment.
3. Starting the newly deployed application.

Information on these tasks is available from ″Chapter 17, “Deploying and managing applications,” on page 653.″ The information describes how to deploy applications using the WebSphere Application Server administrative console. You can also deploy applications using the wsadmin tool, which provides deployment capabilities identical to those available using the administrative console.

Packaging and class loading

You can package your business logic as a Java 2 Platform, Enterprise Edition (J2EE) application enterprise archive (EAR) file or as an enterprise bean (EJB) or Web module for deployment to WebSphere Application Server. You must also consider the class loading relationships among modules.

Uninstalling and redeploying applications

At some point, you will need to uninstall your deployed applications. Or you might need to update your applications and deploy them again. You might be able to use hot deployment and dynamic reloading, where you do not need to restart the application server (or the application in some cases) after deploying an updated application.

Chapter 2. Using 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. **(Optional)** Implement JavaScript within JSP tags using the Bean Scripting Framework (BSF).

   5.1 + Support in the JSP Engine for the Bean Scripting Framework is deprecated with WebSphere Application Server 5.1.
4. Develop the Web application, using WebSphere Application Server extensions to enhance its functionality.
5. Assemble the Web application into a Web module using theAssembly Toolkit. Web module assembly properties might include the ability to:
   - Configure servlet page lists.
   - Configure servlet filters.
   - Serve servlets by class name.
   - Enable file serving.
6. 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.
7. **(Optional)** Troubleshoot your Web application.
8. **(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.
9. **(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 servlet 2.3 specification dictates the format of the `web.xml` file, which makes this file portable among Java Two Enterprise Edition (J2EE) compliant products.

© Copyright IBM Corp. 2003

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 Assembly Toolkit.
- If so, what triggers its update?

  The Assembly 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"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.
        //DTD Web Application 2.3//EN"
               "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

   <web-app id="WebApp_1">
      <display-name>Persistence Manager Web Client</display-name>
      <description>Peristence Manager Web Client</description>
      <servlet id="Servlet_1">
        <servlet-name>CustomerLocalServlet</servlet-name>
         <description>Local Customer Servlet</description>
         <servlet-class>CustomerLocalServlet</servlet-class>
     </servlet>
      <servlet id="Servlet_2">
         <servlet-name>CustomerServlet</servlet-name>
        <description>Remote Customer Servlet</description>
         <servlet-class>CustomerServlet</servlet-class>
     </servlet>
     <servlet id="Servlet_3">
         <servlet-name>CreditCardServlet</servlet-name>
         <description>Credit Card Servlet - PM Verification</description>
         <servlet-class>CreditCardServlet</servlet-class>
      </servlet>
       <servlet-mapping id="ServletMapping_1">
        <servlet-name>CustomerLocalServlet</servlet-name>
        <url-pattern>/CustomerLocal</url-pattern>
      </servlet-mapping>
      <servlet-mapping id="ServletMapping_2">
        <servlet-name>CustomerServlet</servlet-name>
         <url-pattern>/Customer</url-pattern>
      </servlet-mapping>
      <servlet-mapping id="ServletMapping_3">
        <servlet-name>CreditCardServlet</servlet-name>
        <url-pattern>/CreditCard</url-pattern>
     </servlet-mapping>
     <welcome-file-list id="WelcomeFileList_1">
         <welcome-file>index.html</welcome-file>
     </welcome-file-list>
     <security-role id="SecurityRole_1">
         <description>Everyone role</description>
         <role-name>Everyone Role</role-name>
     </security-role>
      <security-role id="SecurityRole_2">
         <description>AllAuthenticated role</description>
         <role-name>All Role</role-name>
      </security-role>
      <security-role id="SecurityRole_3">
         <description>Deny all access role</description>
         <role-name>DenyAllRole</role-name>
      </security-role>
  </web-app>
```

Migrating Web application components

Supported open specification levels in WebSphere Application Server Version 5 are documented in article, Migrating.

Migration of Web applications deployed in WebSphere Application Server Version 4.x is not necessary; version 2.2 of the servlet specification and version 1.1 of the JavaServerPages (JSP) specification are still supported. However, where there are behavioral differences between the Java Two Enterprise Edition (J2EE) 1.2 and J2EE 1.3 specifications, bear in mind that J2EE 1.3 specifications are implemented in WebSphere Application Server Version 5 and will override any J2EE 1.2 behaviors.

Servlet migration might be a concern if your application:
- implements a WebSphere internal servlet to bypass a WebSphere Application Server Version 4.x single application path restriction.
- extends a PageListServlet that relies on configuration information in the servlet configuration XML file.
- uses a servlet to generate Hyper Text Markup Language (HTML) output.
- calls the `response.sendRedirect()` method for a servlet using the `encodeRedirectURL` function or executing within a non-context root.

JSP migration might be a concern if your application references JSP page implementation classes in unnamed packages, or if you install WebSphere Application Server Version 4.x EAR files (deployed in Version 4.x with the JSP `Precompile` option), in Version 5.

Follow these steps if migration issues apply to your Web application:
1. Use WebSphere Application Server Version 5 package names for any WebSphere Application Server Version 4.x internal servlets, which are implemented in your application.

   In WebSphere Application Server Version 4.x, Web modules with a context root setting of / are not supported. Accessing Web modules with this root context results in `HTTP 404 - File not Found` errrors.

   To bypass the errors, and to enable the serving of static files from the root context, WebSphere Application Server Version 4.x users are advised to add the servlet class, `com.ibm.servlet.engine.webapp.SimpleFileServlet`, to their Web module.

   The Version 4.x single path limitation does not exist in Version 5. However, users who choose to use the `com.ibm.servlet.engine.webapp.SimpleFileServlet` in Version 5 must do one of the following:
   - Rename `com.ibm.servlet.engine.webapp.SimpleFileServlet` to `com.ibm.ws.webcontainer.servlet.SimpleFileServlet`.
   - Opena Web deployment descriptor editor in the Assembly Toolkit and select **File serving enabled** on the **Extensions** tab.

   The following list identifies the other internal servlets affected by the Version 5 package name change:
   - DefaultErrorReporter
   - AutoInvoker

   Use the Version 5 package name, `com.ibm.ws.webcontainer.servlet.`*`servlet class name`* for these servlets.

2. Use the WASPostUpgrade tool to migrate servlets that extend PageListServlet and rely on configuration information in the associated XML servlet configuration file. In Version 4.x, the XML servlet configuration file provides configuration data for page lists and augments servlet configuration information. This file is named as either `*servlet_class_name*.servlet` or `*servlet_name*.servlet`, and is stored in the same directory as the servlet class file.

   The XML servlet configuration file is not supported in WebSphere Application Server Version 5.
3. Set a content type if your servlet generates Hyper Text Markup Language (HTML) output.

   The default behavior of the Web container changed in WebSphere Application Server Version 5. If the servlet developer does not specify a content type in the servlet then the container is forbidden to set one automatically. Without an explicit content type setting, the content type is set to null. The Netscape browser displays HTML source as plain text with a null content type setting.

   To resolve this problem, do one of the following:
   - Explicitly set a content type in your servlet.
   - Opena Web deployment descriptor editor in the Assembly Toolkit and select **Automatic Response Encoding enabled** on the **Extensions** tab.
4. Set the Java environment variable, `com.ibm.websphere.sendredirect.compatibility`, to **true** if you want your URLs interpreted relative to the application root.

   The default value of the Java environment variable `com.ibm.websphere.sendredirect.compatibility` changed in WebSphere Application Server Version 5. In Version 4, the default setting of this variable is true. In Version 5, the setting is false.

   When this variable is set to false, if a URL has a leading slash, the URL is interpreted relative to the Web module/application root. However, if the URL does not have a leading slash, it is interpreted relative to the Web container root (also known as the Web server document root). Therefore, if an application has a WAR file that has a context root of `myPledge_app` and a servlet that has a servlet mapping of `/Intranet/`, a JSP file in the WAR file cannot access the servlet when its `encodeRedirectURL` is set to `/Intranet/myPledge`. The JSP file can access the servlet if the `encodeRedirectURL` is set to `myPledge_app/Intranet/myPlege`, or if the `com.ibm.websphere.sendredirect.compatibility` variable is set to true.

   See the Setting the sendredirect variable article for more information.
5. Use the WASPostUpgrade tool to migrate WebSphere Version 4.x enterprise applications to Version 5.

   Note: The WebSphere Application Server Version 4.x JSP page implementation class files are not compatible with the WebSphere Application Server Version 5 JSP container.

   The `WASPostUpgrade` tool automatically precompiles JSP files, which ensures the JSP page implementation class files are compatible with Version 5.

   If you install Version 4.x EAR files, deployed with the `JSP Precompile` option, in Version 5, and you choose not to follow the migration path, do one of the following:
   - Select the `Pre-compile JSP` option in the administrative console Install New Application window.

     See article Installing a new application for more information.
   - Specify the `-preCompileJSPs` option when using the Wsadmin tool.
6. Import your classes if your application uses unnamed packages.

   Section 8.2 of the JSP 1.2 specification states:

   ```
   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. The package used may vary between one JSP and another, so
   minimal assumptions should be made. The unnamed package should not be used
   without an explicit import of the class.
   ```

   For example, if `myBeanClass` is in the unnamed package, and you reference it in a `jsp:useBean` tag, then you must explicitly import `myBeanClass` with the page directive import attribute, as shown in the following example:

```
<%@page import="myBeanClass" %>
              . . .
<jsp:useBean id="myBean" class="myBeanClass" scope="session"/>
```

In WebSphere Application Server Version 5, the JSP engine creates JSP page implementation classes in the `org.apache.jsp` package. If a class in the unnamed package is not explicitly imported, then the `javac` compiler assumes the class is in package `org.apache.jsp`, and the compilation fails.

Note: Avoid using the unnamed package altogether because of a change made in JDK 1.4 that will affect the JSP 2.0 specification. WebSphere Application Server Version 5 ships with JDK 1.3.1, so this is not an issue with the Version 5 JSP engine, but it will become an issue in future releases.

The *Incompatibilities* section of the version 1.4.Java 2 Platform, Standard Edition (J2SE) documentation states:

```
     The compiler now rejects import statements that import a type from the
unnamed namespace. Previous versions of the compiler would accept such
import declarations, even though they were arguably not allowed by the
language (because the type name appearing in the import clause is not in
scope). The specification is being clarified to state clearly that you
cannot have a simple name in an import statement, nor can you import from
the unnamed namespace.
     To summarize, the syntax:

        import SimpleName;

is no longer legal. Nor is the syntax

        import ClassInUnnamedNamespace.Nested;

which would import a nested class from the unnamed namespace.
    To fix such problems in your code, move all of the classes from the
unnamed namespace into a named namespace.
```

See ″Resources for learning″ for links to the J2SE, JSP, and Servlet specification documentation.

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/WSamples/`

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
- Perferred 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 incrementing 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 ommit 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 IBM WebSphere Application Server, discussions of servlets focus on Hyper Text Transfer Protocol (HTTP) servlets, which serve Web-based clients.

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 theAssembly Toolkit as a prerequisite to deploying the code to the application server.

Application lifecycle listeners and events

Application lifecycle listeners and events, now part of the Servlet API, enable you to 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 lifecycle 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 associated with a series of requests from a single client. Listeners are available for lifecycle events and for attribute modification events. The listener developer creates a class that implements the javax listener interface, corresponding to the desired listener functionality.

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.

When the servlet context is destroyed, your `contextDestroyed` method is invoked, which releases the database connection, if this context is for your application.

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.

Two new listener interfaces are defined as part of the javax.servlet package:

- ServletContextListener

- ServletContextAttributeListener

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

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

Three new listener interfaces are defined as part of the javax.servlet.http package:
- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionActivationListener

One new 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 is an integral part of the Servlet 2.3 API. 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:
- 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.
- If other filters need to go in the chain, the Web container places them after the `LoggingFilter` and before the requested resource.
- 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.

- 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).
- 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.

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 2.3 Specification
- HttpServletResponseWrapper - methods: See the Servlet 2.3 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 theAssembly Toolkit.

To configure and implement page lists:

1. To configure page list information, use the Add Markup Language entry dialog of the Assembly Toolkit. On the **Servlets** tab of a Web deployment descriptor editor, select a servlet and click **Add** under **WebSphere Extensions**.
2. Add the `callPage()` method to your servlet to invoke a JavaServer Page (JSP) file in response to a client request.

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

     where the method arguments are:

     `pageName`
     A page name defined in the PageListServlet configuration

     `request`
     The HttpServletRequest object

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

     where the method arguments are:

     `mlName` A markup language type

     `pageName`
     A page name defined in the PageListServlet configuration

     `request`
     The HttpServletRequest object

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

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(HttpServletRequestrequest)`, provided by the PageListServlet, inspects the HttpServletRequest object request headers, and searches for a match in the `client_types.xml` file.

The client type detection method does the following:
- Uses the input HttpServletRequest and the `client_types.xml` file, to check for a matching HTTP request name and value.
- Returns the markup-language value configured for the `<client-type>` element, if a match is found.

  If multiple matches are found, this method returns the markup-language for the first `<client-type>` element for which a match is found.
- If no match is found, returns the value of the markup-language for the default page defined in the PageListServlet configuration.

Location

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

Usage notes
- Is this file read-only?

  No
- Is this file updated by a product component?

  No
- If so, what triggers its update?

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

  Servlets, 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 the correct markup-language for the response is *Hello.page*.

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

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

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

autoRequestEncoding and autoResponseEncoding

Two new WebSphere Application Server extensions are available in Version 5, `autoRequestEncoding` and `autoResponseEncoding`.

In 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. 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 the Assembly Toolkit to change the default values for the `autoRequestEncoding` and `autoResponseEncoding` extensions.

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

Examples: autoRequestEncoding and autoResponseEncoding encoding examples

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

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

The Web container performs each step in the following list until a match is found:
- Looks at the character set (charset) in the *Content-Type* header.
- Attempts to map the servers locale to a character set using defined properties.
- Attempts to use the `DEFAULT_CLIENT_ENCODING` system property, if one is set.
- Uses the ISO-8859-1 character encoding as the default.

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

JavaServer Pages files

JavaServer Pages (JSP) files are application components coded to the Sun Microsystems JavaServer Pages (JSP) Specification. JSP files 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.

The IBM extensions to the JSP Specification include JSP tags that resemble HTML tags. These JSP tags make it easy for HTML authors to add the power of Java technology to Web pages, without being experts in Java programming.

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.

Developing JavaServer Pages files with WebSphere extensions

Several IBM WebSphere extensions are provided for enhancing your JavaServer Pages (JSP) files. This task provides a summary of the extensions that you can utilize.

1. Review the supported specifications.

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

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

   WebSphere Application Server Version 3.5 added IBM extensions to the base Application Programming Interfaces (APIs). Since the JavaServer Pages (JSP) 1.1 and JSP 1.2 Specifications are backward compatible to the JSP 1.0 Specifications, you can invoke the APIs with the IBM extensions without modification.

   The extensions belong to these categories:

   Syntax for variable data
   : Put variable fields in JSP files and have servlets and beans dynamically replace the variables with values from a database when the JSP output is returned to the browser.

   Syntax for database access
   : Add a database connection to a Web page and then use that connection to query or update the database. You can provide the user ID and password for the database connection at request time, or you can hard code the user ID and password within the JSP file.
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.
4. **(Optional)** Batch compile your JSP files if necessary.

Tag libraries

Java ServerPages (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 JAR file.

tsx:dbconnect tag JavaServer Pages syntax

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

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

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

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

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

where:

- **id**

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

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

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

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

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

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

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

    ```
    url="jdbc:db2:sample"
    driver="COM.ibm.db2.jdbc.app.DB2Driver"
    ```
  - For an ODBC database, use the Sun JDBC-to-ODBC bridge driver included in their Java2 Software Developers Kit (SDK) or another vendor's ODBC driver.

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

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

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

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

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

- **jndiname**

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

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

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

dbquery tag JavaServer Pages syntax

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 contect. 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:getProperty name="request" property=request.getParameter("corporation") />
```

In most cases, the value of the property attribute is just the property name. However, to access the request bean or to access a property of a property (sub property), specify the full form of the property attribute. The full form also gives you the option to specify an index for indexed properties. You can specify the optional index as a constant (such as 2), or an index like the one described in the <tsx:repeat> tag. Some examples using the full form of the property attribute follow:

```
<tsx:getProperty name="staffQuery" property=address(currentAddressIndex) />
<tsx:getProperty name="shoppingCart" property=items(4).price />
<tsx:getProperty name="fooBean" property=foo(2).bat(3).boo.far />
```

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.

The output of a <tsx:repeat> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

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 value is case-sensitive and its scope is the JSP file.
- **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="FIRSTNME" />
    <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**
myBean.class	**JSP file**
`// Code to get a connection`	`<tsx:dbconnect id="conn"`
	`userid="alice"passwd="test"`
`// Code to get the data`	`url="jdbc:db2:sample"`
`   Select * from myTable;`	`driver="COM.ibm.db2.jdbc.app.DB2Driver">`
	`</tsx:dbconnect >`
`// Code to close the connection`	
	`<tsx:dbquery id="dynamic"`
JSP file	` connection="conn" >`
`<tsx:repeat index=abc>`	`  Select * from myTable;`
`  <tsx:getProperty name="myBean"`	`</tsx:dbquery>`
`    property="col1(abc)" />`	
`</tsx:repeat>`	`<tsx:repeat index=abc>`
	`  <tsx:getProperty name="dynamic"`
	`    property="col1(abc)" />`
	`</tsx:repeat>`
Notes:	**Notes:**
• The bean (myBean.class) is a static bean.	• 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 myBean.get(*property*(*index*)).	• The method to access the bean properties is dynamic.getValue(*"property"*, *index*).
• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.	• 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.	• 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>
```

JspBatchCompiler tool

As an IBM enhancement to JavaServer Pages support, IBM WebSphere Application Server provides a batch JSP compiler. 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.

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

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

```
JspBatchCompiler -enterpriseapp.name <name>
                              [ -webmodule.name <name>]
                              [ -cell.name  <name>]
                              [ -node.name  <name>]
                              [ -server.name  <name>]
                              [ -filename <jsp name>]
                              [ -keepgenerated <true|false>]
                              [ -verbose <true|false>]
                              [ -deprecation <true|false>]
```

If the names specified for these arguments are comprised of two or more words separated by spaces, you must add quotation marks around the names.

where:

- **enterpriseapp.name**

  Represents the name of the enterprise application you want to compile.
- **webmodule.name**

  Represents the name of the specific Web module that you want to compile. If this argument is not set, all Web modules in the enterprise application are compiled.
- **cell.name**

  Represents the name of the cell in which the application is deployed. The default is `BaseApplicationServerCell`.
- **node.name**

  Represents the name of the node in which the application is deployed. The default is `DefaultNode`.
- **server.name**

  Represents the name of the server in which the application is deployed. The default is `server1`.
- **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 are compiled.
- **keepgenerated**

  Represents the option to save or erase the generated files.

  If set to `yes`, WebSphere Application Server saves the generated `.java` files used for compilation on your server. By default, this argument is set to `no` and the `.java` files are erased after the class files have compiled.
- **verbose**

  Indicates the compiler should generate verbose output while compiling the generated sources.
- **deprecation**

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

Bean Scripting Framework

The Bean Scripting Framework (BSF) enables you to use scripting language functions in your Java server-side applications. This framework also extends scripting languages so that you can use existing Java classes and Java beans in the JavaScript language. Support in the JSP Engine for the Bean Scripting Framework is deprecated with WebSphere Application Server 5.1.

With BSF, you can write scripts that create, manipulate and access values from Java objects, or you can write Java programs that evaluate and access results from scripts.

WebSphere Application Server provides the Bean Scripting Framework, which consists of a BSF manager, a BSF engine, and a scripting engine.

BSF provides an access mechanism to Java objects for the scripting languages it supports, so that both the scripting language and theJava code can access code exclusive functions. The access mechanism is implemented through a registry of objects maintained by BSF.

BSF in WebSphere Application Server supports the Rhino ECMAScript.

The ″Resources for Learning″ article provides external BSF links that document future supported languages.

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.

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. In previous releases of WebSphere Application Server, if a `.class` file did not exist, the JSP engine always translated and compiled the JSP file. You had to turn off the Web applications reload capability to prevent additional translations and recompiles of the file.

With Version 5.0.1 of WebSphere Application Server, you can now change the JSP engine default behavior 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` and `.class` files generated during a run-time compilation.
- Forces you to verify 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** >
  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 16, “Assembling applications with the Assembly Toolkit,” on page 637.

  Web Modules > *component_instance* > **Assembly Property Extensions**

  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 attribute assembly 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. Subsequent requests for the file also result in compilations and translations, but only if the following conditions are met:
  – Compilations and translations are required.
  – 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 501 (Not implemented) to the browser. If the JSP file does not exist, the JSP engine returns HTTP error 404 (File not found) to the browser. In both cases, 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.

Perform the following steps to determine whether the `disableJspRuntimeCompilation` option is enabled in WebSphere Application Server:

1. Enable the Diagnostic Trace Service and set the trace specification to `com.ibm.ws.webcontainer.jsp.servlet.*=all=enabled`.
2. Request a JSP file.
3. Locate the string, `disableJspRuntimeCompilation:true`, in the `trace.log` file.
4. Ensure the `jspUri:` entry matches the requested JSP file.

If both the `disableJspRuntimeCompilation:true` string and the matching `jspUri:` entry appear in the trace, the `disableJspRuntimeCompilation` setting is enabled for the Web application.

Example: Converting JavaScript source to the Bean Scripting Framework

JavaScript code is one of the most popular languages of Web developers. This language supports the following base objects, plus additional objects from the Document Object Model:
- array
- date
- math
- number
- string

Server-side JavaScript code supports the same base objects, and additional objects that support user access to databases, file systems and e-mail systems.

Like client-side JavaScript code, server-side JavaScript code is also platform, browser, and language independent.

You can convert server-side JavaScript applications to the Bean Scripting Framework. This article describes how to perform this conversion.

Server-side JavaScript source code

Suppose you have the following server-side JavaScript application:

```
<html>
<head>
<title>Hello World server-side JavaScript example</title>
</head>
<body>
<br><br>
</body>
</html>

<server>
function writePage()
  write("<center><font size='6'>Hello World</font></center>");
</server>
```

Converting server-side JavaScript source code to the Bean Scripting Framework (BSF)

Make the following changes to the JavaScript source code to enable BSF:

```
<%@  page language="javascript" %>
<html>
<head>
<title>Hello World server-side BSF/JavaScript example</title>
</head>
<body>
<br><br>
</body>
</html>

<%
  out.println("<center><font size='6'>Hello World</font></center>");
%>
```

Review the other BSF reference articles for deployment information and additional programming examples.

Scenario: Creating a Bean Scripting Framework application

Scenario description

Programming skills in JavaScript code are more prevalent than programming skills using JavaServer Pages (JSP) tags. Using the Bean Scripting Framework, JavaScript programmers can gradually introduce JSP tags in their JavaScript applications without completely rewriting the source code. The BSF method not only reduces the potential of programming errors, but also provides a painless way to learn a new technology.

The following scenario illustrates how to implement a BSF application using JavaScript within JSP tags.

Developing the BSF application

At ABC elementary school, John Doe teaches third grade mathematics. He wants to help his students memorize their multiplication tables, and thinks a small Web-based quiz could help meet his objective. However, John Doe only knows JavaScript.

Using the Bean Scripting Framework to help leverage his JavaScript skills, John Doe creates two JSP files, `multiplication_test.jsp` and `multiplication_scoring.jsp`.

In the `multiplication_test.jsp` file, John Doe uses both client-side and server-side JavaScript code to generate a test of 100 random multiplication questions, displayed using a three minute timer. He then writes the `multiplication_scoring.jsp` file to read the data submitted by the `multiplication_test.jsp` file and to generate the scoring results.

John Doe creates the following two files:

`multiplication_test.jsp:`
```
<html>
<head>
<title>Multiplication Practice Test</title>
<script language="javascript">
var countMin=3;
var countSec=0;
function updateDisplay (min, sec) {
    var disp;
    if (min <= 9) disp = " 0";
    else disp = " ";
    disp += (min + ":");
    if (sec <= 9) disp += ("0" + sec);
    else disp += sec;
    return(disp);
}
function countDown() {
    countSec--;
    if (countSec == -1) {
        countSec = 59;
        countMin--;
    }
    document.multtest.counter.value = updateDisplay(countMin, countSec);
    if((countMin == 0) &&(countSec == 0)) document.multtest.submit();
    else var down = setTimeout("countDown();", 1000);
}
</script>
</head>
<body bgcolor="#ffffff" onLoad="countDown();">
<%@ page language="javascript" %>
<h1>Three Minute Multiplication Drill</h1>
<hr>
<h2>Remember: this is an opportunity to excel!</h2>
<p>
```

```
<form method="POST" name="multtest" action="multiplication_scoring.jsp">
<div align="center">
<table>
<tr>
<td>
<h3>Time left:
<input type="text" name="counter" size="9" value="03:00" readonly>
</h3>
</td>
<td>
<input type="submit" value="Submit for scoring!">
</td>
</tr>
</table>
<table border="1">
<%
var newrow = 0;
var q_num = 0;
function addQuestion(num1, num2) {
    if (newrow == 0) out.println("<tr>");
    out.println("<td>");
    out.println(num1 + " x " + num2 + " = ");
    out.println("</td><td>");
    out.print("<input name=\"" + q_num + "|" + num1 + ":" + num2 + "\" ");
    out.println("type=\"text\" size=\"10\">");
    out.println("</td>");
    if (newrow == 3) {
        out.println("</tr>");
        newrow = 0;
    }
    else newrow++;
    q_num++;
}
for (var i = 0; i < 100; i++) {
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);
    addQuestion(rand1, rand2);
}
%>
</table>
</div>
</form>
</body>
</html>
```

multiplication_scoring.jsp:

```
<html>
<head>
<title>Multiplication Practice Test Results</title>
</head>
<body bgcolor="#ffffff">
<%@ page language="javascript" %>
<h1>Multiplication Drill Score</h1>
<hr>
<div align="center">
<table border="1">
<tr><th>Problem</th><th>Correct Answer</th><th>Your Answer</th></tr>
<%
var total_score = 0;
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;
    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");
    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}
var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
   var currElt = equations.nextElement();
   var splitPos1 = currElt.indexOf("|");
   var splitPos2 = currElt.indexOf(":");
   if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}
%>
</table>
<h2>Total Score: <%= total_score %></h2>
<h3><a href="multiplication_test.jsp">Try again?</a></h3>
</div>
</body>
</html>
```

Follow these steps to see how John Doe uses BSF to implement JavaScript in a JSP application:

1. Give your files a `.jsp` extension.
2. Use server-side JavaScript code in your application.

   The `multiplication_test.jsp` file incorporates both client-side and server-side JavaScript. Server-side JavaScript is similar to client-side JavaScript; the primary difference consists of using a different set of objects. Whereas client-side Javascript programmers invoke document and window objects, server-side JavaScript programmers, using the Bean Scripting Framework, invoke a set of objects provided by the JSP technology. Also, client-side scripts are enclosed in <script> tags, but server-side scripts use JSP scriptlet and expression tags.
3. Examine the following blocks of code:

   ```
   <script language="javascript">
   var countMin=3;
   var countSec=0;
   function updateDisplay (min, sec) {
       var disp;
       if (min <= 9) disp = " 0";
       else disp = " ";
       disp += (min + ":");
       if (sec <= 9) disp += ("0" + sec);
       else disp += sec;
       return(disp);
   }
   function countDown() {
       countSec--;
       if (countSec == -1) {
           countSec = 59;
           countMin--;
       }
       document.multtest.counter.value = updateDisplay(countMin, countSec);
       if((countMin == 0) && (countSec == 0)) document.multtest.submit();
       else var down = setTimeout("countDown();", 1000);
   }
   </script>
   ....
   <body bgcolor="#ffffff" onLoad="countDown();">
   ...
   ```

```
<form method="POST" name="multtest" action="multiplication_scoring.jsp">
...
<input type="text" name="counter" size="9" value="03:00" readonly>
...
```

The JavaScript code contained in the <script> block implements a timer set within the <input> field named `counter`. The `onLoad` event handler in the <body> tag causes the browser to load and execute the code when the the page is loaded.

The `document.multtest.submit()` statement causes the form named `multtest` to be submitted when the timer expires.

4. Identify the code to the BSF function.

   The following code example, from the `multiplication_test.jsp` file, displays the use of a JSP directive. This directive tells the WebSphere Application Server BSF function that this file is using the JavaScript language, and that the JavaScript code is enclosed by the `<% ... %>` scriptlet tags. The `out` implicit JSP object in this code example, creates the body section of a table from 100 randomly generated questions.

```
...
<%@ page language="javascript" %>
...
<%
var newrow = 0;
var q_num = 0;

function addQuestion(num1, num2) {
    if (newrow == 0) out.println("<tr>");

    out.println("<td>");
    out.println(num1 + " x " + num2 + " = ");
    out.println("</td><td>");
    out.print("<input name=\"" + q_num + "|" + num1 + ":" + num2 + "\" ");
    out.println("type=\"text\" size=\"10\">");
    out.println("</td>");

    if (newrow == 3) {
        out.println("</tr>");
        newrow = 0;
    }
    else newrow++;

    q_num++;
}

for (var i = 0; i < 100; i++) {
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);

    addQuestion(rand1, rand2);
}

%>
...
```

5. Read the results.

   To score the results of the practice drill, John Doe uses the `request` implicit JSP object in the `multiplication_scoring.jsp` file to obtain the `POST` data created within the <form> tags in the `multiplication_test.jsp` file.

   The `multiplication_scoring.jsp` file uses the `POST` data to build an output file containing the original question, the student's answer, and the correct answer, and then prints the text in a table format using the `out` implicit object.

   The following code example from the `multiplication_scoring.jsp` file illustrates the use of the `request` and `out` JSP objects:

```
...
<%@ page language="javascript" %>
...
<%
var total_score = 0;
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;
    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");
    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}
var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
   var currElt = equations.nextElement();
   var splitPos1 = currElt.indexOf("|");
   var splitPos2 = currElt.indexOf(":");
   if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}

%>
...
<h2>Total Score: <%= total_score %></h2>
...
```

Note: Although using separate scriptlet blocks of code for different portions of a conditional expression is common in JSP files implemented in Java, it is invalid for JSP files implemented using JavaScript through the Bean Scripting Framework. The JavaScript code must be entirely contained within the scriptlet tags.

The following code example illustrates invalid usage:

```
<% if (pass == 0) %>
    <i>pass is true</i>
<% else %>
    <i>pass is not true</i>
```

Deploying the BSF application

You assemble and deploy BSF applications in the same manner as JSP applications. Review the Assembling applications article for more information.

Deploy the BSF code examples in WebSphere Application Server to view this applications processing and output. Use the following quick steps to deploy the application.

The intent of these ″quick steps″ is to provide you with instant application output. However, the supported method for deployment is the same as for standard JSP files.

1. Use the DefaultApplication to add your BSF files.

   Copy your `.jsp` files to the DefaultApplication directory:

   `<`*`app server install directory`*`>/installedApps/<`*`node name`*`>/DefaultApplication.ear/DefaultApplication.war`

2. Start the application server.
3. Open a browser and request your BSF application.

   Use the following URL to request your application:

   `http://hostName:9080/<jspFileName>.jsp`

Example: Bean Scripting Framework code example

The following code examples show how to implement JavaScript using the Bean Scripting Framework (BSF).

For a quick demonstration of the BSF function, copy these code examples into 2 separate files, and deploy them in WebSphere Application Server using the instructions in the BSF scenario article.

Multiplication practice test

```
<html>
<head>
<title>Multiplication Practice Test</title>
<!--
This file and its companion, multiplication_score.jsp, illustrate the
use of ECMAScript within the BSF framework. The task is a simple
timed math quiz, which is 3 minutes in duration.  When the quiz ends,
the score is computed and displayed.
Users are  then asked if they wish to try
the quiz again.
-->

<!--
This code fragment  displays and  updates the quiz
countdown in client side JavaScript code.
-->
<script language="javascript">
var countMin=3;
var countSec=0;

// This code computes the current countdown time.
function updateDisplay (min, sec) {
    var disp;

    if (min <= 9) disp = " 0";
    else disp = " ";

    disp += (min + ":");

    if (sec <= 9) disp += ("0" + sec);
    else disp += sec;

    return(disp);
}

//This code fragment displays  the current countdown time in the user's
//browser window,and submits the results for scoring when the countdown
//ends.

function countDown() {
    countSec--;
    if (countSec == -1) {
        countSec = 59;
        countMin--;
    }
    document.multtest.counter.value = updateDisplay(countMin, countSec);
    if((countMin == 0) && (countSec == 0)) document.multtest.submit();
    else var down = setTimeout("countDown();", 1000);
}
```

```
</script>
</head>
<body bgcolor="#ffffff" onLoad="countDown();">

<!--
The body of the quiz runs as  JavaServer Pages (JSP) code using BSF.
The code  outputs the problems in table format using the  POST method
and invokes  the scoring module when the user chooses to end the quiz
or when the countdown ends.
-->
<%@ page language="javascript" %>

<h1>Three Minute Multiplication Drill</h1>
<hr>

<h2>Remember: this is an opportunity to excel!</h2>
<p>

<form method="POST" name="multtest" action="multiplication_scoring.jsp">
<div align="center">
<table>
<tr>
<td>
<h3>Time left:
<input type="text" name="counter" size="9" value="03:00" readonly>
</h3>
</td>
<td>
<input type="submit" value="Submit for scoring!">
</td>
</tr>
</table>
<table border="1">
<%
var newrow = 0;
var q_num = 0;

// This code  generates  a new random multiplication problem up to the number
//twelve, and enters  it into the table of problems.

function addQuestion(num1, num2) {
    if (newrow == 0) out.println("<tr>");

    out.println("<td>");
    out.println(num1 + " x " + num2 + " = ");
    out.println("</td><td>");
    out.print("<input name=\"" + q_num + "|" + num1 + ":" + num2 + "\" ");
    out.println("type=\"text\" size=\"10\">");
    out.println("</td>");

    if (newrow == 3) {
        out.println("</tr>");
        newrow = 0;
    }
    else newrow++;

    q_num++;
}


//This code obtains two random operands and formats 100 quiz problems.

for (var i = 0; i < 100; i++) {
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);
    addQuestion(rand1, rand2);
}

%>
</table>
</div>
</form>

</body>
</html>
```

Multiplication practice test results

```
<html>
<head>
<title>Multiplication Practice Test Results</title>
</head>
<body bgcolor="#ffffff">

<!--
This JSP code is invoked when the user submits a math quiz for scoring,
or when the quiz countdown expires. The JSP code tabulates the problem list,
the correct answer, the user's answer, and scores the test. It then offers
the user an opportunity to try the quiz again.
-->
<%@ page language="javascript" %>

<h1>Multiplication Drill Score</h1>
<hr>

<div align="center">
<table border="1">
<tr><th>Problem</th><th>Correct Answer</th><th>Your Answer</th></tr>
<%
var total_score = 0;

// This code parses the submitted form, extracts the a problem generated by the
// multiplication_test.jsp file, outputs it, computes the correct answer,
// and displays  this information and  the user answer.  The code scores
// the quiz using a running sum of correct answers.

function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;

    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");

    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}


// This is the main body of the scoring application. It parses the posted quiz,
// and calls  the score() function to score remaining problems.

var equations = request.getParameterNames();
```

```
while(equations.hasMoreElements()) {
   var currElt = equations.nextElement();
   var splitPos1 = currElt.indexOf("|");
   var splitPos2 = currElt.indexOf(":");

   if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}

%>
</table>

<h2>Total Score: <%= total_score %></h2>
<h3><a href="/multiplication_test.jsp">Try again?</a></h3>
</div>

</body>
</html>
```

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 J2EE applications. You install and run a Web module in the Web container of an application server.

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 stand-alone application or combine it with other modules into an enterprise application.

Use the Assembly Toolkit 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 the Assembly Toolkit.
2. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).

4. Migrate WAR files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import your WAR files to the Assembly Toolkit.
5. Create a new Web module.
6. Copy code artifacts (such as servlets) from one Web module into a new Web module.
7. Verify the contents of the new Web module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Web Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the Web module in a Navigator view.

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 is 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.

Servlet 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.

Invoker attributes

Invoker attributes are used by the servlet that implements the invocation behavior.

Error pages

Error page locations allow a servlet to find and serve a URI to a client based on a specified error status code or exception type.

These properties are used if the error handler is another servlet or JSP file. The properties specify a mapping between an error code or exception type and the path of a resource in the Web application. The container examines the list in the order that it is defined, and attempts to match the error condition by status code or by exception class. On the first successful match of the error condition, the container serves back the resource defined in the Location property.

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.

Initialization parameters

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

Servlet caching

Dynamic caching can be used to improve the performance of servlet and JavaServer Pages (JSP) files by serving requests from an in-memory cache. Cache entries contain the servlet's output, results of the servlet's execution, and metadata.

Web components

A web component is a servlet, Java Server Page (JSP), 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.

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.

Symptom	Your Web module does not run when you migrate it to Version 5
Problem	In Version 4.x, the classpath setting that affected visibility was *Module Visibility Mode*. In Version 5, 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. in the class loader policies. See article Migration of module visibility modes from Version 4.x for more information and examples.

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.

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.

Modifying the default Web container configuration

The Web container is created initially with default properties values suitable for simple Web applications. However, these values might not be appropriate for more complex Web applications.

Your application is considered complex if it requires any of the following features:
- virtual host
- servlet caching
- special client request loads
- persistent HTTP session support
- special HTTP transport settings
- transaction class mappings

Modify the following properties if you have a complex application:

1. If your Web application requires a virtual host, other than the default_host, or requires servlet caching, modify the Web container **General Properties**.
2. If your application handles special client request loads, modify the Web Container **Additional Properties > Thread Pool** setting.
3. If your application requires persistent HTTP session support, modify the Web Container **Additional Properties > Session Management** setting.
4. If your application requires one of the following HTTP transport settings:
   - Unique hostname and port for client access
   - SSL enablement

   modify the Web Container **Additional Properties > HTTP transports** setting.
5. If your application requires global settings for internal servlets for WAR files packaged by third-party tools, modify the Web Container **Additional Properties > Custom Properties** setting.
6. If your application uses transaction class mappings to classify workload, modify the Web Container **Additional Properties > Advanced Settings**.

Web container

A Web container handles requests for servlets, JavaServer Pages (JSP) files, and other types of files that include server-side code. The Web container creates servlet instances, loads and unloads servlets, creates and manages request and response objects, and performs other servlet management tasks.

The Web server plug-ins, provided by the WebSphere Application Server, help supported Web servers pass servlet requests to Web containers.

Web container settings

Use this page to configure the web container settings.

To view this administrative console page, click **Servers** > **Application Servers** > *server_instance* > **Web container**.

Configuration - General Properties

Default virtual host

Specifies a virtual host that enables a single host machine to resemble multiple host machines. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine.
Select a virtual host option:

Default Host
: The product provides a default virtual host with some common aliases, such as the machine IP address, short host name, and fully qualified host name. The alias comprises the first part of the path for accessing a resource such as a servlet. For example, it is `localhost:9080` in the request `http://localhost:9080/myServlet`.

Admin Host
: This is another name for the application server; also known as *server1* in the base installation. This process supports the use of the administrative console.

Servlet caching

Specifies that if a servlet is invoked once and it generates output to be cached, a cache entry is created containing not only the output, but also side effects of the invocation. These side effects can include calls to other servlets or Java Server Pages (JSP) files, as well as metadata about the entry, including timeout and entry priority information.

Enable servlet caching
: Check this box to enable servlet caching.

Web module settings

Use this page to configure Web module settings.

To view this page in the Application Assembly Tool, click

`Applications > Enterprise Application > application_instance > Web Module`

URI

Specifies a URI that, when resolved relative to the application URL, specifies the location of the module archive contents on a file system. The URI must match the ModuleRef URI in the deployment descriptor of an application if the module was packaged as part of a deployed application or enterprise archive (EAR) file.

Name

Specifies the unique display name for the module.

Alternate DD

Specifies the file name for an alternative deployment descriptor file to use instead of the original deployment descriptor file in the module JAR file.
This file is the *post-assembly* version of the deployment descriptor file. You can edit the original deployment descriptor file to resolve dependencies and security information. Specifying the use of the alternative deployment descriptor keeps the original deployment descriptor file intact.

The value of the *Alternate DD* property must be the full path name of the deployment descriptor file, relative to the module root directory. By convention, the file is in the `ALT-INF` directory. If this property is not specified, the deployment descriptor file is read from the module JAR file.

Starting weight

Specifies the order in which modules are started. Lower weighted modules are started before higher weighted modules.

Prefer WEB-INF Classes

Specifies classes to load in WEB-INF before any other classes. Implementing the application class loader is recommended so that classes and resources packaged within the WAR file load before classes and resources residing in container-wide library JAR files.

Initial State

Specifies the default state of this application at server startup.

Web Module Deployment settings

Use this page to configure an instance of Web module deployment.

To view this administrative console page, click **Applications** > **Enterprise Application** > *application_instance* > **Web Modules** > *Web Module_instance*.

URI

Specifies a URI that, when resolved relative to the application URL, specifies the location of the module archive contents on a file system. The URI must match the ModuleRef URI in the deployment descriptor of an application if the module was packaged as part of a deployed application or enterprise archive (EAR) file.

Alternate DD

Specifies the file name for an alternative deployment descriptor file to use instead of the original deployment descriptor file in the module JAR file.
This file is the *post-assembly* version of the deployment descriptor file. You can edit the original deployment descriptor file to resolve dependencies and security information. Specifying the use of the alternative deployment descriptor keeps the original deployment descriptor file intact.

The value of the *Alternate DD* property must be the full path name of the deployment descriptor file, relative to the module root directory. By convention, the file is in the `ALT-INF` directory. If this property is not specified, the deployment descriptor file is read from the module JAR file.

Starting weight
Specifies the order in which modules are started. Lower weighted modules are started before higher weighted modules.

Classloader Mode
Specifies whether the class loader should search in the parent class loader or in the application class loader first to load a class. The standard for JDK class loaders and WebSphere 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 overriden classes and non-overriden 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

Web container custom properties

Use this page to configure arbitrary name-value pairs of data, where the name is a property key and the value is a string value that can be used to set internal system configuration properties. Defining a new property enables you to configure a setting beyond that which is available in the administrative console.

To view this administrative console page, click **Servers > Application Servers >***server_name***> Web Container > Custom Properties**.

Name
Specifies the name (or key) for the property.

Data type	String

Value
Specifies the value paired with the specified name.

Data type	String

Description
Provides information about the name-value pair.

Data type	String

Global settings for internal servlets

Web Archive (WAR) files packaged using third-party tools cannot specify behavior for the services exposed by the Web container internal servlets. You can globally enable/disable internal servlets for all Web applications at the Web container level by creating name-value pairs such as:

Name	Value
fileServingEnabled	true
directoryBrowsingEnabled	true
serveServletsByClassnameEnabled	true

Settings defined in an assembly tool level take precedence over the global settings set through the custom properties at the Web container level.

Web application deployment extensions continue to hold configuration information for the services provided by the internal servlets, and take precedence over the global settings set through the custom properties at the Web container level.

UTF-8 encoded URLs

WebSphere Application Server Version 5.1, introduces support for UTF-8 encoded Uniform Resource Locators (URLs) to support the double byte characters in URLs. The UTF-8 encoded URL feature is enabled by default. You can prevent the web container from explicitly decoding URLs in UTF-8 and have them use the ISO-8859 standard as per the current HTTP specification by using the following name-value pair:

Name	Value
DecodeUrlAsUTF8	false

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"
- "Web applications: Resources for learning"
- "Web applications: Resources for learning"

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^{TM} Tutorial on Servlets
- Introduction to JavaServer Pages - Tutorial
- Bean Scripting Framework description
- Web delivered samples in the Samples Gallery

Programming specifications
- Java 2 Software Development Kit (SDK)
- Servlet 2.3 Specification
- JavaServer Pages 1.2 Specification
- Differences between JavaScript and ECMAScript
- ISO 8859 Specifications

Chapter 3. 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 2.3 API.

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

Sessions

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

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

For example, a servlet might use sessions to provide ″shopping carts″ to online shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she wants to purchase from the Web site. It is important that the servlet be able to associate incoming requests with particular shoppers. Otherwise, the servlet might mistakenly add Shopper_1's choices to the cart of Shopper_2.

A servlet distinguishes users by their unique session IDs. The session ID arrives with each request. If the user's browser is cookie-enabled, the session ID is stored as a cookie. As an alternative, the session ID can be conveyed to the servlet by URL rewriting, in which the session ID is appended to the URL of the servlet or JavaServer Pages (JSP) file from which the user is making requests. For requests over HTTPS or Secure Sockets Layer (SSL), Another alternative is to use SSL information to identify the session.

Migrating HTTP sessions

Note: In Version 5 default write frequency mode is TIME_BASED_WRITES, which is different from Version 4.0 and 3.5 default mode of END_OF_SERVICE.

Migrating from Version 4.0

No programmatic changes are required to migrate from version 4.0 to version 5.

Migrating from Version 3.5

If you have Version 3.5 applications running in Servlet 2.1 mode, some of the following Version 5 differences might influence how you choose to track and manage sessions.

1. During application development, modify session-related APIs as needed.

   Some API changes are required in order to redeploy existing applications on Version 5. These include changes to the HttpSession API itself as well as issues associated with moving to support for the Servlet 2.3 specification. Certain Servlet 2.1 API methods have been deprecated in Servlet 2.3 API . These deprecated APIs still work in Version 5.0, but they may be removed in a future version of the API. Changes are summarized in the following list:

- Replace instances of getValue() with getAttribute()
- Replace instances of getValueNames() with getAttributeNames()
- Replace instances of removeValue() with removeAttribute()
- Replace instances of putValue() with setAttribute()

2. During application development, modify Web application behavior as needed.

   In accordance with the Servlet 2.3 specification, HttpSession objects must be scoped within a single Web application context; they may not be shared between contexts. This means that a session can no longer span Web applications. Objects added to a session by a servlet or JSP in one Web application cannot be accessed from another Web application. The same session ID may be shared (because the same cookie is in use), but each Web application will have a unique session associated with the session ID. Version 5 provides a feature that can be used to extend scope of a session to enterprise application.
3. Use administrative tools to configure Session Manager security settings as needed. Relative to session security, the default Session Manager setting for Integrate Security is now false. This is different from the default setting in some earlier releases.
4. Use administrative tools to configure the JSP enabler and application server as needed.

   In Version 3.5 of the product, JSP files that contained the usebean tag with scope set to session did not always work properly when session persistence was enabled. Specifically, the JSP writer needed to write a scriplet to explicitly set the attribute (that is, to call setAttribute()) if it was changed as part of JSP processing.

   Two new features in Version 5.0 help address this problem:
   - You can set dosetattribute to true on the JSP InitParameter.
   - You can set the Write Contents option to Write all.

   The differences between the two solutions are summarized in the following table:

	Applies to	**Configured at**	**Action**
dosetattribute set to *true*	JSP	JSP enabler	Assures that JSP session-scoped beans always call setAttribute()
Write Contents option set to *Write all*	servlet or JSP	application server	All session data (changed or unchanged) is written to the external location

   If session persistence is enabled and a class reload for the Web application occurs, the sessions associated with the Web application are maintained in the persistent store and will be available after the reload.

Developing session management in servlets

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

1. Get the HttpSession object.

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

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

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

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

2. Store and retrieve user-defined data in the session.

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

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

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

Example: SessionSample.java

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


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


   // Step 1: Get the Session object

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

   // Step 2: Get the session data value


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

   // Step 3: Output the page

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

Assembling so that session data can be shared

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

Restriction: To use this option, you must install all the Web modules in the enterprise application on a given server. You cannot split up Web modules in the enterprise application by servers. For example, with an enterprise application containing two Web modules, you cannot use this option when one Web module is installed on one server and second Web module is installed on a different server. In such split installations, applications might share session attributes across Web modules using distributed sessions, but session data integrity is lost when concurrent access to a session is made in different Web modules. It also severely restricts use of some Session Management features, like TIME_BASED_WRITES. For enterprise applications on which this option is enabled, the Session Management configuration on the Web module inside the enterprise application is ignored. Then Session Management configuration defined on enterprise application is used if Session Management is overwritten at the enterprise application level. Otherwise, the Session Management configuration on the Web container is used.

Servlet API Behavior

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

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

1. Start theAssembly Toolkit.
2. In the Assembly Toolkit, 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 Toolkit, 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.
4. Save the application (EAR) file. In the Assembly Toolkit, after you close the application deployment descriptor editor, confirm that you want to save changes made to the application.

Session security support

You can integrate HTTP sessions and security in IBM 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 unsecure page.

Programmatic details and scenarios

IBM 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`. IBM 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 or not.

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

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

* A com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException error is thrown 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 module. 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.

Configuring session management by level

When you configure session management at the Web container level, all applications and the respective Web modules in the Web container normally inherit that configuration, setting up a basic default configuration for the applications and Web modules below it.

However, you can set up different configurations individually for specific applications and Web modules that vary from the Web container default. These different configurations override the default for these applications and Web modules only.

Note: When you overwrite the default session management settings on the application level, all the Web modules below that application inherit this new setting unless they too are set to overwrite these settings.

1. Open the Administrative console.
2. Select the level that this configuration applies to:
   - For the web container level:
     a. Click **Servers** > **Application Servers**.
     b. Select a server from the list of application servers.
     c. Under Additional Properties, click **Web Container**.
   - For the enterprise application level:
     a. Click **Applications** > **Applications**.
     b. Select an applications from the list of applications.
   - For the Web module level:
     a. Click **Applications** > **Enterprise Applications**.
     b. Select an applications from the list of applications.
     c. Under Related Items, click **Web Modules**.
     d. Select a Web module from the list of Web modules defined for this application.
3. Under **Additional Properties**, click **Session Management**.
4. Make whatever changes you need to manage sessions
5. If you are working on the Web module or application level and want these settings to override the inherited Session Management settings, under **General Properties**, select **Overwrite**.
6. Click **Apply** and **Save**.

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 Java Server 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 encodeURL( ) or encodeRedirectURL( ) 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() and encodeRedirectURL() methods 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 response.encodeURL() or encodeRedirectURL() 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. Internal Http Server of WebSphere 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.

Configuring session tracking

To configure session tracking, complete the following:

1. Go to the appropriate level of Session Management.
2. Specify which session tracking mechanism you want to pass the session ID between the browser and the servlet:
   - To track sessions with cookies, click **Enable Cookies**.

     To change the cookie settings, click **Modify**.
   - To track sessions with URL rewriting, click **Enable URL Rewriting**.

     If you want to enable protocol switch rewriting, click **Enable protocol switch rewriting.**
   - To track sessions with SSL information, click **Enable SSL ID tracking**.
3. Click **Apply**.
4. Click **Save**.
5. Define the session recovery characteristics.

Serializing access to session data

The Servlet API supports concurrent access to a session in a given server instance. WebSphere Application Server provides an option to prevent the concurrent access to a session in a given server instance so that concurrent modification of a session does not occur in a given server instance. This prevention is achieved by synchronizing the requests based on session. When this feature is turned on, a session is obtained for the request before invoking the servlet and requests are synchronized by locking the session for the servlet execution time. Note that synchronization is based on the memory copy of session. So this feature cannot serialize requests across servers based on session when session affinity fails.

Restriction: Use this feature only when concurrent modification of the same session data is possible and is not desirable by the application. This feature has overhead of serializing the requests based on a session.

Do the following to synchronize session access:

1. Select the level of Session Management on which you want to serialize session access.
2. Under Serialize Session access, click **Allow serial access**.
3. In the Maximum wait time box, type the amount of time, in milliseconds, a servlet waits on a session before continuing execution. The default is 120000 milliseconds or two minutes.
4. Select **Allow access on timeout** if you want the servlet to gain access to the session and continue normal execution even if the session is still locked by another servlet. If you do not select this box, the servlet execution will abort when the session request times out.
5. Click **Apply**.
6. Click **Save**.

Configuring session tracking for Wireless Application Protocol (WAP) devices

Most Wireless Application Protocol (WAP) devices do not support cookies. The preferred way to track sessions for WAP devices is to use URL rewriting. However on most WAP devices, the maximum allowed URL length is 128 characters. With URL rewriting, a session indentifier is added to the URL itself, effectively decreasing the space available for the actual URL and the number of parameters that can be sent on a request.

To reduce the length of session identifier, you can configure key (jsessionid), session ID length and clone ID. To make these configuration changes, complete the following:

1. Open the Administrative console.
2. Click **Servers** > **Application Servers**.
3. Select a server from the list of application servers.
4. Under Additional Properties, click **Web Container**
5. Under Additional Properties, click **Custom Properties**.
6. Add the appropriate properties from the following list:
   - HttpSessionIdLength
   - SessionRewriteIdentifier
   - HttpSessionCloneId
   - CloneSeparatorChange
   - NoAdditionalSessionInfo
   - SessionIdentifierMaxLength
7. Click **Apply** and **Save**.

Session management custom properties

Custom properties for session management:

CloneSeparatorChange
Use this property to maintain session affinity. The clone ID of the server is appended to session identifier separated by colon. On some Wireless Application Protocol (WAP) devices, a colon is not allowed. Set this property to ″true″ to change clone separator to a plus sign (+).

HttpSessionCloneId
Use this property to change the clone ID of the cluster member. Within a cluster, this name must be unique to maintain session affinity. When set, this name overwrites the default name generated by WebSphere Application Server. Default clone ID length: 8 or 9.

HttpSessionIdLength
Use this property to configure the session identifier length. Do not use an extremely low value; using a low value results in reduced number of combinations possible, thereby increasing risk of guessing the session identifier. In a cluster, all cluster members should be configured with same ID length. Allowed range: 8 to 128. Default length: 23.

HttpSessionReaperPollInterval
Use this property to set a wake-up interval for the process that removes invalid sessions. Default is based on maximum inactive interval set in Session Management. Allowed value: integer.

NoAdditionalSessionInfo
Set this value to ″true″ to force removal of information that is not needed in session identifiers. In WebSphere Application Server base edition,a clone ID of -1 is never used; therefore, a clone ID is not included in base edition when this is set. Also, cache ID is not used with nonpersistent sessions; so the cache ID is not included with nonpersistent sessions when this value is set.

NoAffinitySwitchBack
Set this property to ″true″ to maintain affinity to the new member even after original one comes back up. When a cluster member fails, its requests routed to a different cluster member, and sessions are activated in that other member. Thus, session affinity is maintained to the new

member, and when failed cluster member comes back up, the requests for sessions that were created in the original cluster member are routed back to it. Allowed values, true or false. Default: false.

It is recommended that you set this property to ″true″ when distributed sessions with time-based write is configured. Note that this property has no affect on the behavior when distributed sessions is not enabled.

SessionIdentifierMaxLength
Use this value to set maximum length that a session identifier can grow. In a cluster, because of fail-over when a request goes to new cluster member, Session Management appends a new clone ID to the existing clone ID. In a large cluster, if for some reason servers are failing more often, then it is possible that the session identifier length can be more than expected reducing room for URL. So this property helps to find out the condition and take appropriate action to address servers fail-over. When this is specified, message is logged when specified maximum length is reached. Allowed value: integer.

SessionRewriteIdentifier
Use this property to change the key used with URL rewriting. Default key: jsessionid.

Distributed sessions

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

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

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

Session recovery support

For session recovery support, WebSphere Application Server provides distributed session support in the form of database sessions and memory-to-memory replication. Use session recovery support under the following conditions:

- When the user’s session data must be maintained across a server restart
- When the user’s session data is too valuable to lose through an unexpected server failure

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

Distributed Environment settings

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

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management** > **Distributed Environment Settings**.

Distributed Sessions

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

None	Specifies that the session management facility discards the session data when the server shuts down.
Database	Specifies that the session management facility stores session information in the data source specified by the data source connection settings. Click **Database** to change these data source settings.
Memory to Memory Replication	Specifies that the session management facility stores the session information in a data source in memory. The session information is copied to other session management facilities for failure recovery. Click **Memory to Memory Replication** to specify the replicator to use and to change these memory to memory settings. *(For WebSphere Application Server Network Deployment only.)*

Configuring for database session persistence

To configure the session management facility for database session persistence, complete the following:

1. Define a JDBC provider.
2. Create a data source pointing to an existing database, using the JDBC provider that you defined. The data source should be non-JTA, for example, non-XA enabled. Note the JNDI name of the data source. Under **Data Sources >** *datasource_name* **> Custom Properties**, make sure the correct database is entered for the value of the **databaseName** property. If necessary, contact your database administrator to verify the correct database name.
3. Go to the appropriate level of Session Management.
4. Click **Distributed Environment Settings**
5. Select and click **Database**.
6. Specify the Data Source JNDI name from step 2.
7. Specify the database user ID and password for accessing the database.
8. Retype the password for confirmation.
9. Configure a table space and page sizes for DB2 session databases.
10. Switch to a multirow schema.
11. Click **OK**.
12. If you want to change the tuning parameters, click **Custom Tuning Parameters** and select a setting or customize one.
13. Click **Apply**.
14. Click **Save**.

Switching to a multirow schema

By default, a single session maps to a single row in the database table used to hold sessions. With this setup, there are hard limits to the amount of user-defined, application-specific data that WebSphere Application Server can access.

1. Modify the Session Management facility properties to switch from single to multirow schema.
2. Manually drop the database table or delete all the rows in the database table that the product uses to maintain HttpSession objects.

   To drop the table:
   a. Determine which data source configuration Session Management is using.
   b. In the data source configuration, look up the database name.
   c. Use the database facilities to connect to the database.
   d. Drop the SESSIONS table.

Configuring tablespace and page sizes for DB2 session databases

If you are using DB2 for session persistence, you can increase the page size to optimize performance for writing large amounts of data to the database. Page sizes of 8K, 16K, and 32K are supported.

To use a page size other than the default (4K), do the following:

1. If the SESSIONS table already exists, drop it from the DB2 database.
2. Create a new DB2 buffer pool and table space, specifying the same page size (8K, 16K or 32K) for both, and assign the new buffer pool to this table space.

   ```
   DB2 Connect to session
   DB2 CREATE BUFFERPOOL sessionBP SIZE 1000 PAGESIZE 8K
   DB2 Connect reset
   DB2 Connect to session
   DB2 CREATE TABLESPACE sessionTS PAGESIZE 8K MANAGED BY SYSTEM
       USING ('D:\DB2\NODE0000\SQL00005\sessionTS.0') BUFFERPOOL sessionBP
   DB2 Connect reset
   ```

   Refer to DB2 product documentation for details.
3. Configure the correct table space name and page size in the Session Management facility. Page size is referred to as *row size* on the Session Management page.)

When the product is restarted, the Session Management facility creates the new SESSIONS table in the specified tablespace based on the indicated page size.

Multirow schema considerations

IBM WebSphere Application Server supports the use of a multirow schema option in which each piece of application specific data is stored in a separate row of the database. With this setup, the total amount of data you can place in a session is now bound only by the database capacities. The only practical limit that remains is the size of the session attribute object.

The multirow schema potentially has performance benefits in certain usage scenarios, such as when larger amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet processing of an HTTP request. In such a scenario, avoiding unneeded Java object serialization is beneficial to performance.

Understand that switching between multirow and single row is not a trivial proposition.

In addition to allowing larger session records, using multirow schema can yield performance benefits. However, it requires a little work to switch from single-row to multirow schema, as shown in the instructions below.

Coding considerations and test environment

Consider configuring direct single-row usage to one database and multirow usage to another database while you verify which option suits your application needs. (Do this in code by switching the data source used; then monitor performance.)

Programming issue	Application scenario
Reasons to use single-row	• You can read or write all values with just one record read and write. • This takes up less space in a database because you are guaranteed that each session is only one record long.
Reasons **not** to use single-row	2-megabyte limit of stored data per session.

Programming issue	Application scenario
Reasons to use multirow	• The application can store an unlimited amount of data; that is, you are limited only by the size of the database and a 2-megabyte-per-record limit. • The application can read individual fields instead of the whole record. When large amounts of data are stored in the session but only small amounts are specifically accessed during servlet processing of an HTTP request, multirow sessions can improve performance by avoiding unneeded Java object serialization.
Reasons **not** to use multirow	If data is small in size, you probably do not want the extra overhead of multiple row reads when you can store everything in one row.

In the case of multirow usage, design your application data objects not to have references to each other, to prevent circular references. For example, suppose you are storing two objects A and B in the session using HttpSession.put(..) method, and A contains a reference to B. In the multirow case, because objects are stored in different rows of the database, when objects A and B are retrieved later, the object graph between A and B is different than stored. A and B behave as independent objects.

Tuning session management

IBM 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)

Configuring scheduled invalidation

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

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

Configuring write contents

In Session Management, you can configure which session data is written to the database or to another WebSphere instance, depending on whether you are using database pesistent sessions or memory to memory replication. This flexibility allows for fewer code changes for the JSP writer when the application will be operating in a clustered environment. The following options are available in Session Management for tuning what is to be written back:
- Write changed (the default) - Write only session data properties that have been updated through setAttribute() and removeAttribute() method calls.
- Write all - Write all session data properties.

The **Write all** setting might benefit servlet and JSP writers who change Java objects' states that reside as attributes in HttpSession and do not call HttpSession.setAttribute().

However, the use of **Write all** could result in more data being written back than is necessary. If this situation applies to you, consider combining the use of **Write all** with **Time-based write** to boost performance overall. As always, be sure to evaluate the advantages and disadvantages for your installation.

With either Write Contents setting, when a session is first created, complete session information is written, including all of the objects bound to the session. When using database session persistence, in subsequent session requests, what is written to the database depends on whether a single-row or multirow schema has been set for the session database, as follows:

Write Contents setting	Behavior with single-row schema	Behavior with multirow schema
Write changed	If any session attribute is updated, all objects bound to the session are written.	Only the session data modified through setAttribute() or removeAttribute() calls is written.
Write all	All bound session attributes are written.	All session attributes that currently reside in the cache are written. If the session has never left the cache, all session attributes are written.

1. Go to the appropriate level of Session Management.
2. Click Distributed Environment Settings
3. Click Custom Tuning Parameters.
4. Select Custom Settings, and click Modify.
5. Select the appropriate write contents setting.

Configuring write frequency

In the Session Management facility, you can configure the frequency for writing session data to the database or to a WebSphere instance, depending on whether you use database distributed sessions or memory-to-memory replication. This flexibility enables you to weigh session performance gains against varying degrees of failover support. The following options are available in the Session Management facility for tuning write frequency:
- **End of service servlet**- Write session data at the end of the servlet service() method call.
- **Manual update**- Write session data only when the servlet calls the IBMSession.sync() method.
- **Time based** (the default) - Write session data at periodic intervals, in seconds (called the *write interval*).

When a session is first created, session information is always written at the end of the service() call.

Using the time based write or manual update options can result in loss of data in failover scenarios since the backup copy of the session in the persistent store (for example, a database or another JVM) may not be in sync with the session in the session cache.

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 nondistributed 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.

Controlling write operations

You can manually control when modified session data is written out to the database or to another WebSphere Application Server instance by using the sync() method in the com.ibm.websphere.servlet.session.IBMSession interface, which extends the javax.servlet.http.HttpSession interface. By calling the sync() method from the service() method of a servlet, you send any changes in the session to the external location. When *manual update*is selected as the write frequency mode, session data changes are written to an external location only if the application calls the sync() method. If the sync() method is not called, session data changes are lost when a session object leaves the server cache. When *end of service servlet* or *time based* is the write frequency mode, the session data changes are written out whenever the sync() method is called. If the sync() method is not called, changes are written out at the end of service method or on a time interval basis based on the write frequency mode selected.

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

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

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 transactionally 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 classpath (the Application Classpath if utilizing sharing across Web modules in an enterprise application, or the WebModule Classpath 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 will read the cookie data (or encoded URL) from the browser and helps direct the request to the appropriate application or clone based on the assigned session key. This functionality increases use of the in-memory cache and reduces hits to the database or another WebSphere Application Server instance
- **Maximize use of session affinity and avoid breaking affinity.**

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

  You can improve performance by not breaking session affinity. Some suggestions to help avoid breaking session affinity are:
  - Combine all Web applications into a single application server instance, if possible, and use modeling or cloning to provide failover support.

- Create the session for the frame page, but do not create sessions for the pages within the frame when using multiframe 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, JSPs get a HTTPSession using `request.getSession(true)` method. So by default JSPs 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 will 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 unsecure page. When a request from an unsecure page occurs, access is denied and an `UnauthorizedSessionRequestException` error is thrown. (`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 method called sync(). 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, define replicators only on the servers and have the client attach to server replicator.
  - 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 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
- 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 2.3 API specification download site
- J2EE 1.3 specification download site

Chapter 4. Using enterprise beans in applications

1. Design a J2EE application and the enterprise beans that it needs. See ″Resources for learning″ for links to design information that is specific to enterprise beans.
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 using theAssembly Toolkit into one or more EJB modules. This includes setting security.
5. Assemble the modules into a J2EE application using the Assembly Toolkit .
6. 5.1 + 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 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 and deployment problems.
9. Assemble the production application using theAssembly Toolkit.
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 administrative console settings.
    b. Manage other aspects of the J2EE application.
12. Update the module and redeploy it using theAssembly Toolkit.
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. Entity beans with container-managed persistence (CMP) require connections to a form of persistent storage. This storage might be a database, an existing legacy application, a file, or other types of persistent storage. Entity beans with bean-managed persistence manage permanent data in whichever manner is defined in the bean code. This can include writing to databases or XML files, for example.

Session beans do not require database access, although they can obtain it indirectly as needed through entity beans. Session beans can also obtain direct access to databases (and other resources) through the use of resource references. Session beans can be either *stateful* or *stateless*.

New in the Enterprise JavaBeans (EJB) specification, version 2.0, 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.

© Copyright IBM Corp. 2003

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.″

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 2.0 session bean requires a bean class, a home or local home interface, and a remote or local interface. An EJB 2.0 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.
   - Available only through EJB 2.0, 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 Assembly Tool automatically generates SQL code for creating database tables for CMP entity beans. If your CMP beans require complex database mappings, it is recommended that you use the IBM WebSphere Studio 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 2.0 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 `*Name*BeanFinderHelper`, where *Name* is the name of the enterprise bean (for example, AccountBeanFinderHelper).
  - The logic must be contained in a String constant named `*findMethodName* WhereClause`, where *findMethodName* is the name of the finder method. The String constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method's arguments when that method is called.

5.1 + Assemble the beans in one or more EJB modules.

Migrating enterprise bean code to the supported specification

Support for Version 2.0 of the Enterprise JavaBeans (EJB) specification is new for Version 5 of this product. Migration of enterprise beans deployed in Version 4.0.x of this product is not generally necessary; Version 1.1 of the EJB specification is 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 2.0 of the EJB specification is not required for redeployment on this version of the product. However, if your application requires the capabilities of Version 2.0, migrate your Version 1.1-compliant code.

   Note: The EJB Version 2.0 specification mandates that prior to the EJB container's executing a findBy*Method* query, the state of all enterprise beans enlisted in the current transaction be synchronized with the persistent store. (This is so the query is performed against current data.) If Version 1.1 beans are reassembled into an EJB 2.0-compliant module, the EJB container synchronizes the state of Version 1.1 beans as well as that of Version 2.0 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. Modify enterprise bean code for changes in deployment requirements. If the enterprise beans were previously deployed in Version 3.0.x of this product, modify import statements to match standard package names. In Version 3.0.2.x, the following standard packages were present under nonstandard names:

   ```
   javax.sql.*
   javax.transaction.*
   ```

   Any code using WebSphere data sources, including BMP entity beans and session beans that access databases, must be modified.
3. You might have to modify code for some EJB 1.1-compliant modules that were not migrated to Version 2.0. 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.
   - If the CMP beans write to databases with mixed-case table or column names and you used IBM VisualAge for Java, Version 3.5.x, to generate the original deployment code, you cannot simply reassemble the beans in this product. You must export the original EJB project from the VisualAge

for Java product as an EJB 1.1 JAR. This preserves the metadata needed to generate the correct deployment code for mixed-case database tables and columns. For more information, see the documentation for the Deployment Tool for Enterprise JavaBeans.

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

4. Reassemble and redeploy all modules to incorporate migrated code.

Migrating enterprise bean code from Version 1.0 to Version 1.1

The following information generally applies to any enterprise bean that currently complies with Version 1.0 of the Enterprise JavaBeans (EJB) specification. For more information about migrating code for beans produced with the IBM WebSphere Studio Application Developer tool, 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 2.0

Enterprise JavaBeans (EJB) Version 2.0-compliant beans may be assembled only in an EJB 2.0-compliant module, although an EJB 2.0-compliant module can contain a mixture of Version 1.x and Version 2.0 beans.

The EJB Version 2.0 specification mandates that prior to the EJB container's executing a findBy*Method* query, the state of all enterprise beans enlisted in the current transaction be synchronized with the persistent store. (This is so the query is performed against current data.) If Version 1.1 beans are reassembled into an EJB 2.0-compliant module, the EJB container synchronizes the state of Version 1.1 beans as well as that of Version 2.0 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 IBM WebSphere Studio Application Developer tool, see the documentation for that product. For more information about migrating code in general, see ″Resources for learning.″

1. In beans with container-managed persistence (CMP) version 1.x, replace each CMP field with abstract get and set methods. In doing so, you must make each bean class abstract.
2. In beans with CMP version 1.x, change all occurrences of `this.`*`field`*` = `*`value`* to `set`*`Field`*`(`*`value`*`)`.
3. In each CMP bean, create abstract get and set methods for the primary key.
4. In beans with CMP version 1.x, create an EJB Query Language statement for each finder method.
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 2.0, 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 2.0, 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.

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 WebSphere Studio Application Developer product.

Optimistic concurrency control for container-managed persistence

This product supports optimistic concurrency control of data access.

Access intents for EJB persistence

This product supports the application of named data-access policies.

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 WebSphere Studio 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

5.1 + 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 2.0 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 2.0 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 version 5.0.2 enables 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.

Setting the run time for batched commands:
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*.

Deferred Create for container managed persistence

The specification for Enterprise Java Beans (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 version 5.0.2 enables 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).

Setting the run time for deferred create:

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*.

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 Toolkit (ATK) 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, superceding 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. So far this is current WebSphere Application Server behavior for long-lifetime beans. Transaction 2 ends.

4. Another application, which does not use the *Department* CMP bean, is executed. This application might or might not be run on the 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 *Deparment* 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 *Department* bean *dept01*.
5. Transaction 3 begins. Application code looks up *Department*'s home and calls a finder method (such as *findByPrimaryKey("dept01")*). Because this is the first finder after *Department dept01* is invalidated, a new database trip is made to obtain the data. Transaction 3 ends.

Persistence Manager cache invalidation API

The PM cache invalidation API is in the form of a JMS message that the client sends to a specially-named JMS topic using a connection from a specifically named JMS *TopicConnectionFactory*. The JMS message must be an *ObjectMessage* created by the client. The client code creates a *PMCacheInvalidationRequest* object that describes the bean data to invalidate. Client code places the *PMCacheInvalidationRequest* object in the *ObjectMessage* and publishes the *ObjectMessage* (for further details on the JMS objects and terms used here, please see the Java Message Service documentation).

The public class *PMCacheInvalidationRequest* is central to the API, so we include a portion of its code here for illustration purposes (if you see any differences between this illustration and the actual class, the class is to be considered correct):

```
packagecom.ibm.websphere.ejbpersistence;

/**
*An instance of this class represents a request to invalidate one or more
*CMP beans in the PMcache.When an invalidate occurs,cached datafor 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 PMcache so applications do not behave falsely
*based on stale data.
*/
public class PMCacheInvalidationRequestimplementsSerializable{

. . .

/**
 * Constructor used to invalidate a single bean
 * @param beanHomeJNDIName the JNDI name of the bean home. This is the same value
 * used to look up the bean home prior to calling findByPrimaryKey, for example.
 * @param beanKey the primary key of the bean to be invalidated. The actual
 * object type must be the primary key type for this bean type.
 */
public PMCacheInvalidationRequest(String beanHomeJNDIName, Object beanKey)
 throws IOException {
. . .
}
/**
 * Constructor used to invalidate a Collection of beans
 * @param beanHomeJNDIName java.lang.String the JNDI name of the bean home.
 * This is the same value used to look up the bean home prior to calling
 * findByPrimaryKey, for example.
 * @param beanKeys a Collection of the primary keys of the beans to be
 * invalidated. The actual type of each object in the Collection must be the
 * primary key type for this bean type.
*/
```

```
 public PMCacheInvalidationRequest(String beanHomeJNDIName, Collection beanKeys)
 throws IOException {
. . .
}
/**
 * Constructor used to invalidate all beans of a given type
 * @param beanHomeJNDIName java.lang.String the JNDI name of the bean home.
 * This is the same value used to look up the bean home prior to calling
 * findByPrimaryKey, for example.
 */
public PMCacheInvalidationRequest(String beanHomeJNDIName) {
 . . .
}

}
```

If the client wants to perform the invalidation in a synchronous way, it can opt to receive an acknowledgement JMS message when the invalidation is complete. To ask for an acknowledgement (ACK) message, the client sets a *Topic* of its own choosing in the *JMSReplyTo* field of the *ObjectMessage* for the invalidation request (see JMS 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 (*TopicConnectionFactory*, *TopicDestination*, and so forth) must be configured by the user (using the Administration console or other method) if they want to use PM Cache Invalidation. In this way the user can chose 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, and use names 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 TopicConnectionFactory
 private static final String topicConnectionFactoryJNDIName =
   "com.ibm.websphere.ejbpersistence.InvalidateTCF";
// The JNDI name of the TopicDestination
  private static final String topicDestinationJNDIName =
   "com.ibm.websphere.ejbpersistence.invalidate";
// The Topic name (part of the TopicDestination)
  private static final String topicString =
   "com.ibm.websphere.ejbpersistence.invalidate";
```

Here are examples of how these constants can be used in client code:

```
// Look up the TopicConnectionFactory...
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 provider, available in WebSphere Application Server as the **Embedded Messaging** feature (selectable during installation), supports the use of MQ classes (or structures in other languages) to create a topic connection and topic that are compatible with the *TopicConnectionFactory* and *TopicDestination* you configure using WebSphere Application Server Application 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 Java Bean Version 2.0, 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.

Using access intent policies

You can use access intent policies to help the product run-time environment manage various aspects of Enterprise JavaBeans (EJB) persistence. You apply access intent policies to EJB Version 2.0 entity beans and their methods by using an application assembly tool. A set of default access intent policies comes with the Assembly Toolkit.

1. Apply default access intent to CMP entity beans. For more information, see the online help available with the Assembly Toolkit.
2. Apply access intent policies to methods of CMP entity beans.

Access intent policies

An access intent policy is a named set of properties (access intents) that governs data access for Enterprise JavaBeans (EJB) persistence. You can assign policies to an entity bean and to individual methods on an entity bean’s home, remote, or local interfaces during assembly. Access intents are settable 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 2.0 enterprise beans.

Access intent policies configured on an entity basis define the default access intent for that entity. The default access intent is used to control that entity in the absence of a more specific configuration based upon either method-level policy configuration or application profiling.

Access intent can be controlled in a more precise way by using either application profiling or by using method-level access intent policies. Application profiling is only available in the WebSphere Application Server Enterprise product. 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 WebSphere Studio Application Developer product, this product provides support for selecting a subset of the nonnullable columns that are to be reflected in the overqualified update statement that is generated in the deployment code to support optimistic policies.

An entity that is configured with a read-only policy that causes a bean to be activated can cause problems if updates are attempted within the same transaction. Those changes will not be committed, and an exception will be thrown because data integrity might be compromised.

Concurrency control

Concurrency control is the management of contention for data resources. A concurrency control scheme is considered *pessimistic* when it locks a given resource early in the data-access transaction and does not release it until the transaction is closed. A concurrency control scheme is considered *optimistic* when locks are acquired and released over a very short period of time at the end of a transaction.
The objective of optimistic concurrency is to minimize the time over which a given resource would be unavailable for use by other transactions. This is especially important with long-running transactions, which under a pessimistic scheme would lock up a resource for unacceptably long periods of time.

Under an optimistic scheme, locks are obtained immediately before a read operation and released immediately afterwards. Update locks are obtained immediately before an update operation and held until the end of the transaction.

To enable optimistic concurrency, this product uses an *overqualified update scheme* to test whether the underlying data source has been updated by another transaction since the beginning of the current transaction. With this scheme, the columns marked for update and their original values are added explicitly through a WHERE clause in the UPDATE statement so that the statement fails if the underlying column values have been changed. As a result, this scheme can provide column-level concurrency control; pessimistic schemes can control concurrency at the row level only.

Optimistic schemes typically perform this type of test only at the end of a transaction. If the underlying columns have not been updated since the beginning of the transaction, pending updates to container-managed persistence fields are committed and the locks are released. If locks cannot be acquired or if some other transaction has updated the columns since the beginning of the current transaction, the transaction is rolled back: All work performed within the transaction is lost.

Pessimistic and optimistic concurrency schemes require different transaction isolation levels. Enterprise beans that participate in the same transaction and require different concurrency control schemes cannot operate on the same underlying data connection.

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 roundtrips 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 canonical 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.
Read-ahead hints can be set only using the WebSphere Application Server Enterprise assembly tool or through the Add Access Intent wizard of the IBM WebSphere Studio Application Developer product.

Read-ahead is only supported for access intent policies that can be applied by the backend against which the application is deployed. Otherwise, the read-ahead hint is disregarded.

Currently, only findByPrimaryKey methods can have read-ahead hints. Only beans related to the requested beans by a container-managed relationship (CMR), either directly or indirectly through other beans, can be read ahead. Beans that use EJB inheritance should not be used in a read-ahead hint.

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 Websphere Studio Application Developer product.

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 theAssembly Toolkit.

1. Start theAssembly Toolkit.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, select **File > Open**, then select the EAR file.
3. Select **EJB Modules >** ***moduleName*** **> Access Intent**.

4. To configure a new access intent policy, right-click and select **New**.
5. On the **New Access Intent** panel, specify a name and a description. These attributes are provided as a convenience to the developer and are not used at run time.
6. To select the methods to which the access intent policy should apply, click **Add** beside the Methods table.
7. From the **Applied access intent** list, select an access intent policy.
8. To override an attribute defined in the applied policy, click **Add** beside the Access intent attribute overrides table.
9. Click **OK** to exit the New Access Intent panel.
10. Save your configuration by selecting **File > Save**.

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 FOR UPDATE clause on the SQL SELECT statement to be executed, but a read-only query cannot support FOR UPDATE. 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 in the WebSphere Application Assembly Tool. Some examples of misconfiguration include:

- A method configured with two different access intent policies
- A method configured with an undefined access intent policy

javax.ejb.NoSuchEntityException

If an update fails under optimistic concurrency because fields changed within another transaction between load and store requests, a NoSuchEntityException is raised and the commit fails.

Access intent best practices

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

- **Start by configuring the default access intent policy for an entity.** After your application is built and running, you can more finely tune certain access paths in your application using application profiling or method-level access intent.
- **Don't mix access types.** Avoid using both pessimistic and optimistic policies in the same transaction. For most databases, pessimistic and optimistic policies use different isolation levels. This can result in multiple database connections, which prevents you from taking advantage of the performance benefits possible through connection sharing.
- **Take care when applying `wsPessimisticUpdate-NoCollision`.** This policy does not ensure data integrity. No database locks are held, so concurrent transactions can overwrite each other's updates. Use this policy only if you can be sure that only one transaction will attempt to update persistent store at any given time.

Frequently asked questions: Access intent

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 FOR UPDATE clause. On Oracle databases, however, the weakest lock is an update lock; this means that the SQL query must contain a FOR UPDATE clause. However, not every SQL statement necessarily supports FOR UPDATE; for example, if the query is being run against multiple tables in a join, FOR UPDATE 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 execption 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.

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.

An EJB module can be used as a stand-alone application, or it can be combined 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.″

Assembling EJB modules

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 stand-alone 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 the Assembly Toolkit 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 the Assembly Toolkit.
2. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
4. Migrate enterprise bean (JAR) files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import your enterprise bean files to the Assembly Toolkit.
5. Create a new EJB module.

6. Copy code artifacts (such as entity beans) from one EJB module into a new EJB module.
7. Verify the contents of the new EJB module in either of the following ways:
   - In the J2EE Hierarchy view, expand **EJB Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the EJB module in a Navigator view.

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.

EJB containers

An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean's operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

One or more EJB modules, each containing one or more enterprise beans, can be installed in a single container.

The EJB container provides many services to the enterprise bean, including the following:
- Beginning, committing, and rolling back transactions as necessary.
- Maintaining pools of enterprise bean instances ready for incoming requests and moving these instances between the inactive pools and an active state, ensuring that threading conditions within the bean are satisfied.

- Most importantly, automatically synchronizing data in an entity bean's instance variables with corresponding data items stored in persistent storage.

By dynamically maintaining a set of active bean instances and synchronizing bean state with persistent storage when beans are moved into and out of active state, the container makes it possible for an application to manage many more bean instances than could otherwise simultaneously be held in the application server's memory. In this respect, an EJB container provides services similar to virtual memory within an operating system.

Between transactions, the state of an entity bean can be cached. The EJB container supports option A, B, and C caching.
- With option A caching, the application server assumes that the entity bean is used within a single container. Clients of that bean must direct their requests to the bean instance within that container. The entity bean has exclusive access to the underlying database, which means that the bean cannot be cloned or participate in workload management if option A caching is used.
- With option B caching, the entity bean remains active in the cache throughout the transaction but is reloaded at the start of each method call.
- With option C caching (the default), the entity bean is always reloaded from the database at the beginning of each transaction. A client can attempt to access the bean and start a new transaction on any container that has been configured to host that bean. This is similar to the session clustering facility described for HTTP sessions in that the entity bean's state is maintained in a shared database that can be accessed from any server when required.

This product supports the cloning of stateful session bean home objects among multiple application servers. However, it does not support the cloning of a specific instance of a stateful session bean. Each instance of a particular stateful session bean can exist in just one application server and can be accessed only by directing requests to that particular application server. State information for a stateful session bean cannot be maintained across multiple members of a server cluster.

For more information about EJB containers, see ″Resources for learning.″

Managing EJB containers

Each application server can have a single EJB container; one is created automatically for you when the application server is created. The following steps are to be performed only as needed to improve performance after the EJB application has been deployed.

1. Adjust EJB container settings.
2. Adjust EJB cache settings.

If adjustments do not improve performance, consider adjusting access intent policies for entity beans, reassembling the module, and redeploying the module in the application.

EJB container system properties

In addition to the settings accessible from the administrative console, you can set the following system property by command-line scripting:

com.ibm.websphere.ejbcontainer.poolSize

Specifies the size of the pool for the specified bean type. This property applies to stateless, message-driven and entity beans. If you do not specify a default value, the container defaults of 50 and 500 are used.

Set the pool size for a given entity bean as follows:

```
beantype=min,max[:beantype=min,max...]
```

beantype is the J2EE name of the bean, formed by concatenating the application name, the **#** character, the module name, the **#** character, and the name of the bean (that is, the string assigned to the `<ejb-name>` field in the bean's deployment descriptor). *min* and *max* are the minimum and maximum pool sizes, respectively, for that bean type. Do not specify the square brackets shown in the previous prototype; they denote optional additional bean types that you can specify after the first. Each bean-type specification is delimited by a colon (:).

Use an asterisk (*) as the value of *beantype* to indicate that all bean types are to use those values unless overridden by an exact bean-type specification somewhere else in the string, as follows:

```
*=30,100
```

To specify that a default value be used, omit either *min* or *max* but retain the comma (,) between the two values, as follows (split for publication):

```
SMApp#PerfModule#TunerBean=54,
   :SMApp#SMModule#TypeBean=100,200
```

You can specify the bean types in any order within the string.

Container interoperability

Container interoperability describes the ability of WebSphere Application Server clients and servers at different versions to successfully negotiate differences in native Enterprise JavaBeans (EJB) Version 1.1 finder methods support and Java 2 Platform, Enterprise Edition (J2EE) Version 1.3 compliance.

At one time, there were significant interoperability problems among WebSphere Application Server, versions 4.0.x and 3.5.x distributed, and Version 4.0.x for zSeries. The introduction of interoperable versions of some class types solved these problems for distributed versions 3.5.6, 4.0.3, and 5 as well as for zSeries Version 4.0.x.

Older 4.0.x and 3.5.x client and application server versions do not support the interoperability classes, which makes them uninteroperable with versions that use the classes. The system property *com.ibm.websphere.container.portable* remedies this situation by enabling newer versions of the application server to turn off the interoperability classes. This lets a more recent application server return class types that are interoperable with an older client.

Depending on the value of com.ibm.websphere.container.portable, application servers at versions 5, 4.0.3 and later, and 3.5.6 and later, return different classes for the following:

- Enumerations and collections returned by EJB 1.1 finder methods
- EJBMetaData
- Handles to:
  - Entity beans
  - Session beans
  - Home interfaces

If the property is set to `false`, application servers return the old class types, to enable interoperability with versions 3.5.5 and earlier, and 4.0.2 and earlier. If the property is set to `true`, application servers return the new classes.

Instructions for setting the com.ibm.websphere.container.portable property are in the release notes for versions 3.5.6 and later, and 4.0.3 and later. The following tables show interoperability characteristics for various version combinations of application servers and clients as well as default property values for each combination.

Interoperability of Version 3.5.x client with Version 5 application server

Clients at Version 3.5.5 and earlier are not interoperable with Version 5 servers when using:

- EJBMetaData
- Enumerations returned by EJB 1.x finder methods

- Handles to entity beans

  If you would like to use updated Handle classes in EJB 2.x-compliant beans but have one of the older clients (versions 3.5.5 and earlier) installed, set the system property com.ibm.websphere.container.portable.finder to `false`. With this setting in place, the Version 5 application server uses the updated handles but returns the enumerations and collections that were used in the earlier clients.

To interoperate with Version 5 application servers, you must upgrade all Version 3.5.x clients to Version 3.5.6 or later.

Interoperability of Version 5 client with Version 3.5.x application server

Client at Version 5, using this function	Application server at Version 3.5.6, property true	Application server at Version 3.5.6, property false (default)	Application server at Version 3.5.5 and earlier
EJBMetaData	Does not work across domains	Works	Does not work
Handle to session bean	Works	Works	Does not work
Handle to entity bean	Does not work across domains	Does not work across domains	Does not work across domains
Enumeration returned by EJB 1.x finder method	Works	Works	Works

Interoperability of Version 4.0.x client with Version 5 application server

Ideally, all 4.0.x clients that use Version 5 application servers should be at Version 4.0.3 or later.

Version 5 application servers return the interoperability class types by default (`true`). This can cause interoperability problems for distributed clients at versions 4.0.1 or 4.0.2. In particular, problems can occur with collections and enumerations returned by EJB 1.1 finder methods.

Although it is strongly discouraged, you can set com.ibm.websphere.container.portable to `false` on a Version 5 application server. This causes the application server to return the old class types, providing interoperability with clients at Version 4.0.2 and earlier. This is discouraged because:
- The Version 5 application server instance would become non-J2EE 1.3 compliant with regard to handles, home interface handles, and EJBMetaData.
- EJB 1.x finder methods return collection and enumeration objects that do not originate from ejbportable.jar.
- Interoperability restrictions still exist with the property set to `false`.
- Version 5 client handles to entity beans and home interfaces do not work across domains for the server you set to `false`.

  If you would like to use updated Handle classes in EJB 2.x-compliant beans but have one of the older clients (versions 4.0.2 and earlier) installed, set the system property com.ibm.websphere.container.portable.finder to `false`. With this setting in place, the Version 5 application server uses the updated handles but returns the enumerations and collections that were used in the earlier clients.

Interoperability of client at Version 4.0.2 and earlier with Version 5 application server

Client at Version 4.0.2 and earlier, using this function	Application server at Version 5, property true (default)	Application server at Version 5, property false
EJBMetaData	Does not work	Works for 4.0.2 client
Handle to session bean	Does not work	Works

Client at Version 4.0.2 and earlier, using this function	Application server at Version 5, property true (default)	Application server at Version 5, property false
Handle to entity bean	Does not work	Does not work across cells
Enumeration returned by EJB 1.x finder method	Does not work	Works
Collection returned by EJB 1.x finder method	Does not work	Works
Handle to home interface	Does not work	Does not work across cells

If you would like to use updated Handle classes in EJB 2.x-compliant beans but have one of the older clients (versions 3.5.5 and earlier, and 4.0.2 and earlier) installed, set the system property com.ibm.websphere.container.portable.finder to `false`. With this setting in place, the Version 5 server uses the new Handle classes but returns the older enumeration and collection classes.

Interoperability of client at Version 4.0.3 and later with Version 5 application server

Clients at Version 4.0.3 and later work well with Version 5 application servers. However, if you set the com.ibm.websphere.container.portable to `false`, client handles to entity beans and home interfaces do not work across domains for the server you set to `false`.

Client at Version 4.0.3 and later, using this function	Application server at Version 5, property true (default)	Application server at Version 5, property false
EJBMetaData	Works	Works
Handle to session bean	Works	Works
Handle to entity bean	Works	Does not work across cells
Enumeration returned by EJB 1.x finder method	Works	Works
Collection returned by EJB 1.x finder method	Works	Works
Handle to home interface	Works	Does not work across cells

Interoperability of Version 5 client with Version 4.0.x application server

Clients at Version 5 work well with Version 4.0.3 application servers if you set com.ibm.websphere.container.portable to `true`. Client handles to entity beans and home interfaces do not work across domains for any Version 4.0.3 server with com.ibm.websphere.container.portable at the default value, `false`. Version 5 client handles to application servers at Version 4.0.2 and earlier also have restrictions.

Client at Version 5, using this function	Application server at Version 4.0.3, property true	Application server at Version 4.0.3, property false (default)	Application server at Version 4.0.2 or earlier
EJBMetaData	Works	Works	Works for 4.0.2 server only
Handle to session bean	Works	Works	Works
Handle to entity bean	Works	Does not work across domains	Does not work across domains
Enumeration returned by EJB 1.x finder method	Works	Works	Works
Collection returned by EJB 1.x finder method	Works	Works	Works

Client at Version 5, using this function	Application server at Version 4.0.3, property true	Application server at Version 4.0.3, property false (default)	Application server at Version 4.0.2 or earlier
Handle to home interface	Works	Does not work across domains	Does not work across domains

Interoperability of zSeries Version 4.0.x client with Version 5 application server

The only valid configuration for container interoperability with zSeries Version 4.0.x clients is the default configuration for the Version 5 application server.

Interoperability of Version 5 client with zSeries Version 4.0.x application server

Version 5 clients should work with a zSeries Version 4.0.x application server with the correct interoperability fixes described in the zSeries documentation. The interoperability characteristics should be the same as for a Version 4.0.3 distributed application server with the property set to `true`.

Client at Version 5, using this function	zSeries application server at Version 4.0.x
EJBMetaData	Works
Handle to session bean	Works
Handle to entity bean	Works
Enumeration returned by EJB 1.x finder method	Works
Collection returned by EJB 1.x finder method	Works
Handle to home interface	Works

Interoperability of the handle formats in WebSphere Application Server, Version 5 and Version 5.0.1

Applications that attempt to persist handles to enterprise beans and **EJBHome** needed to subclass ObjectInputStream in WebSphere Application Server, Version 5. This action was required so that the subclass ObjectInputStream could utilize the context class loader to resolve the classes for enterprise beans and EJBHome stubs.

In addition, handles created and persisted in WebSphere Application Server, Version 5 only work with objects that have an unchanged remote interface. If the remote interface is changed, the handle is no longer valid because the stub is serialized inside the handle and its serial Version UID changes if the remote interface changes.

This release introduces a new handle persistence mechanism that avoids the implementation drawbacks of the previous version. However, if handles are used for this WebSphere Application Server deployment, you should consider the following issues when applying this update, future WebSphere Application Server Fix Packs and EJB Container cumulative fixes for WebSphere Application Server, Version 5.

If a WebSphere Application Server, Version 5 persisted handle or home handle is encountered by a WebSphere Application Server, Version 5.0.1 system, it can be read and utilized. In addition, it will be converted to WebSphere Application Server, Version 5.0.1 format if it is re-persisted. The WebSphere Application Server, Version 5.0.1 format cannot be read by a WebSphere Application Server, Version 5 system unless PQ72184 is applied.

Problems arise when handles are persisted and shared across systems that are not at the WebSphere Application Server, Version 5.0.1 level or later. However, a Version 5 system can receive a handle from Version 5.0.1 remotely through a call to get a handle on an enterprise bean or a getHomeHandle on an

EJBHome. The remote call will succeed, however, any attempt to persist it on the Version 5 system will have the same limitations regarding the use of ObjectInputStream and changes in remote interface invalidating the persisted handle.

When your application stores handles persistently and shares this persistence with multiple clients or application servers, apply WebSphere Application Server, Version 5.0.1 or PQ72184 to both the client and server systems at the same time. Failure to do so can result in the inability of these systems to read the handle data stored by upgraded systems. Also, handles stored by the WebSphere Application Server, Version 5 can force the applications of the updated system to still subclass `ObjectInputStream`. Applications using the WebSphere Application Server Enterprise, Version 5 scheduler and process choreographer, are affected by these changes. These users should update their Version 5 systems at the same time with either Version 5.0.1 or PQ72184.

If the applications store handles in the session context, or locally in a file on the same system, that is not shared by other applications, on different systems, they might be able to update their systems individually, rather than all at once. If Client Container and thin client applications do not share persisted handle data, they can be updated as needed as well. However, handles created and persisted in WebSphere Application Server, Version 5, Version 4.0.3 and later (with the property flag set), or Version 3.5.7 and later (with the property flag set) are not usable if either the home or the remote interface changes.

If any WebSphere Application Server, Version 3.5.7 or Version 4.0.3 and later enables the system property com.ibm.websphere.container.portable to **true**, any handles to objects on that server have the same interoperability limitations. In addition, if any WebSphere Application Server, Version 3.5.7 and later or Version 4.0.3 applications store a handle obtained from a WebSphere Application Server, Version 5 or Version 5.0.1, the same restrictions apply, regarding the need to subclass ObjectInputStream and the usability of handles after a change to the remote interface is made.

Replication of the Http Session and Handles

This note applies to you if you place Handles to Homes or EJBs, or EJB or EJBHome references in the Http Session in your application and you use Http Session Replication. If you intend to replicate a mixed environment of Version 5.0.0 and Version 5.0.1 or 5.0.2 machines you should first apply the latest Version 5.0.0 container cumulative e-fix to the Version 5.0.0 machines before allowing the Version 5.0.1 or 5.0.2 server into the typology. The reason for this is that Version 5.0.0 servers are not able to understand the persisted Handle format used on the Version 5.0.1 and 5.0.2 server. This is similar to the case of Version 5.0.0 and Version 5.0.1 or 5.0.2 systems trying to use a shared database, mentioned above. But in this case, it is the Http Session object and not the database providing the persistence.

Top Down Deployment Mapping

The size of the Handle objects has grown due to the fix put in to allow serialization and deserialization to occur without the previous requirements of subclassing the ObjectInputStream and so on. Top down deployment of an object that contains EJB and EJBHome references create a database table ddl that has a field of 1000 bytes of VARCHAR for BITDATA which will contain the Handle. It might be that your object's Handle does not fit in the 1000 byte default field, and you might need to adjust this to a higher value. You might try increments of 250 bytes, that is, 1250, 1500, and so on.

Deploying EJB 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. Deploy the application.
3. 5.1 + Update the configuration for each EJB module as needed for the deployment environment.

4. For information about the EJB deployment tool, see documentation for the EJB deployment tool.

The next step is to test and debug the module.

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 specification.
- 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
- What's new in the Enterprise JavaBeans 2.0 Specification?

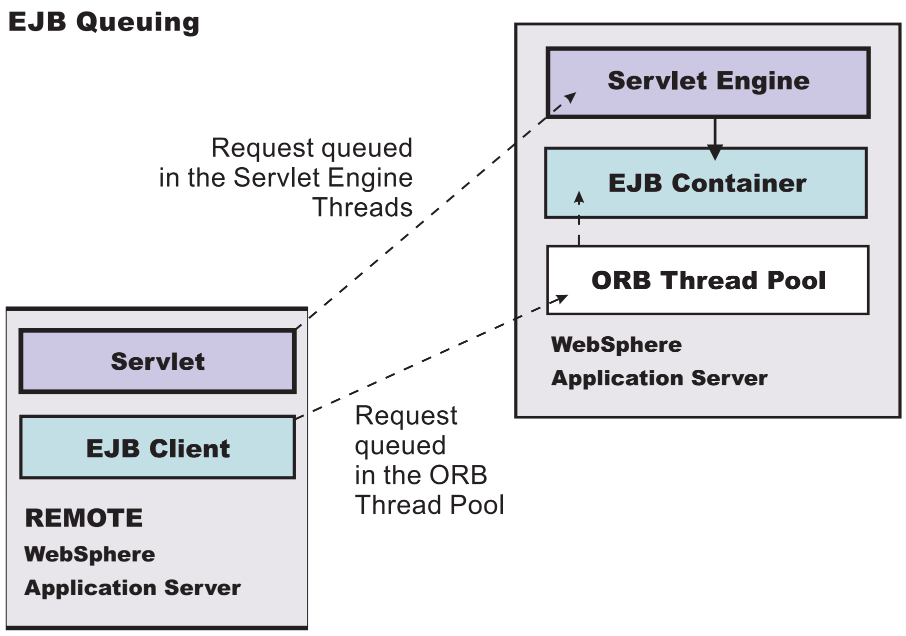
  You can also download the specification itself from this URL.
- JavaTM 2 Platform: Compatibility with Previous Releases

  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.

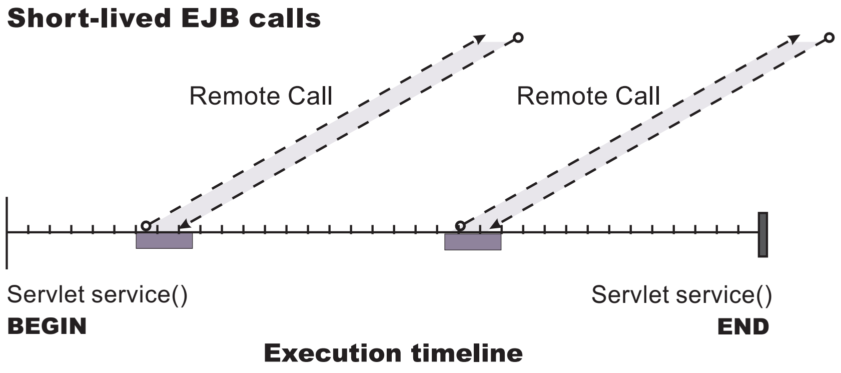
Remote enterprise beans communicate by using the Remote Method Invocation over an Internet Inter-Orb Protocol (RMI-IIOP). Method invocations initiated over RMI-IIOP are processed by a server-side object request broker (ORB). The thread pool acts as a queue for incoming requests. However, if a remote 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.

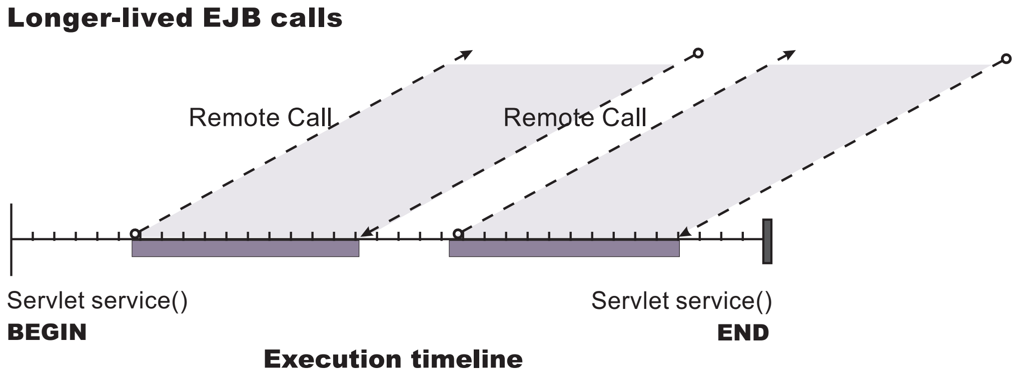

EJB Queuing
Servlet Engine
EJB Container
ORB Thread Pool
WebSphere Application Server
Request queued in the Servlet Engine Threads
Servlet
EJB Client
REMOTE WebSphere Application Server
Request queued in the ORB Thread Pool

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.





The degree to which the ORB thread pool value needs increasing is a function of the number of simultaneous servlets, that is, clients, calling enterprise beans and the duration of each method call. If the method calls are longer or the applications spend a lot of time in the ORB, consider making the ORB thread pool size equal to the Web container size. If the servlet makes only short-lived or quick calls to the ORB, servlets can potentially reuse the same ORB thread. In this case, the ORB thread pool can be small, perhaps even one-half of the thread pool size setting of the Web container.

- **Monitor the percentage of configured threads in use**.

  Tivoli Performance Viewer shows a metric called *percent maxed*, which is used to determine how often the configured threads are used. A value that is consistently in the double-digits, indicates a possible bottleneck a the ORB. Increase the number of threads.

See also Queuing network.

Chapter 5. Using message-driven beans in applications

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interface.

Message-driven beans (a type of enterprise bean defined in the EJB 2.0 specification) extend the base JMS support and the Enterprise JavaBean component model to provide automatic asynchronous messaging. When a message arrives on a destination, a listener passes the message to a new instance of a user-developed message-driven bean for processing.

You can use WebSphere Studio 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-drive beans, see the following topics:
- An overview of message-driven beans
- 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
- Configuring message listener resources for message-driven beans
- Troubleshooting problems with message-driven beans

Message-driven beans - an overview

WebSphere Application Server supports automatic asynchronous messaging with message-driven beans (a type of enterprise bean defined in the EJB 2.0 specification). Messaging with message-driven beans is shown in the figure “Message-driven beans - an overview.”

The support for message-driven beans is based on the message listener service, which comprises a listener manager that controls and monitors one or more listeners. Each listener monitors a JMS destination for incoming messages. When a message arrives on the destination, the listener passes the message to a new instance of a user-developed message-driven bean (an enterprise bean) for processing. The listener then looks for the next message without waiting for the bean to return.

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.

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.

© Copyright IBM Corp. 2003

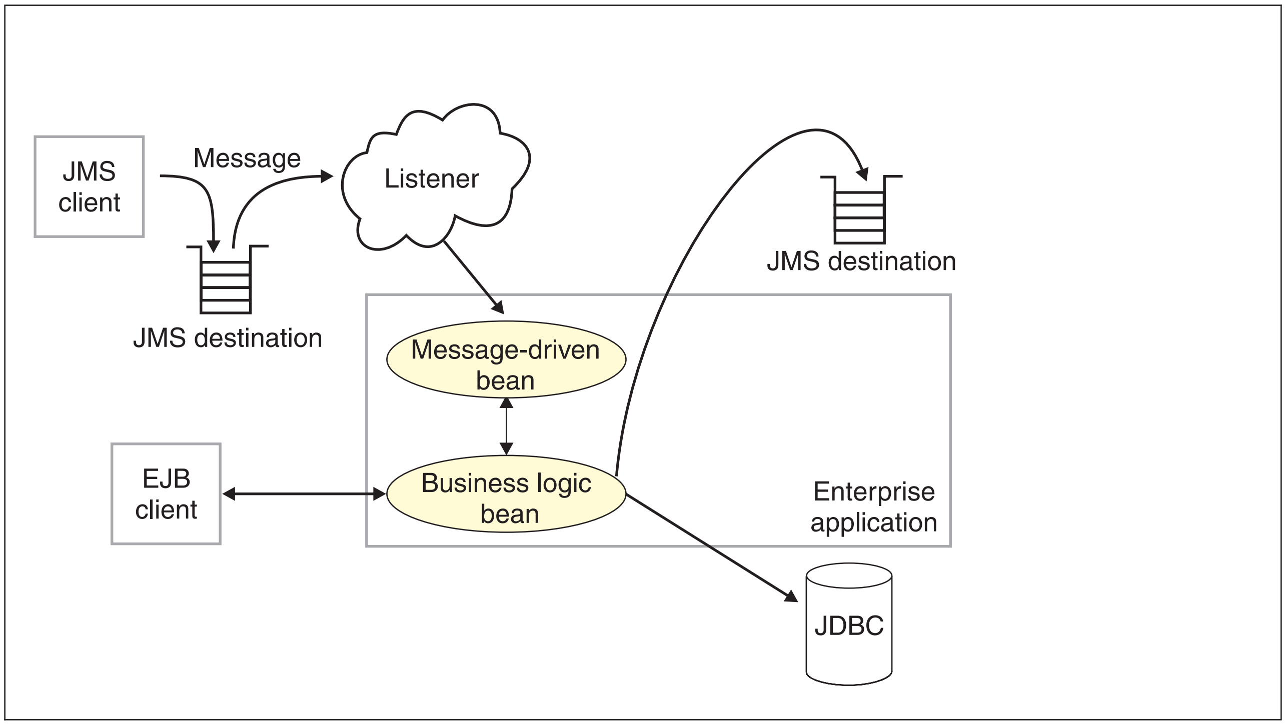


Figure 1. Message-driven beans and the message listener service. This figure shows an incoming message being passed by a JMS listener to a message-driven bean, which passes the message on to a business logic bean for business processing. This messaging is controlled by the listener manager. For more information, see the text that accompanies this figure.

Message-driven beans - components

The WebSphere Application Server support for message-driven beans is based on JMS message listeners and the message listener service, and builds on the base support for JMS. The main components of WebSphere Application Server support for message-driven beans are shown in the following figure and described after the figure:

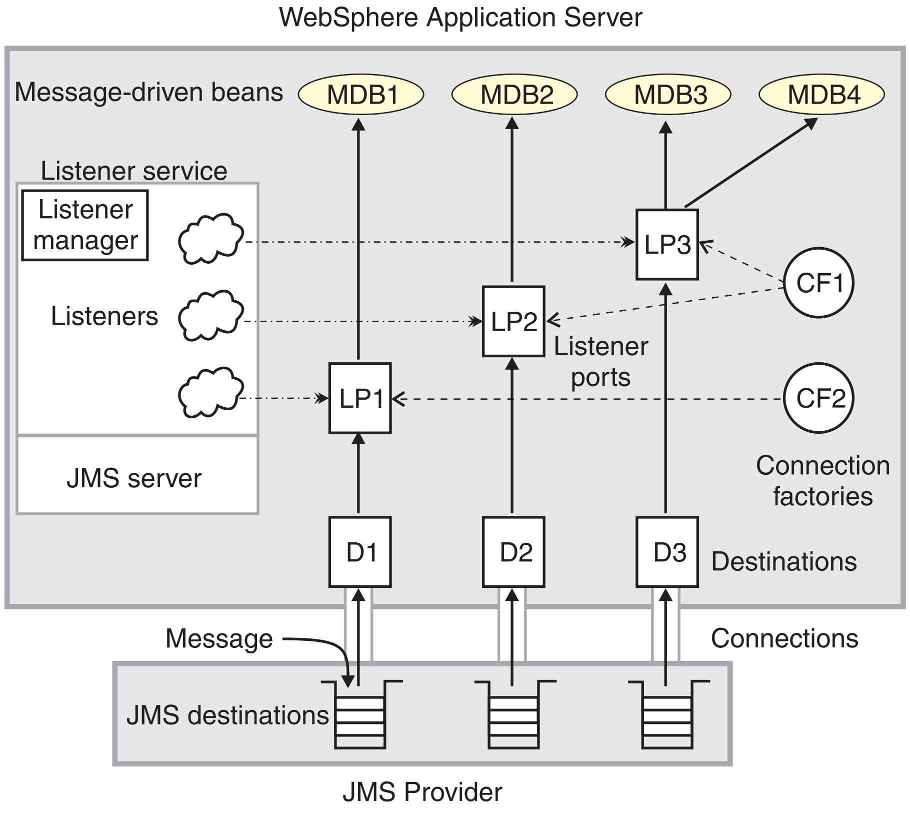


Figure 2. The main components for message-driven beans. This figure shows the main components of WebSphere support for message-driven beans, from JMS provider through a connection to a destination, listener port, then deployed message-driven bean that processes the message retrieved from the destination. Each listener port defines the association between a connection factory, destination, and a deployed message-driven bean. The other main components are the message listener service, which comprises a listener for each listener port, all controlled by the same listener manager. For more information, see the text that accompanies this figure.

The message listener service is an extension to the JMS functions of the JMS provider and provides a listener manager, which controls and monitors one or more JMS listeners.

Each listener monitors either a JMS queue destination (for point-to-point messaging) or a JMS topic destination (for publish/subscribe messaging).

A connection factory is used to create connections with the JMS provider for a specific JMS queue or topic destination. Each connection factory encapsulates the configuration parameters needed to create a connection to a JMS destination.

A listener port defines the association between a connection factory, a destination, and a deployed message-driven bean. Listener ports are used to simplify the administration of the associations between these resources.

When a deployed message-driven bean is installed, it is associated with a listener port and the listener for a destination. When a message arrives on the destination, the listener passes the message to a new instance of a message-driven bean for processing.

When an application server is started, it initializes the listener manager based on the configuration data. The listener manager creates a dynamic session thread pool for use by listeners, creates and starts

listeners, and during server termination controls the cleanup of listener message service resources. Each listener completes several steps for the JMS destination that it is to monitor, including:
- Creating a JMS server session pool, and allocating JMS server sessions and session threads for incoming messages.
- Interfacing with JMS ASF to create JMS connection consumers to listen for incoming messages.
- If specified, starting a transaction and requesting that it is committed (or rolled back) when the EJB method has completed.
- Processing incoming messages by invoking the onMessage() method of the specified enterprise bean.

Message-driven beans - transaction support

Message-driven beans can handle messages read from JMS destinations within the scope of a transaction. If transaction handling is specified for a JMS destination, the JMS listener starts a global transaction *before* it reads any incoming message from that destination. When the message-driven bean processing has finished, the JMS listener commits or rolls back the transaction (using JTA transaction control).

Note:
- 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.

You can configure the **Maximum retries** property of the listener port to define the maximum number of times the listener attempts to read a message from a destination. When the Max retries limit is reached, the listener for that destination is stopped. When you have resolved the problem, you must then restart the listener.

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 JMS 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
Queue connection factory	**Name:** `SamplePtoPQueueConnectionFactory` **JNDI Name:** `Sample/JMS/QCF`
Queue destination	**Name:** `Q1` **JNDI Name:** `Sample/JMS/Q1`
Listener port (for the destination)	**Name:** `SamplePtoPListenerPort` **Connection Factory JNDI Name:** `Sample/JMS/QCF` **Destination JNDI Name:** `Sample/JMS/Q1` **Maximum Sessions:** `5` **Maximum Retries:** `10` **Maximum Messages:** `1`

JMS resource type	Properties
Message-driven bean (deployment properties)	**Name:** `JMSppSampleMDBBean` **Transaction type:** `Container` **Destination type:** `Queue` **Listener port name:** `SamplePtoPListenerPort`
Business logic bean	**Name:** `MyLogicBean`

Ensure that you use consistent values where needed; for example, the JNDI names for the connection factory and destination must be the same for both those resources and the equivalent properties of the listener port.

2. 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.
3. 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.
4. General JMS considerations For Publish/Subscribe messaging, choose the JMS server port to be used depending on your needs for transactions or performance:

   Queued port
   : The TCP/IP port number of the listener port used for all point-to-point and Publish/Subscribe support.

   Direct port
   : The TCP/IP port number of the listener port used for direct TCP/IP connection (non-transactional, non-persistent, and non-durable subscriptions only) for Publish/Subscribe support.

   Note: Message-driven beans cannot use the direct listener port for Publish/Subscribe support. Therefore, any topic connection factory configured with **Port**set to `Direct` cannot be used with message-driven beans.

   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 JMS listener when a message arrives on the input queue 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.

This topic describes how to develop a completely new message-driven bean class. If you have a WAS 4.0 enterprise application that uses the JMS listener, you can migrate that application to use message-driven beans, as described in Migrating a WAS 4.0 JMS listener application to use message-driven beans.

For more information about writing the message-driven bean class, see *Creating a message-driven bean* in the WebSphere Studio help bookshelf.

To develop an enterprise application to use a message-driven bean, complete the following steps:

1. Creating the Enterprise Application project, as described in the WebSphere Studio article .
2. Creating the message-driven bean class.

   You can use the New Enterprise Bean wizard of WebSphere Studio Application Developer to create an enterprise bean with a bean type of Message-driven bean. The wizard creates appropriate methods for the type of bean.

   By convention, the message bean class is named *name*Bean, where *name* is the name you assign to the message bean; for example:

   ```
   public class MyJMSppMDBBean implements MessageDrivenBean, MessageListener
   ```

   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.)
   - 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).

   For example, the following code extract shows how to access the text and the JMS MessageID, from a JMS message of type TextMessage:

```
public void onMessage(javax.jms.Message msg)
{
        String text      = null;
        String messageID = null;

        try
        {
                text = ((TextMessage)msg).getText();

                System.out.println("senderBean.onMessage(), msg text2: "+text);

                //
                // store the message id to use as the Correlator value
                //
                messageID = msg.getJMSMessageID();

                // Call a private method to put the message onto another queue
                putMessage(messageID, text);
        }
        catch  (Exception err)
        {
                err.printStackTrace();
        }
        return;
}
```

Figure 3. Code example: The onMessage() method of a message bean. This figure shows a code extract for a basic onMessage() method of a sample message-driven bean. The method unpacks the incoming text message to extract the text and message identifier and calls a private putMessage method (defined within the same message bean class) to put the message onto another queue.

The result of this step is a message-driven bean that can be assembled into an .EAR file for deployment.

3. Assembling and packaging the application for deployment.

   You can use WebSphere Studio to assemble and package the application for deployment.

The result of this task is an .EAR file, containing an application message-driven bean, 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 the listener ports 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

Migrating a JMS listener application to use message-driven beans

Use this task to migrate an enterprise application that uses message beans with the JMS Listener from WebSphere Application Server 4.0 to use EJB 2.0 message-driven beans.

This task uses a command line utility, `mb2mdb`, that takes as its input either a deployed MessageBean.jar module or a deployed Enterprise Application (.ear) that contains a message bean, along with the JMS listener configuration XML file that defines the WebSphere Application Server 4.0 message beans. The result is a new .jar/.ear module that can then be deployed directly into a WebSphere Application Server 5.0 application server.

You can display the usage help for the migration utility, by typing the command `mb2mdb` at a command line.

To migrate a WebSphere Application Server 4.0 enterprise application that uses message beans to use EJB 2.0 message-driven beans, type the following command at an operating system command line:

```
mb2mdb inputMB.jar-ear jmsListenerConfig.xml workingDirectory outputMDB.jar-ear options
```

Where:

inputMB.jar-ear
: The name of the deployed WebSphere Application Server 4.0 jar or ear file containing a stateless session message bean.

jmsListenerConfig.xml
: The name of the XML configuration file used to configure the WebSphere Application Server 4.0 JMS listeners.

workingDirectory
: The name of a new or existing directory that is used to generate the new message-driven bean and package the outputMDB.jar or .ear file.

  Note: By default, the tool clears the working directory after it has completed. If you want to preserve the contents of the working directory, you must specify the -keep option.

outputMDB.jar-ear
: The name of the output .jar or .ear file for the migrated message-driven bean application.

options
: An optional set of parameters that you can use to control the mb2mdb utility.

  -keep This prevents the tool from clearing out the working directory after completion.

  -verbose
  This causes the tool to display informational messages as to the progress of the migration and its parameters.

  -map listenerHome=bindingHome
  This option provides a mechanism to map between the JNDIHomeName specified for a listener in the JMS listener configuration XML file and the default binding home name specified in the *inputMB.jar-ear* file.

  If the *jmsListenerConfig.xml* file contains a deployed EJB home JNDI name that is different to the default binding within the *inputMB.jar-ear*, use this option to map between the two names.

  This enables you to install the output .jar or .ear file for the message-driven bean into an application server and bind the bean with a different JNDIHomeName than is specified in the bean's bindings.xmi.

The result of this task is a new .jar or .ear file for a message-driven bean that can then be deployed directly into a WebSphere Application Server 5.0 application server.

To successfully install the .jar or .ear file, you need to bind the message-driven bean against a listener port defined to the message listener service of the application server. You need to have used the WebSphere Application Server administrative console to define the listener port, which defines the JMS connection factory and destination that a message-driven bean bound to it listens on. For more information about installing and configuring a .jar or .ear file for a message-driven bean, see Deploying an enterprise application to use message-driven beans.

Deploying an enterprise application to use message-driven beans

Use this task to deploy an enterprise application to use 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 in WebSphere Application Server.

To deploy an enterprise application to use message-driven beans, complete the following steps:

1. Use the WebSphere administrative console to define the listener ports for the application, as described in Adding a new listener port.

2. 5.1 + For each message-driven bean in the application, configure the deployment attributes to match the listener port definitions, as described in Configuring deployment attributes using the Assembly Toolkit.
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 using the Assembly Toolkit

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 the Deployment Descriptor Editor of WebSphere Studio Application Developer or the Assembly Toolkit.

This topic describes the use of the Assembly Toolkit 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 using the Assembly Toolkit, see Assembling applications with the Assembly Toolkit.

To configure the message-driven beans deployment attributes for an enterprise bean, use the Assembly Toolkit to configure the deployment attributes of the application to match the listener port definitions:

1. Start the Assembly Toolkit.
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 Toolkit. 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. Specify the following properties:

      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.

Acknowledge mode
How the session acknowledges any messages it receives.

This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).

Auto Acknowledge
The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

Dups OK Acknowledge
The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

Destination type
Whether the message bean uses a queue or topic destination.

Queue
The message bean uses a queue destination.

Topic The message bean uses a topic destination.

Subscription 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.

6. Specify bindings deployment properties.
   a. 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. Generate code for deployment for EJB modules or for enterprise applications that use EJB modules.
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 Assembly Toolkit to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

Important

Important: Use **Run On Server** for unit testing only. Assembly Server Toolkit controls the WebSphere Application Server installation and, when an application is published remotely, the Toolkit overwrites the server configuration file for that server. Do not use on production servers.

For instructions on remote testing, see the article "Setting Up a Remote WebSphere Application Server in WebSphere Studio V5" at http://www7b.boulder.ibm.com/wsdd/techjournal/0303_yuen/yuen.html.

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 message listener resources for message-driven beans

Use the following tasks to configure resources needed by the message listener service to support message-driven beans.

- Configuring the message listener service
- Adding a new listener port
- Configuring a listener port
- Configuring security for message-driven beans

Configuring the message listener service

Use this task to configure the properties of the message listener service for an application server.

To configure the properties of the message listener service for an application server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. Specify appropriate properties of the message listener service.
5. Optional: Specify any of the following optional properties that you need, as **Custom properties** of the message listener service: NON.ASF.RECEIVE.TIMEOUT, MQJMS.POOLING.TIMEOUT, MQJMS.POOLING.THRESHOLD, MAX.RECOVERY.RETRIES, and RECOVERY.RETRY.INTERVAL.

   For more information about these custom properties, see Custom Properties.
6. Click **OK**.
7. Save your configuration.
8. To have the changed configuration take effect, stop then restart the Application Server.

Message listener service

The message listener service is an extension to the JMS functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

This panel displays links to the Additional Properties pages for Listener Ports, Thread Pool, and Custom Properties for the message listener service.

To view this administrative console page, click **Servers->** ***application_server*-> Message Listener Service**

Name:

The name by which the message listener service is known for administrative purposes.

Data type	String
Default	MsgLService

Description:

A description of the message listener service, for administrative purposes

Data type	String
Default	Null

Thread pool:

Controls the maximum number of threads the Message Listener Service is allowed to run. Select this link to display the service thread pool properties.

Adjust this parameter when multiple message-driven beans are deployed in the same application server and the sum of their maximum session values exceeds the default value of 10.

Data type	Integer
Units	Not applicable
Default	Minimum: 10, maximum 50
Range	Not applicable
Recommended	Set the minimum to the sum of all message-driven beans maximum session values. Set the maximum to anything equal or greater than the minimum.

Custom Properties:

An optional set of name and value pairs for custom properties of the message listener service.

You can use the Custom properties page to define the following properties for use by the message listener service.
- NON.ASF.RECEIVE.TIMEOUT
- MQJMS.POOLING.TIMEOUT
- MQJMS.POOLING.THRESHOLD
- MAX.RECOVERY.RETRIES
- RECOVERY.RETRY.INTERVAL

Message listener service custom properties:

Use this panel to view or change an optional set of name and value pairs for custom properties of the message listener service.

To view this administrative console page, click **Servers->** ***application_server*-> Message Listener Service-> (In content pane, under Additional Properties) Custom Properties**

You can use the Custom properties page to define the following properties for use by the message listener service.

- NON.ASF.RECEIVE.TIMEOUT
- MQJMS.POOLING.TIMEOUT
- MQJMS.POOLING.THRESHOLD
- MAX.RECOVERY.RETRIES
- RECOVERY.RETRY.INTERVAL

NON.ASF.RECEIVE.TIMEOUT:

The timeout in milliseconds for synchronous message receives performed by message-driven bean listener sessions in the non-ASF mode of operation.

You should set this property to a non-zero value only if you want to enable the non-ASF mode of operation for all message-driven bean listeners on the application server.

The message listener service has two modes of operation, Application Server Facilities (ASF) and non-Application Server Facilities (non-ASF).

- The ASF mode is meant to provide concurrency and transactional support for applications. For publish/subscribe message-drive beans, the ASF mode provides better throughput and concurrency, because in the non-ASF mode the listener is single-threaded.
- The non-ASF mode is mainly for use with generic JMS providers that do not support JMS ASF, which is an optional extension to the JMS specification. The non-ASF mode is also transactional but, because the path length is shorter than the ASF mode, usually provides improved performance.

  Use non-ASF if:
  - Your generic JMS provider does not provide JMS ASF support
  - You are using message-driven beans with WebSphere topic connections with the DIRECT port, because the embedded publish/subscribe broker using that port does not support XA transactions or JMS ASF.
  - Message order is a strict requirement

Data type	Integer
Units	Milliseconds
Default	ASF mode (custom property not created)
Range	0 or greater milliseconds **0** non-ASF mode is disabled **1 or more** The timeout in milliseconds for non-ASF message-driven bean listener synchronous session receives
Recommended	If a transaction timeout occurs, the message must recycle causing extra work. If you want to use the non-ASF mode, set this property to lower than the transaction timeout, but leave spare at least the maximum duration of your message-driven bean's onMessage() method. For example, if your message-driven bean's onMessage() method typically takes a maximum of 10 seconds, and the transaction timeout is set to 120 seconds, you might set the NON.ASF.RECEIVE.TIMEOUT property to no more than 110000 (110000 milliseconds, that is 110 seconds).

MQJMS.POOLING.TIMEOUT:

The number of milliseconds after which a connection in the pool is destroyed if it has not been used.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes.

Data type	Integer
Units	Milliseconds
Default	5 minutes
Range	

MQJMS.POOLING.THRESHOLD:

The maximum number of unused connections in the pool.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if there are more than ten unused connections in the pool.

Data type	Integer
Units	Number of connections
Default	10
Range	

MAX.RECOVERY.RETRIES:

The maximum number of times that the listener service tries to get a message from a listener port before the associated listener is stopped, in the range 0 through 2147483647.

Data type	Integer
Units	Retry attempts
Default	0 (no retries)
Range	0 (no retries) through 2147483647

RECOVERY.RETRY.INTERVAL:

The time in seconds between retry attempts by the listener service to get a message from a listener port.

Data type	Integer
Units	Seconds
Default	10
Range	1 through 2147483647

Message listener port collection:

The message listener ports configured in the administrative domain

This panel displays a list of the message listener ports configured in the administrative domain. Each listener port is used with a message-driven bean to automatically receive messages from an associated JMS destination. You can use this panel to add new listener ports or to change the properties of existing listener ports. For more information about the property fields for listener ports, see Listener port properties.

To view this administrative console page, click **Servers->** ***application_server*****-> Message Listener Service-> Listener Ports**

Listener port settings:

A listener port is used to simplify administration of the association between a connection factory, destination, and deployed message-driven bean.

Use this panel to view or change the configuration properties of the selected listener port.

To view this administrative console page, click **Servers->** ***application_server*****-> Message Listener Service-> Listener Ports->** ***listener_port***

Name:

The name by which the listener port is known for administrative purposes.

Data type	String
Default	Null

Initial state:

The state that you want the listener port to have when the application server is next restarted

Data type	Enum
Units	Not applicable
Default	Started
Range	**Started** When the application server is next started, the listener port is started automatically. **Stopped** When the application server is next started, the listener port is not started automatically. If message-driven beans are to use this listener port on the application server, the system administrator must start the port manually or select the Started value of this property then restart the application server.

Description:

A description of the listener port, for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

Connection factory JNDI name:

The JNDI name for the JMS connection factory to be used by the listener port; for example, `jms/connFactory1`.

Data type	String
Default	Null

Destination JNDI name:

The JNDI name for the destination to be used by the listener port; for example, `jms/destn1`.

If the extended messaging service is to use this listener port to handle late responses, the value of this property must match the JMS response destination on the output port used by the sender bean.

You cannot use a temporary destination for late responses.

Data type	String
Default	Null

Maximum sessions:

Specifies the maximum number of concurrent sessions that a listener can have with the JMS server to process messages.

Each session corresponds to a separate listener thread and therefore controls the number of concurrently processed messages. Adjust this parameter when the JMS server does not fully use the available capacity of the machine and if you do not need to process messages in a specific message order.

Data type	Integer
Units	Sessions
Default	1
Range	1 through 2147483647
Recommended	• If you want to process messages in a strict message order, set the value to 1, so only one thread is ever processing messages. • If you want to process multiple messages simultaneously (known as “message concurrency”), set this property to a value greater than 1. Keep this value as low as possible to prevent overloading client applications. A good starting point for a 100% JMS workload with short transaction times is 2 to 4 sessions per processor. If longer running transactions exist, you may need more sessions, which should be determined by experimentation.

Maximum retries:

The maximum number of times that the listener tries to deliver a message before the listener is stopped, in the range 0 through 2147483647.

The maximum number of times that the listener tries to deliver a message to a message-driven bean instance before the listener is stopped.

Data type	Integer
Units	Retry attempts
Default	0 (no retries)
Range	0 (no retries) through 2147483647

Maximum messages:

The maximum number of messages that the listener can process in one session with the JMS server.

For WebSphere embedded messaging or WebSphere MQ as the JMS provider, the listener processes all messages in the session as one batch within the same transaction. For a generic JMS provider, the listener processes each message in the session within a separate transaction.

Data type	Integer
Units	Number of messages
Default	1

Range	1 through 2147483647
Recommended	*For WebSphere embedded messaging or WebSphere MQ as the JMS provider*, if you want to process multiple messages in a single transaction, then set this value to more than 1. This enables multiple messages to be batch-processed into a single transaction, and eliminates much of the overhead of transactions on JMS messages. **CAUTION:** • If one message in the batch fails processing with an exception, the entire batch of messages is put back on the queue for processing. • Any resource lock held by any of the interactions for the individual messages are held for the duration of the entire batch. • Depending on the amount of processing that messages need, and if XA transactions are being used, setting a value greater than 1 can cause the transaction to time out. If an XA transaction does time out routinely because processing multiple messages exceeds the transaction timeout, reduce this property to 1 (to limit processing to one message per transaction) or increase your transaction timeout.

Adding a new listener port

Use this task to add a new listener port to the message listener service, so that message-driven beans can be associated with the port to retrieve messages.

To add a new listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, click **New**.
6. Specify appropriate properties for the listener port.
7. Click **OK**.
8. To save your configuration, click **Save** on the task bar of the Administrative console window.
9. To have the changed configuration take effect, stop then restart the application server.

If enabled, the listener port is started automatically when a message-driven bean associated with that port is installed.

Configuring a listener port

Use this task to change the properties of an existing listener port, used by message-driven beans associated with the port to retrieve messages.

To configure the properties of a listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.

2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, click **Listener Ports**. This displays a list of the listener ports.
5. Click the listener port that you want to modify. This displays the properties of the listener port in the content pane.
6. Specify appropriate properties for the listener port.
7. Click **OK**.
8. To save your configuration, click **Save** on the task bar of the Administrative console window.
9. To have the changed configuration take effect, stop then restart the application server.

Deleting a listener port

Use this task to delete a listener port from the message listener service, to prevent message-driven beans associated with the port from retrieving messages.

To delete a listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, select the checkbox for the listener port that you want to delete.
6. Click **Delete**. This action stops the port (needed to allow the port to be deleted) then deletes the port.
7. To save your configuration, click **Save** on the task bar of the Administrative console window.
8. To have the changed configuration take effect, stop then restart the application server.

Configuring security for message-driven beans

Use this task to configure resource security and security permissions for message-driven beans.

Messages arriving at a listener port have no client credentials associated with them. The messages are anonymous.

To call secure enterprise beans from a message-driven bean, the message-driven bean 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.

JMS 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 a J2C container-managed alias on the JMS connection factory definition that the MDB is using to listen upon (defined by the **Connection factory JNDI name** property of the listener port). If defined, the listener uses the container-managed authentication alias for its JMSConnection security credentials instead of any application-managed alias. To set the container-managed alias, use the administrative console to complete the following steps:

1. To display the listener port settings, click **Servers->** ***application_server*****-> Message Listener Service-> Listener Ports->** ***listener_port***

2. To get the name of the JMS connection factory, look at the **Connection factory JNDI name** property.
3. Display the JMS connection factory properties. For example, to display the properties of a queue connection factory provided by the embedded WebSphere JMS provider, click **Resources-> WebSphere JMS Provider-> (In content pane, under Additional Properties) WebSphere Queue Connection Factories->** ***connection_factory***
4. Set the **Container-managed Authentication Alias** property.
5. Click **OK**

Administering listener ports

Use the following tasks to administer listener ports, which each define the association between a connection factory, a destination, and a message-driven bean.

You can use the WebSphere administrative console to administer listener ports, as described in the following tasks.

- Adding a new listener port

  Use this task to create a new listener port, to specify a new association between a connection factory, a destination, and a message-driven bean. This enables deployed message-driven beans associated with the port to retrieve messages from the destination.
- Configuring a listener port

  Use this task to view or change the configuration properties of a listener port.
- Starting a listener port

  Use this task to start a listener port manually.
- Stopping a listener port

  Use this task to stop a listener port manually.

Note: If configured as enabled, a listener port is started automatically when a message-driven bean associated with that port is installed. You do not normally need to start or stop a listener port manually.

Starting a listener port

Use this task to start a listener port on an application server, to enable the listeners for message-driven beans associated with the port to retrieve messages.

A listener is active, that is able to receive messages from a destination, if the deployed message-driven bean, listener port, and message listener service are all started. Although you can start these components in any order, they must all be in a started state before the listener can retrieve messages.

If configured as enabled, a listener port is started automatically when a message-driven bean associated with that port is installed. However, you can start a listener port manually, as described in this topic.

When a listener port is started, the listener manager tries to start the listeners for each message-driven bean associated with the port. If a message-driven bean is stopped, the port is started but the listener is not started, and remains stopped. If you start a message-driven bean, the related listener is started.

To start a listener port on an application server, use the administrative console to complete the following steps:

1. If you want the listener for a deployed message-driven bean to be able to receive messages at the port, check that the message-driven bean has been started.
2. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
3. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.

4. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
5. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
6. Select the checkbox for the listener port that you want to start.
7. Click **Start**.
8. To save your configuration, click **Save** on the task bar of the Administrative console window.

Stopping a listener port

Use this task to stop a listener port on an application server, to prevent the listeners for message-driven beans associated with the port from retrieving messages.

When you stop a listener port as described in this topic, the listener manager stops the listeners for all message-driven beans associated with the port.

To stop a listener port on an application server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, select the listener port that you want to stop.
6. Click **Stop**.
7. To save your configuration, click **Save** on the task bar of the Administrative console window.
8. To have the changed configuration take effect, stop then restart the application server.

Important files for message-driven beans and extended messaging

The following files in the WAS_HOME/temp directory are important for the operation of the WebSphere Application Server messaging service, so should not be deleted. If you do need to delete the WAS_HOME/temp directory or other files in it, ensure that you preserve the following files.

server_name-**durableSubscriptions.ser**
: You should not delete this file, because the messaging service uses it to keep track of durable subscriptions for message-driven beans. If you uninstall an application that contains a message-driven bean, this file is used to unsubscribe the durable subscription.

server_name-**AsyncMessageRequestLog.ser**
: You should not delete this file, because the messaging service uses it to keep track of late responses that need to be delivered to the late response message handler for the extended messaging provider.

Troubleshooting message-driven beans

Use this overview task to help resolve a problem that you think is related to message-driven beans.

Message-driven beans support uses the standard WebSphere Application Server troubleshooting facilities. If you encounter a problem that you think might be related to the message-driven beans, complete the following stages:

1. Check for messages about message-driven beans in the application server's SystemOut log at *was_home*\logs\server\SystemOut. Look in the SystemOut log for messages that indicate a problem with JMS resources for message-driven beans, such as listener ports.

2. Check for more messages in the application server's SystemOut log. If the JMS server is running, but you have problems accessing JMS resources, check the SystemOut log file, which should contain more error messages and extra details about the problem.
3. Check the Release Notes for specific problems and workarounds The section *Possible Problems and Suggested Fixes* of the Release Notes, available from the WebSphere Application Server library web site, is updated regularly to contain information about known defects and their workarounds. Check the latest version of the Release Notes for any information about your problem. If the Release Notes does not contain any information about your problem, you can also search the Technotes database on the WebSphere Application Server web site.
4. Check that message listener service has started. The message listener service is an extension to the JMS functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.
5. Check your JMS resource configurations If the WebSphere Messaging functions seem to be running properly (the JMS server is running without problems), check that the JMS resources have been configured correctly. For example, check that the listener ports have been configured correctly and have been started.
6. Check for problems with the WebSphere Messaging functions For more information about troubleshooting WebSphere Messaging, see the related topics.
7. Get a detailed exception dump for messaging. If the information obtained in the preceding steps is still inconclusive, you can enable the application server debug trace for the ″Messaging″ group to provide a detailed exception dump.

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.

Chapter 6. Using application clients

An application client module is a JAR (Java ARchive) file containing a client for accessing a Java application.

1. Decide on a type of application client.
2. Develop the application client code.

   Develop ActiveX application client code.

   Develop applet client code.

   Develop J2EE application client code.

   Develop pluggable application client code.

   Develop thin application client code.

View the Samples gallery for more information about application clients. Before you run the basicCalculator Sample, ensure the JMS Server is started.

Application clients

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 now these requests make use of 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.

With WebSphere Application Server Version 5, application clients now consist of the following models:
- ActiveX application client
- Applet client
- J2EE application client
- Pluggable application client
- Thin application client

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)) 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), 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 best suited 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

© Copyright IBM Corp. 2003

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. The J2EE application client 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 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 allows the application client to use *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 use. 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 the enterprise beans and local resources.

The application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access enterprise bean references and to use 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 application clients. Before you run the basicCalculator Sample, ensure the JMS Server is started.

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 to support communication between client and server	Yes	Yes	Yes	Yes	Yes

Allows 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 to Java Naming and Directory Interface (JNDI) 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 WebSphere Enterprise JavaBeans through a set of ActiveX automation objects.

The bridge accomplishes this 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 re-thrown 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 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 cross between 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.

Client side bindings generate using the Application Assembly Tool. An applet can utilize these bindings, or you can generate client side bindings using the **rmic** command that is part of the IBM Developer Kit, Java edition, 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 appropriately.

The applet client provides the run time to support the J2EE applet client. The applet client does not have tooling support for developing, assembling or deploying the applet. The applet client run time is provided through the Java applet browser plug-in that you install on the client machine using the WebSphere Application Server Client CD.

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 CosNaming or Java Naming and Directory Interface (JNDI) APIs.

J2EE application clients

The J2EE application client programming model provides the benefits of Java TM 2 Platform for WebSphere Application Server Enterprise(J2EE).

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 (Java Database Connectivity (JDBC), 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 RMI-IIOP protocol. Using this protocol enables the client application to access enterprise bean references and to use 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 client is only available on the Windows platform.*

This client is designed to support those users who want a lightweight Java client application programming environment, without the overhead of the J2EE platform on the client machine. The programming model for this client is heavily influenced by the CORBA programming model, but supports access to enterprise beans.

When accessing enterprise beans from this client, the client application can consider the enterprise beans object references as CORBA object references.

Tooling does not exist on the client; however, tooling does exists on the server. You are responsible for developing the client application, generating the necessary client bindings for the enterprise bean and CORBA objects, and after bundling these pieces together, installing them on the client machine.

The pluggable application client provides the necessary run time to support the communication needs between the client and the server.

The pluggable application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access not only enterprise bean references and CORBA object references, but the protocol 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 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 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 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 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 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 pluggable 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 pluggable client must know the fully qualified physical location of the enterprise bean Home 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 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.

Migration tips for application clients

Tips for migrating thin application client code:

- The Java invocation used to run non-J2EE application clients has changed in Version 5.0. You must specify `-Xbootclasspath/p:%WAS_BOOTCLASSPATH%` on Windows systems or `-Xbootclasspath/p:$WAS_BOOTCLASSPATH` on Unix systems when you invoke the Java command. Set the `WAS_BOOTCLASSPATH` environment variable in one of the following:
  - `setupClient.bat` for Windows systems or `setupClient.sh` for Unix systems on a WebSphere Application Server client installation.
  - `setupCmdLine.bat` for Windows systems or `setupCmdLine.sh` for Unix systems on a WebSphere Application Server installation.

  For more information about using `-Xbootclasspath`, view sample code at the following path after you preform the application client installation:

  ```
  install_root\samples\bin\ActiveXBridgeClients\VB\XJBExamples\modXJBHelpers.bas
  ```

Tips for migrating J2EE application client code:

- If your J2EE application client uses resource references and you have configured those resources using the Application Client Resource Configuration Tool (ACRCT), you must run the **ClientUpgrade** command to migrate the resource configuration information in WebSphere Application Server V5.

Installing application clients

All client applications run on a machine with the WebSphere Application Server installed. However, if the system does not have the Application Server installed, you can install Application Server clients, which provide a stand-alone client run-time environment for your client applications. See the Supported Prerequisites page on the IBM external Web site for more information on supported product platforms.

This article describes how to install the WebSphere Application Server clients using the installation image on the product CD-ROM labelled, **Application Clients**. The steps that follow provide enough detail to guide you through preparing for, choosing, and installing the variety of options and features provided. To prepare for installation and to make yourself familiar with installation options, complete the steps in this article and read the related topics, before you start to use the installation tools. Specifically, read these topics before installing the product:

- Tips for installing the embedded messaging feature
- Installing silently
- Best practices for installing

As a general rule, if you launch an installation and there is a problem such as not having enough temporary space or not having the right packages on your Linux or UNIX-based systems, then cancel the installation, make the required changes, and restart the installation to pick up changes you made.

Although it is not supported or recommended, you can install this product as a non-root user on a UNIX operating system, or from a user ID that is not part of the Administrator group on a Windows platform. However, there are certain components, such as the Embedded Messaging Client feature, that require you to install as root or as part of the Administrator group.

As previously mentioned, this installation method is not supported or recommended, but you can install the application client product on a machine with WebSphere Application Server installed. However, there are certain components, such as the Embedded Messaging Client feature, which might not install if the feature has already been installed during the WebSphere Application Server installation. On Windows platforms, WebSphere Application Server clients assume that the Embedded Messaging Client is installed in its default location; whereas, the WebSphere Application Server installation can install the messaging client in a different location.

1. Prepare a Linux or UNIX operating platform for the Embedded Messaging Client feature.

   If you are installing the embedded messaging feature, you must create two operating system groups as described in Installing WebSphere embedded messaging as the JMS provider.

   The Solaris Operating Environment and HP-UX also require you to increase kernel settings as described in Installing WebSphere embedded messaging as the JMS provider.

   For other platform-specific information about using the embedded messaging feature, see Tips for installing the embedded messaging feature.
2. Start the installation.
   a. Issue LaunchPad.sh (or LaunchPad.bat) to initiate the LaunchPad tool and begin the installation process.
   b. Click **Install the product** from the LaunchPad tool to launch the InstallShield for MultiPlatforms installation wizard. This action launches the installation wizard.

      The Readme documentation to which the LaunchPad links is the `readme.html` file in the CD root directory. The readme directory off the root of the CD has more detailed Readme files. The Installation Guide is in the `/docs` directory of the CD root directory.

      When you install application clients, the current working directory must be the directory where the installer binary program is located. This placement is important because the resolution of the class files location is done in the current working directory. For example:

      ```
      cd /install_root
      ./install
      ```

      or when installing from the product CD-ROM:

      ```
       cd <CD mount point>
           ./install
      ```

      Failing to use the correct working directory can cause ISMP errors that abort the installation.

      The installation wizard does not upgrade or remove previous WebSphere Application Server clients installation automatically. However, you must uninstall any previous installation manually or the installation wizard aborts the installation.
   c. As indicated in the previous example, you can start the installation wizard from the product CD-ROM, using the command line. The installation program is in the operating system platform directory on the product CD-ROM.

      On other Linux platforms and UNIX-based platforms, run the `./install` command.

      On Windows platforms, run the `Install.exe` command.
   d. You can also perform a silent installation using the `-options responsefile` parameter, which causes the installation wizard to read your responses from the options response file, instead of from the interactive graphical user interface. Customize the response file before installing silently. After customizing the file, issue the command to silently install. Silent installation is particularly useful if you install the product often.

      The rest of this procedure assumes that you are using the installation wizard. There are corresponding entries in the response file for every prompt that is described as part of the wizard. Review the description of the response file for more information. Comments in the file describe how to customize its options.

3. Select a language for the wizard GUI and click **Next**. The installation wizard opens and a Welcome panel is displayed.
4. Click **Next** to continue. The license displayed during the GUI installation can contain characters that display incorrectly in Japanese. For example, the section labeled Part 1 does not show the number 1. These missing characters do not significantly affect the content of the license agreement.
   a. Click the radio button beside the **I accept the terms in the license agreement** message if you agree to the license agreement, and click **Next** to continue.
5. Choose a type of installation, and click **Next**.

   If you use the GUI, you can choose a Typical installation type, which installs J2EE and Java Thin Application Client, Samples and Embedded Messaging Client features, or a Custom installation type.

   The Custom installation type lets you select which features to install. However you can not install the J2EE and Java Thin Application Client feature and the Pluggable Application Client feature together.
6. Install the samples development environment. If you choose to install any of the Samples features, a message box is displayed requesting conformation to install the Samples development environment.
   a. Click **Yes** to install the Samples development environment that includes the IBM Developer Kit and the Appache Ant tool.
   b. Click **No** to skip installing the Samples development environment.
7. Specify a destination directory. Click **Next** to continue.
   a. Ensure that there is adequate space available in the target directory.
   b. Specify a target directory for the WebSphere Application Server clients product. If you install the Embedded Messaging Client feature, then you cannot change its default installation directory.
   c. Enter the required target directory to proceed to the next panel. Deleting the default target location and leaving an installation directory field empty prevents you from continuing the installation process.
8. Enter the host name of the WebSphere Application Server machine. Click **Next** to continue. If you are connecting to a product Version 4 server or you are not using the default port, you must specify the server port number. The default port number for product Version 4 server is 900, and the default port number for product Version 5 is 2809.
9. Review the summary information, and click **Next** to install the product code or you might also click **Back** to change your specifications. When the installation is complete, the wizard displays the `install_root\logs\mq_install.log` installation log if you selected the Embedded Messaging Client feature, and there are errors with its installation.
10. Review the `mq_install.log` installation log, if it appears. Click **Next** to continue.
11. Click **Finish** to exit the wizard, after the WebSphere Application Server client installs.

You successfully installed WebSphere Application Server clients and the features you selected.

Best practices for installing application clients

The following table offers tips for installing application clients on multiple platforms.

Operating environment	**Tip**
Linux and UNIX systems	Spaces are not supported in the name of the installation directory on Linux and UNIX platforms.
UNIX systems	When client application installations are successful, the return code `1` is issued from the UNIX shell where you issued the `/install` command. Any other return code indicates an unsuccessful installation.
Solaris systems	Double-byte character set (DBCS) characters are not supported in the name of the installation directory on Solaris systems.

Windows NT systems	Spaces are not supported in the name of the installation directory. **Note:** WebSphere Application Server Version 5.1 does not support the Windows NT platform.
All platforms	Reserve at least 4 to 5MB free space in the target platform temporary directory.

All platforms	When specifying a different temporary directory while installing application clients, the following message is displayed if the target platform default temporary directory does not have enough free space to install application clients:
	`Error writing file =  There may not be enough`
	`temporary disk space.`
	`Try using -is:tempdir to use a temporary`
	`directory on a partition with more disk`
	`space.`
	
	Use the `-is:tempdir` installation option to specify a different temporary directory. For example, the following command uses `/swap` as a temporary directory during installation:
	`./install -is:tempdir /swap`

All platforms	After the installation, when changing the installation settings for the WebSphere Application Server host name and the port number, edit the `setupClient.bat` for Windows or `setupClient.sh` for UNIX. Change the `DEFAULTSERVERNAME` and `SERVERPORTNUMBER` to the new WebSphere Application Server host name and port number, respectively. If the `SERVERPORTNUMBER` is not set, then the default is 2809. Review the following example:
	`set DEFAULTSERVERNAME=NDServerName`
	`set SERVERPORTNUMBER=9810`
	
	The `setupClient.bar` file or `setupClient.sh` file is located in the `bin` sub-directory under the application clients installation destination.

Installing application clients silently

Use these steps to perform a silent installation, which uses the installation wizard to install the product. Instead of displaying a user interface, the silent installation provides interaction between you and the wizard by reading all of your responses from a file that you must customize.

1. Ensure that the user ID that you are using to run the silent installation has sufficient authority to perform the task.

   If you are installing the embedded messaging feature, you must create two operating system groups as described in the article, Installing WebSphere embedded messaging as the JMS provider.

   Although it is not supported or recommended, you can install this product as a non-root user on a UNIX-based operating system, or from a user ID that is not part of the Administrator group on a Windows platform. However, there are certain components, such as the Embedded Messaging Client feature, that require you to install as root or as part of the Administrator group.
2. Customize the option response file.
   a. Locate the sample options response file. The file name is `setup.response` in the operating-system platform directory on the product CD-ROM.
   b. Make a copy to preserve the original response file. For example, copy it as `myoptionsfile`.
   c. Edit the copy in your flat file editor of choice, on the target operating system. Read the directions within the response file to choose appropriate values.

Note: To prepare the file for a silent installation on AIX, use UNIX line-end characters (0x0D0A) to terminate each line of the options response file.

   d. Make the first non-commented option `-silent` to have a silent install.
   e. Include custom option responses that reflect parameters for your system.
   f. Follow the instructions in the response file to choose appropriate values.
   g. Save the file.

3. Issue a command to use your custom response file: `Install.exe -options myoptionsfile` for Windows platforms `install -options ./myoptionsfile` for Linux and UNIX platforms

   The sample options response file is located in the `operating-system platform` directory on the product CD.
4. Optional: Restart your machine in response to the prompt that appears on Windows platforms. If you install the Embedded Messaging Client feature, `-P JSMSupport.active="true"`, certain conditions, such as a locked file, might require you to restart your system. You have the option of restarting immediately, after which the installation program resumes the installation process. You can also defer restarting to a more convenient time, such as after the installation is complete.

You installed application clients silently by using the response file.

To further verify that the silent installation was successful, examine the `WAS.Client.Install.log` file for a line similar to this:

```
The InstallShield Wizard has successfully installed IBM WebSphere Application Server clients, Version 5.1.
```

If you installed the Embedded Messaging Client feature, you can search the `mq_install.log` for any errors.

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. You should also consider the information given in ActiveX to EJB bridge, 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 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 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 subtasks:

1. Starting an ActiveX application and configuring service programs
2. Starting 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 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 JVM version available on a machine using ASP.

To add the JRE directories to your System path, complete one of the following subtasks:

1. On Windows 2000, complete the following substeps:
   a. Open the Control Panel, then double-click the System icon.
   b. Click the Advanced tab on the System Properties window.
   c. Click **Environment Variables**.
   d. Edit the Path variable in the System Variables window.
   e. Add the following to the beginning of the path displayed in the Variable Value input box:

```
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 WebSphere Java client
   f. Click **OK** in the Edit System Variable window to apply the changes.
   g. Click **OK** in the Environment Variables window.
   h. Click **OK** in the System Properties window.
   i. Restart Windows 2000.
2. On Windows NT, complete the following substeps:

   Note: WebSphere Application Server Version 5.1 does not support the Windows NT platform.
   a. Open the Control Panel, then double-click the System icon.
   b. Click the Environment tab on the System Properties window.
   c. In the System Variables window, edit the Path variable.
   d. Add the following to the beginning of the path displayed in the Value input box:

```
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 WebSphere Java client
   e. Click **Set** to apply the changes.
   f. Click **OK**.
   g. Restart Windows NT.

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 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 runtime. 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 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 the application can be run, 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 EJB bridge to find its `XJB.JAR` file and the Java run-time.

Location

The include file is located in the *was_client_home*`\aspIncludes` directory. You can include the file into your ASP application with the following syntax in your ASP page:

```
<--  #include virtual ="/WSASPIncludes/setupASPXJB.inc" -->
```

This assumes that you have created a virtual directory in Internet Information Server called WSASPIncludes that points to the *was_client_home*`\aspIncludes` directory.

Usage notes

The following batch files are provided for client applications to use the ActiveX to EJB bridge:

- **setupCmdLineXJB.bat**

  Sets the client environment variables.
- **launchClientXJB.bat**

  Calls the `setupCmdLineXJB.bat` file and launches the application you specify as its arguments; for example:
  ```
  launchClientXJB.bat myapp.exe parm1 parm2
  ```
  or
  ```
  launchClientXJB MyApplication.vbp
  ```
- **Active Server Pages (ASP) include file**

  An include file is provided for ASP users to automatically set the following page-level (local) environment variables:
  - **com_ibm_websphere_javahome** Path to the Java run-time directory installed with the WebSphere Advanced Server Client.
  - **com_ibm_websphere_washome** Path to the WebSphere Advanced Server Client directory.

  – **com_ibm_websphere_namingfactory** Sets the Java java.naming.factory.initial system property.
  – **com_ibm_websphere_computername** (Optional) Name of the computer where the WebSphere Advanced Server Client is installed. If you intend to talk to a single specific computer, you are recommended to change this value to become the server name that you intend to access.
- **System Settings**

  To enable the ActiveX to EJB bridge to access the Java run-time Dynamic Link Library (DLL), the following directories must exist in the system PATH environment variable:

  ```
  was_client_home\java\jre\bin;was_client_home\java\jre\bin\classic
  ```

  Where *was_client_home* is the name of the directory where you installed the WebSphere Application Server Client (for example, `C:\WebSphere\AppClient`).

  Note: This technique enables only *one* Java run-time to activate on a machine, therefore all client services on that machine must use the same Java run-time. Client applications do not have this limitation because they each have their own private, non-system scope.

JClassProxy and JObjectProxy classes

This topic provides reference information about the object classes of the ActiveX to EJB bridge.

JClassFactory is the object used to access the majority of JVM features. It handles JVM initialization, accessing classes and creating 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 would normally send to the java.exe file.

  If you have invalid parameters in the XJBInit() string array, the following error results:

  ```
  Error: 0x6002 "XJBJNI::Init() Failed to create VM" when calling XJBInit()
  ```

  If you have C++ logging enabled, the activity log displays the invalid parameter.
- JClassProxy FindClass(String strClassName)

  Uses the current thread class loader to load the specified fully qualified class name and returns a JClassProxy object representing the Java Class object.
- JObjectProxy NewInstance()

  Creates a Class instance for the specified JClassProxy object using the parameters supplied to call the Class Constructor. For more information about using JMethodArgs, 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 also from any Java method call that would 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 would normally return 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 to a constructor that normally takes a java.lang.Object type).

There are two groups of methods to add arguments to the collection: Add and Set. You can use Add to add arguments in the order that they are declared. Alternatively, you can use Set to set an argument based on its position in the argument list (where the first argument is in position 1).

For example, if you had a Java Object Foo that took a constructor of Foo(int, String, Object), you could use a JMethodArgs object as shown in the following code extract:

```
...
Dim oArgs as Object
set oArgs = oXJB.GetArgsContainer()

oArgs.AddInt(CLng(12345))
oArgs.AddString("Apples")
oArgs.AddObject("java.lang.Object", oSomeJObjectProxy)

Dim clsFoo as Object
Dim oFoo as Object
set clsFoo = oXJB.FindClass("com.mypackage.foo")
set oFoo = oXJB.NewInstance(clsFoo, oArgs)

' To reuse the oArgs object, just clear it and use the add method
' again, or alternatively, use the Set method to reset the parameters
' Here, we will use Set
oArgs.SetInt(1, CLng(22222))
oArgs.SetString(2, "Bananas")
oArgs.SetObject(3, "java.lang.Object", oSomeOtherJObjectProxy)

Dim oFoo2 as Object
set oFoo2 = oXJB.NewInstance(clsFoo, oArgs)
```

- AddObject (String strObjectTypeName, Object oArg)

  Adds an arbitrary object to the argument container in the next available position, casting the object to the class name specified in the first parameter. Arrays are specified using the traditional [] syntax; for example:

```
AddObject("java.lang.Object[][]", oMy2DArrayOfFooObjects)
```

or

```
AddObject("int[]", oMyArrayOfInts)
```

- AddByte (Byte byteArg)

  Adds a primitive byte value to the argument container in the next available position.
- AddBoolean (Boolean bArg)

  Adds a primitive boolean value to the argument container in the next available position.
- AddShort (Integer iArg)

  Adds a primitive short value to the argument container in the next available position.
- AddInt (Long lArg)

  Adds a primitive int value to the argument container in the next available position.
- AddLong (Currency cyArg)

  Adds a primitive long value to the argument container in the next available position.
- AddFloat (Single fArg)

  Adds a primitive float value to the argument container in the next available position.
- AddDouble (Double dArg)

  Adds a primitive double value to the argument container in the next available position.
- AddChar (String strArg)

  Adds a primitive char value to the argument container in the next available position.
- AddString (String strArg)

  Adds the argument in string form to the argument container in the next available position.
- SetObject (Integer iArgPosition, String strObjectTypeName, Object oArg)

  Adds an arbitrary object to the argument container in the specified position casting it to the class name or primitive type name specified in the second parameter. Arrays are specified using the traditional [] syntax; for example:

  ```
  SetObject(1, "java.lang.Object[][]", oMy2DArrayOfFooObjects)
  ```

  or

  ```
  SetObject(2, "int[]", MyArrayOfInts)
  ```
- SetByte (Integer iArgPosition, Byte byteArg)

  Sets a primitive byte value to the argument container in the position specified.
- SetBoolean (Integer iArgPosition, Boolean bArg)

  Sets a primitive boolean value to the argument container in the position specified.
- SetShort (Integer iArgPosition, Integer iArg)

  Sets a primitive short value to the argument container in the position specified.
- SetInt (Integer iArgPosition, Long lArg)

  Sets a primitive int value to the argument container in the position specified.
- SetLong (Integer iArgPosition, Currency cyArg)

  Sets a primitive long value to the argument container in the position specified.
- SetFloat (Integer iArgPosition, Single fArg)

  Sets a primitive float value to the argument container in the position specified.
- SetDouble (Integer iArgPosition, Double dArg)

  Sets a primitive double value to the argument container in the position specified.
- SetChar (Integer iArgPosition, String strArg)

  Sets a primitive char value to the argument container in the position specified.
- SetString (Integer iArgPosition, String strArg)

  Sets a java.lang.String value to the argument container in the position specified.
- Object Item(Integer iArgPosition)

  Returns the value of an argument at a specific argument position.
- Clear()

Removes all arguments from the container and resets the next available position to one.
- Long Count()

  Returns the number of arguments in the container.

Java virtual machine initialization tips

Initialize the Java virtual machine (JVM) code with the ActiveX to enterprise JavaBeans 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 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 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 classpath, 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 JVM code after it has loaded. Therefore, after XJBInit() 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 ActiveX to enterprise bean bridge, using Java proxy objects

To use Java proxy objects with the ActiveX to enterprise JavaBeans bridge:
- After an ActiveX client program (Visual Basic, VBScript, or ASP) has initialized the XJB.JClassFactory (and thereby the JVM), it can access Java classes and initialize Java objects. To do this, the client program uses the XJB.JClassFactory FindClass() and NewInstance() methods.
- In Java programming there are two ways 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 runtime interface to Java, it 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) from it, as described below. For example, the following Visual Basic code extract returns a Proxy object for the Java class java.lang.Integer:

```
...
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 in Java programming are really 32-bits (which translates to Long in VB), 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 java.lang.Integer:

  ```
  ...
  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 bean bridge

In the ActiveX to EJB bridge, methods are called using the native language method invocation syntax.

- The following are important differences between Java invocation and ActiveX Automation invocation:
  – 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.
- You should 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 would fail if CLng() was not present or toHexString 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()` (uncapitalized), Visual Basic would try to capitalize it ″`Close()`″. In Visual Basic, the only way to effectively get 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.) For example:

  ```
  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 returnd 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). 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 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 this:

```
Dim oMyHashtable as Object
Set oMyHashtable = _
    oXJB.NewInstance(oXJB.FindClass("java.utility.Hashtable"))

' This line will not work. The ActiveX to EJB bridge cannot find a method
' called "put" that has a short and String as a parameter:
oMyHashtable.put 100, "Dogs"
oMyHashtable.put 200, "Cats"

' You must use a XJB.JMethodArgs object instead:
Dim oMyHashtableArgs as Object
Set oMyHashtableArgs = oXJB.GetArgsContainer
oMyHashtableArgs.AddObject("java.lang.Object", 100)
oMyHashtableArgs.AddObject("java.lang.Object", "Dogs")

oMyHashtable.put oMyHashTableArgs
' Reuse the same JMethodArgs object by clearing it.
oMyHashtableArgs.Clear
oMyHashtableArgs.AddObject("java.lang.Object", 200)
oMyHashtableArgs.AddObject("java.lang.Object", "Cats")

oMyHashtable.put oMyHashTableArgs
```

Java field programming tips

Using the ActiveX to 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 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 need to use the CallByName() function to set a field in the same way that you would call a method in some cases. For Fields, you 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:

  ```
  ...
  Dim VariantArray(1) as Variant
  VariantArray(0) = CLng(123)
  VariantArray(1) = CDbl(123.4)
  oMyJavaObject.foo(VariantArray) '     Illegal!

  VariantArray(0) = CLng(123)
  VariantArray(1) = CLng(1234)
  oMyJavaObject.foo(VariantArray) '     This works
  ```
- Arrays of Primitive Types are converted using the rules defined in Primitive Data Type Conversion.

- Arrays of Java Objects are handled through arrays of JObjectProxy objects.
- Arrays of JObjectProxy objects must be fully-initialized and of the correct associated Java type. When initializing an array in Visual Basic (for example, Dim oJavaObjects(1) as Object), you must set each object to a JObjectProxy before you send the array to Java. 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 multi-dimensional arrays are treated as zero-based multi-dimensional arrays in Visual Basic and VBScript.
- 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 re-thrown 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 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. 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 JVM code. Therefore, you must be careful that any Java code that you access from a multi-threaded Windows application is thread-safe. Visual Basic 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 multi-threaded C and C++ programs can have issues.

Consider the following scenario:
1. A multi-threaded 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 Thread1.
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 VM 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's Internet Information Server is a multi-threaded 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 and encapsulate the ActiveX to EJB bridge object there. This encapsulation guarantees that all access to the JVM object is on the same thread. You need to use the <OBJECT> tag to assign the XJB.JClassFactory to an Application object and must be aware of the consequences of introducing single-threaded behavior to a Web application.

The Microsoft KnowlegeBase has several articles about ASP and threads, including:
- Q243543 INFO: Do Not Store STA Objects in Session or Application
- Q243544 INFO: Component Threading Model Summary Under Active Server Pages
- Q243548 INFO: Design Guidelines for VB Components Under ASP

Example: Viewing System.out message

The ActiveX to 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 Application 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 will be disabled.

  To enable or disable logging certain types of events to the Windows Application Event Log, you need to 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 `XJBInit()`. Once `XJBInit()` 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 will have no effect.

  – `-Dcom.ibm.ws.client.xjb.native.logging.debug=enabled|disabled`

    Enables or disables debug level messages from appearing 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, click **Start** > **Settings** > **Control Panel**. Double-click **Administrative Tools**, and then double-click **Event Viewer**. All ActiveX to EJB bridge events will have the text ″WebSphere XJB″ in the source column and will appear 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 out of the Visual Basic editor before debugging your program. If you run 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 Visual Basic.
- 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 VBScriptfiles 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 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 classpaths or JVM settings you need separate ASP applications (virtual directories with high isolation or an isolated process).
- Store the XJB.JClassFactory object in application scope.

  Because of the one-to-one relationship required between a JVM instruction and a process, and because the JVM code can never detach or shut down from a process independently, cache the XJB.JClassFactory object at application scope and call the XJBInit() method only once.

  Because the ActiveX to EJB bridge employs a free-threaded marshaler, take advantage of the multi-threaded nature of Internet Information Server and the ASP environment. If you choose to reinitialize the XJB.JClassFactory object at Page scope (local variables), then the XJBInit() method can only initialize your local XJB.JClassFactory variable. It is more efficient to use the XJBInit() method once.
- Use VBScript conversion functions.

  Because VBScript code only supports variant data types, use the CStr(), CByte(), CBool(), CCur(), CInt(), Clng(), CSng() and CDbl() functions to tell the activeX to EJB bridge which data type you are using; for example `oMyObject.Foo(CDbl(1.234))`.

J2EE guidelines

The following guidelines are intended to help optimize your use of the ActiveX to EJB bridge with the J2EE environment;

- Store client container objects globally.

  Because you can only have one JVM instruction per process, and a single J2EE client container (com.ibm.websphere.client.applicationclient.launchClient) per JVM instruction, initialize your J2EE client container only once and reuse it. For ASP applications, store the J2EE client container in an application level variable and initialize it only once (either on the Application_OnStart() event in the `global.asa` file or by checking to see if it IsEmpty()).

  A side effect to storing the client container object globally is that you cannot change the client container parameters without destroying the object and creating a new one. These parameters include the EAR file, BootstrapHost, classpath, 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 applicatio, which is unsupported.
- Reuse custom temp 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 ClassLoader to use the extracted EAR file directory and 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 normaly. 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 launchClient() 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, make sure 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 very 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, Windows NT or Windows 2000 platforms. Version 5.1 does not support the Windows NT platform. Check the prerequisites page for information on new platform support.
- They require one of these browsers:
  - Internet Explorer version 5.0+
  - Netscape Navigator 4.7+
- You must install the browser before installing the client code.

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:

```
-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 above entries are automatically placed into the WebSphere Application Server Java Plug-in control panel for the user who installed the WebSphere Application Sever 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** > **WebSphere 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 Java Plug-in.

View the Samples gallery for more information about application clients. Before you run the basicCalculator Sample, ensure the JMS Server is started.

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 SSL will be used 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** > **WebSphere Java Plug-in** to open the WebSphere Application Server Java control panel.
   - To use the file you created in step number 1, modify the

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

   value.

For more information on the `sas.client.props` file and WebSphere Application Server security, see the Security section of the InfoCenter.

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 permissions:

- The 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 hostname 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.

- For applets to communicate with EJBs in the WebSphere Application Server environment, the `<APPLET>` tag must be replaced with the following tags:

```
<OBJECT>
<EMBED>
```

- 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 <PARM> tag. Although both 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=1.3">
<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-websphere-client" 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>
```

- The value of the type attribute on the </EMBED> tag can also be, for example:

```
<EMBED type="application/x-websphere-client, version=4.0" ...
```

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 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 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) name space 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. Writing 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 lookup operation to access the Java Naming and Directory Interface (JNDI) name space. 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 name space at `java:comp/env/ejb/BasicCalculator` . Since the actual enterprise bean runs 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 lookup returns an instance of the configured type as defined by the client application deployment descriptor. For example, if the program lookup was a JDBC datasource, 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 the Assembly Toolkit.

   The JNDI name space 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.

   Keep the following in mind 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 is a table of the 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 Datasource 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 topmost class loader, the `bootstrap class loader`, contains the JAR files that make up the Java Virtual Machine code, such as `rt.jar`, plus those JAR files defined by the `-Xbootclasspath` parameter on the Java command. The WebSphere Application client run time sets this value to the `WAS_BOOTCLASSPATH` 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 WebSphere Application 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 WebSphere Application Client run time and any classes placed in the WebSphere 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 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 Database Connectivity (JDBC) API, Java Message Service (JMS) API, or Uniform Resource Locator (URL), a different class loader is created to load each of those resources. Finally, the application client run time sets the

WebSphere class loader to load classes within the `.ear` file by processing the client JAR manfest repeatedly. The system classpath, 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 Java 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 WebSphere 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 WebSphere 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 will still function 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 classpath 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 classpath 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 J2EE applications do not use the system class path. Use the MANIFEST Classpath entry to refer to other JARs within the `.ear` file. Configure these values using the module Classpath fields in the Application Assembly Tool. For example, if your client application needs to access the path of the enterprise bean JAR, add the deployed enterprise bean module name to your application client Classpath field in the Application Assembly Tool. The format of the Classpath field for each of the different modules (Application Client, enterprise bean, 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 semi-colons.

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 Classpath 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 Assembly Tool you can add individual class files to the `.ear` file. It is recommended that these additional class files are packaged in a `.jar` file. Add this `.jar` file to the module Classpath fields. If you add `.class` files to the root of the `.ear` file, add **./** to the module Classpath 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 classpath values, which means multiple values are separated by semi-colons or colons. There are many similarities between the client and the server in this respect.

Resource classpaths

When you configure resources used by your client application using the Application Client Resource Configuration Tool, you can specify classpaths that are required by the resource. For example, if your application is using a JDBC to a DB2 database, add `db2java.zip` to the classpath field of the database provider. These classpath 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. You should mimic 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 from the WebSphere Application Client CD 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 -Xbootclasspath/p:%WAS_BOOTCLASSPATH% -classpath
   <list of your 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>
   ```
   ```
   $JAVA_HOME/bin/java -Xbootclasspath/p:$WAS_BOOTCLASSPATH -classpath
   <list of your 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 application clients. Before you run the `basicCalculator` Sample, ensure the JMS Server is started.

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 from the WebSphere Application Client CD by selecting option **J2EE/Thin application client for the WebSphere Application Client**.
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 application clients.

Developing thin application client code on a client machine

You must install the thin application client from the WebSphere Application Client CD 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:

   ```
   install_root\AppClient\bin\setupClient.bat (on Windows)
   install_root/AppClient/bin/setupClient.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 following Java command to invoke your client application: On Windows systems, enter:

```
"%JAVA_HOME%\bin\java" "-Xbootclasspath/p:%WAS_BOOTCLASSPATH%"
-Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"
-classpath "%WAS_CLASSPATH%;<list of your application jars and classes>"
-Djava.ext.dirs=%WAS_EXT_DIRS% -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
"%SERVER_ROOT%" "%CLIENTSAS%" <fully qualified class name to run>
<your application parameters>
```

   On Unix systems, enter:

```
$JAVA_HOME/bin/java -Xbootclasspath/p:$WAS_BOOTCLASSPATH
-classpath "$WAS_CLASSPATH:<list of your application jars and classes>"
-Djava.ext.dirs=$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
"$SERVER_ROOT" "$CLIENTSAS" <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 application clients.

Developing thin application client code on a server machine

You must install WebSphere Application Server before performing this task.

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

   ```
   install_root\bin\setupCmdLine.bat (on Windows)
   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 to call your main class directly:

     On Windows system, enter:

```
"%JAVA_HOME%\bin\java" "-Xbootclasspath/p:%WAS_BOOTCLASSPATH%"
-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:

```
"$JAVA_HOME/bin/java" "-Xbootclasspath/p:$WAS_BOOTCLASSPATH"
-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" "-Xbootclasspath/p:%WAS_BOOTCLASSPATH%"
-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:

```
"$JAVA_HOME/bin/java" "-Xbootclasspath/p:$WAS_BOOTCLASSPATH"
-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 application clients.

Assembling application clients

Assemble a client module to contain application client code. Group enterprise beans, Web components, and resource adapter code in separate modules.

Application client projects contain programs that run on networked client systems. An application client project is deployed as a JAR file.

Use the Assembly Toolkit 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 the Assembly Toolkit.
2. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
4. 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 Toolkit.
5. Create a new application client.
6. Verify the contents of the new application client in either of the following ways:
   - In the J2EE Hierarchy view, expand **Application Client Modules** 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.

Deploying 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 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.

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 client resources.
6. Configure new environment entries.
7. (Optional) Remove application client resources.
8. Save the EAR file.

Starting the Application Client Resource Configuration Tool and opening an EAR file

Note: This task only applies to J2EE application clients.

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 application clients

The J2EE application client does not support looking up or directly accessing data sources configured on WebSphere Application Server because the J2EE application client does not support Java 2 Connection Factories. To use a data source directly from the client application, you must use the ACRCT to configure your data source. In addition, WebSphere Application Server and WebSphere Application Server clients 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. 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 the WebSphere Application Server. For a current list of providers that are supported on the WebSphere Application Server go the following site:

Supported hardware, software, and APIs

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.

1. Start the tool 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.

Configuring new data source providers

During this task, you will create new data source providers, also known as JDBC drivers, for your application client. In a separate administrative task, install the Java code for the required data source provider on the client machine where the application client resides.

1. Start the ACRCT, click **File > Open**, and select 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. Right click the **Data Source Providers** folder and select **New Provider**.
5. Configure the data source provider properties in the resulting property dialog.
6. Click **OK**.
7. Click **File** > **Save** 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 equivalant xmi tags. You must specify these fields in the custom properties.
  - The password is encrypted when you use the Application Client Resource Configuration Tool (ACRCT). If you do not use the ACRCT, the field cannot be encrypted.
- Example:

```
<resources.jdbc:JDBCProvider xmi:id="JDBCProvider_1" name="jdbcProvider:name"
description="jdbcProvider:description" implementationClassName="jdbcProvider:
ImplementationClass">
<classpath>jdbcProvider:classPath</classpath>
<factories xmi:type="resources.jdbc:WAS40DataSource" xmi:id="WAS40DataSource_1"
name="jdbcFactory:name" jndiName="jdbcFactory:jndiName"
description="jdbcFactory:description" databaseName="jdbcFactory:databasename">
<propertySet xmi:id="J2EEResourcePropertySet_13">
<resourceProperties xmi:id="J2EEResourceProperty_13" name="jdbcFactory:customName"
value="jdbcFactory:customValue"/>
<resourceProperties xmi:id="J2EEResourceProperty_14" name="user"
value="jdbcFactory:user"/>
<resourceProperties xmi:id="J2EEResourceProperty_15" name="password"
value="{xor}NTs9PBk+PCswLSZlMT4yOg=="/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_14">
<resourceProperties xmi:id="J2EEResourceProperty_16" name="jdbcProvider:customName"
value="jdbcProvider:customeValue"/>
</propertySet>
</resources.jdbc:JDBCProvider>
```

Data source provider settings for application clients:

Use this page to create a data source under a JDBC provider which provides the specific JDBC driver implementation class.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Data Source Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the display name for the data source.

For example you can set this field to *Test Data Source*.

Data type	String

Description:

Specifies a text description for the resource.

Data type	String

Class Path:

A list of paths or jarfile 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 runtime 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 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.

Password:

Use the password 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.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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. Do one of the following:
   - 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 folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the data source properties in the resulting property dialog.
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. Open 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. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

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.

Re-Enter Password:

Confirms the password.

Mail From:

Specifies the mail originator.

Mail Store Host:

Mail account host (or ″domain″) name.

Mail Store User:

The user ID of the mail account.

Mail Store Password:

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 will be printed to stdout.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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}Mj42Mww6LCw2MDFlMT4yOg=="
mailFrom="mailSession:mailFrom" mailStoreHost="mailSession:mailStoreHost"
mailStoreUser="mailSession:mailStoreUser"
mailStorePassword="{xor}Mj42Mww6LCw2MDFlMT4yOg==" debug="true"
mailTransportProtocol="ProtocolProvider_1" mailStoreProvider="ProtocolProvider_1">
<propertySet xmi:id="J2EEResourcePropertySet_1">
<resourceProperties xmi:id="J2EEResourceProperty_1"
name="mailSession:customName" value="mailSession:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_2">
<resourceProperties xmi:id="J2EEResourceProperty_2" name="mailProvider:customName"
value="mailProvider:customValue"/>
</propertySet>
<protocolProviders xmi:id="ProtocolProvider_1" protocol="smtp"
classname="smtp:className"/>
<protocolProviders xmi:id="ProtocolProvider_2" protocol="pop3"
classname="pop3:className"/>
<protocolProviders xmi:id="ProtocolProvider_3" protocol="imap"
classname="imap:className"/>
</resources.mail:MailProvider>
```

Configuring new mail sessions for application clients

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 resulting property dialog.
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 its contents.
5. Click the folder called **URL Providers**. Complete one of the following actions:
   - Right-click the folder and click **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. Open 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:

Administrative name for the URL

Description:

Optional description of the URL, for your administrative records

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

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 runtime 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 URLs.

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 `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 runtime 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 folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit -> New** on the menu bar.
4. Configure the URL properties in the resulting property dialog.
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 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 Pub and Sub messaging. A J2EE application can explicitly poll for messages on a destination then retrieve messages for processing by business logic beans (enterprise beans).

With the base JMS/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.

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. Click the **JMS Providers** folder and click **New Provider**.
6. Configure the JMS provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File** > **Save**.

Configuring new JMS providers with the Application Client Resource Configuration Tool

During this task, you will create new JMS provider configurations for your application client. The application client can make use of a messaging service through the Java Message Service APIs. A JMS provider provides two kinds of J2EE factories. One is a JMS Connection factory, and the other is a JMS destination factory.

In a separate administrative task, you must install the 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.

1. Start the tool and open the EAR file for which you want to configure the new JMS provider. The EAR file contents will be 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. Click the folder called **JMS Providers**. Do one of the following:
   - Right-click the folder and select **New Provider**.
   - On the menu bar, click **Edit -> New**.
5. In the resulting property dialog, configure the JMS provider properties.
6. When finished, click **OK**.
7. On the menu bar, click **File -> Save** to save your changes.

JMS provider settings for application clients

Use this page to configure properties of the JMS provider, if you want to use a JMS provider other than the internal WebSphere JMS provider or the MQSeries 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 **JMS Providers** > 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 jarfile names which together form the location for the resource provider classes.

Context factory class:

The Java classname 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 runtime 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 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 internal WebSphere JMS provider that is installed with WebSphere Application Server. These configuration properties control how connections are created to the associated JMS queue destination.

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 > **JMS Providers** > *JMS provider instance*. Right-click **WAS 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 JMS provider. A queue connection factory for the internal WebSphere 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. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

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.

WebSphere 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 WebSphere JMS provider. These configuration properties control how connections are created to the associated JMS topic destination.
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 > **JMS Providers** > *JMS provider instance*. Right-click **WAS Topic Connection Factories** and click **New**. The following fields appear on the **General** tab.

A topic connection factory is used to create JMS connections to topic destinations. The topic connection factory is created by the associated JMS provider. A topic connection factory for the internal WebSphere 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 topic 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. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

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
Default	Null

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	Enum
Default	Null
Range	Pull-down list of nodes in the WebSphere 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 non-persistent, non-transactional, non-durable subscriptions only.

Note: Message-driven beans cannot use the direct listener port for publish/subscribe support. Therefore, any topic connection factory configured with **Port** set to `Direct` cannot be used with message-driven beans.

Data type	Enum
Units	Not applicable
Default	QUEUED

Range	**QUEUED** The listener port used for full-function JMS-compliant, publish/subscribe support. **DIRECT** The listener port used for direct TCP/IP connection (non-transactional, non-persistent, and non-durable subscriptions only) for publish/subscribe support. The TCP/IP port numbers for these ports are defined on the WebSphere 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 runtime 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 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 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 > **JMS Providers** > *JMS provider instance*. Right-click **WAS Queue Destinations** and click **New**. The following fields appear on the **General** tab.

A queue destination is used to configure the properties of a JMS queue. Connections to the queue are created by the associated queue connection factory for the internal WebSphere JMS provider. A queue for use with the internal WebSphere 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
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type	Enum
Units	Not applicable
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. **Non persistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level

Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the **Specified expiry** property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds
Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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 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 WebSphere 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 > **JMS Providers** > *JMS provider instance*. Right-click **WAS 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. Connections to the topic are created by the associated topic connection factory. A topic for use with the internal WebSphere 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.

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

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, non-persistent, or have their persistence defined by the application

Data type	Enum
Units	Not applicable
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. **Non persistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type	Enum
Units	Not applicable

Default	APPLICATION_DEFINED
Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the* ***Specified priority*** *property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the* ***Specified expiry*** *property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds

Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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.

MQSeries 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 JMS provider. These configuration properties control how connections are created to the associated JMS queue destination.
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 > **JMS Providers** > *JMS provider instance*. Right-click **MQ Queue Connection Factories** and click **New**. The following fields appear on the **General** tab.

A queue connection factory creates JMS connections to queue destinations. The queue connection factory is created by the MQSeries JMS provider. A queue connection factory for the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries for JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book, located in the WebSphere MQ Family library.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

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. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

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 hostname

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 JNI bindings are used for connection to the WebSphere MQ queue manager. The external JMS provider controls the communication protocols between JMS clients and JMS servers. Tune the transport type when you are using non-ASF nonpersistent, nondurable, nontransactional messaging or when you want to satisfy security issues and the client is local to the queue manager node.

Data type	Enum
Units	Not applicable
Default	BINDINGS
Range	**BINDINGS** JNI 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 comes at some security risks that should be addressed through the use of EJB roles. **CLIENT** WebSphere MQ client connection is used to connect to the queue manager. CLIENT is a typical TCP-based protocol. **DIRECT** For WebSphere MQ Event Broker using DIRECT mode. DIRECT is a lightweight sockets protocol used in nontransactional, nondurable and nonpersistent Publish/Subscribe messasging. DIRECT is only works for clients and message-driven beans using the non-ASF protocol. **QUEUED** QUEUED is a standard TCP protocol.

Recommended	**Queue connection factory transport type** BINDINGS is faster by 30% or more, but it lacks security. When you have security concerns, BINDINGS is more desirable than CLIENT. **Topic connection factory transport type** DIRECT is the fastest 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 fallback for all other cases. Note, WebSphere MQ 5.3 before CSD2 with the DIRECT setting can lose messages when used with message-driven beans and under load. This also happens with client-side based applications unless the broker's maxClientQueueSize is set to 0. You can set this to 0 with the command `#wempschangeproperties WAS_nodeName_server1 -e default -o DynamicSubscriptionEngine -n maxClientQueueSize -v 0 -x executionGroupUUID`, where executionGroupUUID can be found by starting the broker and looking in the Event Log/Applications for event 2201. This value is usually ffffffff-0000-0000-000000000000.

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 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 dealt with 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 dealt with according to their disposition options.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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.

MQSeries 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 MQSeries JMS provider. These configuration properties control how connections are created to the associated JMS topic destination.
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 > **JMS Providers** > *JMS provider instance*. Right-click **MQ Topic Connection Factories** and click **New**.

A topic connection factory is used to create JMS connections to topic destinations. The topic connection factory is created by the MQSeries JMS provider. A topic connection factory for the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

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
Default	Null

JNDI Name:

The JNDI name that is used to bind the topic connection factory into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 45 ASCII characters

User:

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
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 hostname

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:

Whether MQSeries client connection or JNDI bindings are used for connection to the MQSeries queue manager.

Data type	Enum
Units	Not applicable
Default	BINDINGS
Range	**CLIENT** MQSeries client connection is used to connect to the MQSeries queue manager. **BINDINGS** JNDI bindings are used to connect to the MQSeries queue manager.

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

Broker Control Queue:

The name of the broker control queue, to which all command messages (except publications and requests to delete publications) are sent

The name of the broker control queue. Publisher and subscriber applications, and other brokers, send all command messages (except publications and requests to delete publications) to this queue.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker Queue Manager:

The name of the MQSeries queue manager that provides the Pub/Sub message broker.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker Pub Queue:

The name of the broker's 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
Default	Null
Range	1 through 48 ASCII characters

Broker Sub Queue:

The name of the broker queue from which non-durable subscription messages are retrieved

The name of the broker's queue from which non-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
Default	Null
Range	1 through 48 ASCII characters

Broker CCSubQ:

The name of the broker's queue from which non-durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the Web container.

The name of the broker queue from which non-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
Default	Null
Range	1 through 48 ASCII characters

Broker Version:

Specifies whether the message broker is provided by the MQSeries MA0C SupportPac or newer versions of WebSphere message broker products.

Data type	Enum
Units	Not applicable
Default	Advanced
Range	**Advanced** The message broker is provided by newer versions of WebSphere message broker products (MQ Integrator and MQ Publish and Subscribe) **Basic** The message broker is provided by the MQSeries MA0C SupportPac (MQSeries - Publish/Subscribe)

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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 accectable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

MQSeries 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 MQSeries 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 > **JMS Providers** > *JMS provider instance*. Right-click **MQ Queue Destinations** and click **New**. The following fields appear on the **General** tab.

A queue destination configures the properties of a JMS queue. Connections to the queue are created by the associated queue connection factory for the MQSeries JMS provider. A queue for use with the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

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
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type	Enum
Units	Not applicable
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. **Non persistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the* ***Specified expiry*** *property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds
Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • 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 MQSeries 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
Default	Null
Range	A valid MQSeries Queue Manager name, as 1 through 48 ASCII characters

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

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Data type	Enum
Units	Not applicable
Default	NORMAL
Range	**NORMAL** Normal integer encoding is used. **REVERSED** Reversed integer encoding is used. For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Data type	Enum
Units	Not applicable
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:

If native encoding is not enabled, select the type of floating point encoding.

Data type	Enum
Units	Not applicable
Default	IEEENORMAL

Range	**IEEENORMAL** IEEE normal floating point encoding is used. **IEEEREVERSED** IEEE reversed floating point encoding is used. **S390** S390 floating point encoding is used. For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Native encoding:

Select this checkbox to indicate that the queue destination should use native encoding (appropriate encoding values for the Java platform).

Data type	Enum
Units	Not applicable
Default	Cleared
Range	**Cleared** Native encoding is not used, so specify the properties below 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 MQSeries *Using Java* document.

Target client:

Whether the receiving application is JMS-compliant or is a traditional WebSphere MQ application

Data type	Enum
Units	Not applicable
Default	MQSeries
Range	**MQSeries** The target is a non-JMS, traditional WebSphere MQ application. **JMS** The target is a JMS-compliant application.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime 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.

MQSeries 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 MQSeries 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 > **JMS Providers** > *JMS provider instance*. Right-click **MQ 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. Connections to the topic are created by the associated topic connection factory. A topic for use with the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

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. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Persistence:

Specifies whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application.

Data type	Enum
Units	Not applicable
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. **Non persistent** 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
Units	Not applicable
Default	APPLICATION_DEFINED

Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the* ***Specified priority*** *property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type the message priority for this queue, in the range 0 (lowest) through 9 (highest).

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Specifies whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout).

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the* ***Specified expiry*** *property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type the number of milliseconds (greater than 0) after which messages on this queue expire.

Data type	Integer
Units	Milliseconds
Default	Null

Range	Greater than or equal to 0 • 0 indicates that messages never timeout • 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 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

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Data type	Enum
Units	Not applicable
Default	NORMAL
Range	**NORMAL** Normal integer encoding is used. **REVERSED** Reversed integer encoding is used. For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Data type	Enum
Units	Not applicable
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:

If native encoding is not enabled, select the type of floating point encoding.

Data type	Enum
Units	Not applicable

Default	IEEENORMAL
Range	**IEEENORMAL** IEEE normal floating point encoding is used. **IEEEREVERSED** IEEE reversed floating point encoding is used. **S390** S390 floating point encoding is used. For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Native encoding:

Select this check box to indicate that the queue destination should use native encoding (appropriate encoding values for the Java platform).

Data type	Enum
Units	Not applicable
Default	Cleared
Range	**Cleared** Native encoding is not used, so specify the properties above 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 MQSeries *Using Java* document.

BrokerDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved.

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
Default	Null
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.

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
Default	Null
Range	1 through 48 ASCII characters

Target Client:

Specifies whether the receiving application is JMS-compliant or is a traditional MQSeries application.

Data type	Enum
Units	Not applicable
Default	MQSeries
Range	**MQSeries** The target is a non-JMS, traditional MQSeries application. **JMS** The target is a JMS-compliant application.

Custom Properties:

Specifies the name-value pairs for setting additional properties on the object that is created at runtime 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 connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected JMS connection factory for use with the associated JMS provider. These configuration properties control how connections are created to the associated JMS destination.
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 > **JMS Providers** > *new JMS Provider instance*. Right click **JMS Connection Factories** > click **New**. The following fields appear on the **General** tab.

A 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 WebSphere JMS provider or the MQSeries 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. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

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's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String

Connection Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for pub/sub).

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 runtime 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 > **JMS Providers** > *new JMS Provider instance*. Right click **JMS Destinations** > click **New**. The following fields appear 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 internal WebSphere JMS provider or MQSeries 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

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's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String

Destination Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for pub/sub).

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 runtime 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:
  - JMS Provider Properties page: name, and at least one protocol provider
  - JMS Connection Factory Properties page: name, jndiName, destination type
  - JMS Destination Properties page: name, jndiName, destination type
- Special cases:
  - The destination type must be `QUEUE`, or `TOPIC`.
- Example:

```
<resources.jms:JMSProvider xmi:id="JMSProvider_3" name="genericJMSProvider:name"
description="genericJMSProvider:description"
externalInitialContextFactory="genericJMSProvider:contextFactoryClass"
externalProviderURL="genericJMSProvider:providerUrl">
<classpath>genericJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms:GenericJMSDestination"
xmi:id="GenericJMSDestination_1" name="jmsDestination:name"
jndiName="jmsDestination:jndiName" description="jmsDestination:description"
externalJNDIName="jmsDestination:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_15">
<resourceProperties xmi:id="J2EEResourceProperty_17" name="jmsDestination:custonName"
value="jmsDestination:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms:GenericJMSConnectionFactory"
xmi:id="GenericJMSConnectionFactory_1" name="jmsCF:name" jndiName="jmsCF:jndiName"
description="jmsCF:description" userID="jmsCF:user" password="{xor}NTIsHBllMT4yOg=="
externalJNDIName="jmsCF:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_16">
<resourceProperties xmi:id="J2EEResourceProperty_18" name="jmsCF:customName"
value="jmsCF:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_17">
<resourceProperties xmi:id="J2EEResourceProperty_19"
name="genericJMSProvider:customName" value="genericJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Configuring new connection factories for application clients

During this task, you create a new 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. Do one of the following:
   - 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 folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS connection factory properties in the resulting property dialog.
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

During this task, you create 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. Do one of the following:
   - Configure a new JMS provider.
   - Click an existing JMS provider.
2. Expand the JMS provider to view its **JMS Destinations** folder.
3. Click the folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS destination properties in the resulting property dialog.
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, transport type
  - MQ Topic Connection Factory Properties page: name, jndiName, broker version
  - MQ Queue Factory Properties page: name, jndiName, persistence, priority, expiry, baseQueueName, targetClient
  - MQ Topic Factory Properties page: name, jndiName, persistence, priority, expiry, baseQueueName, 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`, `S390`.
  - The target client must be `JMS` or `MQ`.
  - On the MQ Queue Connection Factory Properites page, only set the queueManager, host, and portWhen (required) fields if the transport type is `CLIENT`.
  - On the MQ Topic Connection Factory Properites page, only set the queueManager, host, and port (required) fields if the transport type is `CLIENT`.
  - On the 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 nativeEncoding.
  - 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 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 expiry 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}Mi4OHBllMT4yOg=="
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}Mi4LHBllNTE7NhE+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 to configure WAS Queue connection factory, WAS Topic connection factory, WAS Queue destination factory, and WAS Topic destination factory settings.

- Required fields:
  - JMS Provider Properties page: name
  - WAS Queue Connection Factory Properties page: name, jndiName, node

  - WAS Topic Connection Factory Properties page: name, jndiName, node, port
  - WAS Queue Factory Properties page: name, jndiName, node, persistence, priority, expiry
  - WAS Topic Factory Properties page: name, jndiName, topic name, persistence, priority, expiry
- Special cases:
  - The port 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 must be `APPLICATION_DEFINED`, or `SPECIFIED`.
  - The expiry 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 priority is set to `SPECIFIED` .
  - On the WAS Topic Factory Properties, and the WAS Queue Factory Properties pages, the specified expiry entry field must be an 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}KD4sDhwZZSosOi0="
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 tool and open the EAR file for which you want to configure the new 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 folder called **Resource Environment Providers**. Do one of the following:
   - Right-click the folder and click **New Provider**.
   - Click **Edit > New** on the menu bar.
5. Configure the JMS provider properties in the resulting property dialog.
6. Click **OK** when finished.
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 JAR file. Right-click **Resource Environment Providers** > and click **New**. The following fields appear 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 runtime 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 folder. Complete one of the following actions:
   - Right-click the folder and select **New Factory**.
   - Click **Edit > New** on the menu bar.
6. Configure the data source properties in the resulting property dialog.
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 (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 runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Managing application clients

Perform the following tasks after deploying application clients.

Note: 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.

1. Start the tool and open the EAR file containing the data source or data source provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file 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. Do one of the following:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the resulting property dialog. For detailed field help, go to:
   - Data source provider properties
   - Data source properties
5. Click **OK** when finished.
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 EAR file containing the URL or URL provider. The EAR file contents display in a tree view.
2. Select from the tree the 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. Do one of the following:
   a. Right-click the object and click **Properties**
   b. Click **Edit** > **Properties** on the menu bar.
5. Update the properties in the resulting property dialog.
6. Click **OK** when finished.
7. Click **File** > **Save** to save your changes on the menu bar.

Updating mail session configurations for application clients

During this task, you update the configuration of an existing JavaMail session.

Note:

You cannot update the name of the default JavaMail provider. Also, you cannot delete the default JavaMail provider from the tree.

1. Start the tool and open the EAR file containing the JavaMail session. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the JavaMail session to update.
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. Do one of the following:
   a. Right-click the object and click **Properties**
   b. Click **Edit** > **Properties** from the menu bar.
5. Update the properties in the resulting property dialog.

6. Click **OK** when finished.
7. Select **File** > **Save** from the menu bar to save your changes.

Updating Jave 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 EAR file containing the JMS provider, connection factory, or destination. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the JMS provider, connection factory, or destination to update.
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:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the resulting property dialog. For detailed field help, see:
   - JMS provider properties
   - WAS Queue connection factory properties
   - WAS Topic connection factory properties
   - WAS Queue destination properties
   - WAS Topic destination properties
5. Click **OK**.
6. Click **File > Save** to save your changes.

Updating MQ Java Message Service provider, MQ connection factories, and MQ destination configurations for application clients

During this task, you will update the configuration of an existing MQ JMS provider, MQ connection factory, or MQ destination.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file containing the MQ JMS provider, MQ connection factory, or MQ destination. The EAR file contents are displayed in a tree view.
3. Select the JAR file containing the MQ JMS provider, MQ connection factory, or MQ destination to update.
4. Expand the JAR file to view its contents until you locate the particular MQ JMS provider, MQ connection factory, or MQ destination that you want to update. Complete one of the following actions:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
5. Update the properties in the resulting property dialog. 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 EAR file containing the Resource Environment Entry or Resource Environment Provider. The EAR file contents display in a tree view.
2. Select from the tree the 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. Do one of the following:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the resulting property dialog. 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 article is to help you configure Resource Environment settings.

- Required fields:
  – Resource Environment Provider page: name
  – Resource Environment Entry page: name, jndiName
- 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 article 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

Note: This task only applies to J2EE application clients.

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file from which you want to remove an object. The EAR file contents display in a tree view. If you already have an EAR file open, and have made some changes, click **File** > **Save** to save your work before preceding to delete an object.
2. Locate the object that you want to remove in the tree.
3. Right-click the object, then click **Delete**.
4. Click **File** > **Save**.

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.

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).

Note: 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 the Assembly Toolkit.
   - 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 code ends.
2. Pass parameters to the **launchClient** command. You can pass parameters 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.

   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:
   a. The command line
   b. A properties file
   c. System properties

The parameters are resolved in the order listed above, with command line values having the highest priority and system properties the lowest. This prioritization allows you to set and override default values.

3. Specify the server name. By default, the **launchClient** command uses the environment variable `COMPUTERNAME` 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` 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.

   Note: 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 security is not enabled, the server ignores the security request, and the application client works as expected.

You can store launchClient values in a properties file, 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 command line syntax for the Java TM2 Platform, WebSphere Application Server Enterprise (J2EE) launchClient tool. You can use the launchClient command from a node within a Network Deployment environment. **However, do not attempt to use the launchClient command from Deployment Manager.**

The command line invocation syntax for the launchClient tool follows:

```
launchClient [<userapp.ear> |-help|-?] [-CCname=value] [app args]
```

where *userapp.ear* is the path and the name of the EAR file that contains the application client, *name* is the name of the parameter, *value* is the value to which the parameter ID is set, and *app args* are arguments that pass to the application client.

To print the usage information, the first parameter must be a path and a name to an EAR file, `-help`, or `-?`. All other parameters are optional and can appear in any order. The application client run time ignores any optional parameters that do not begin with a `-CC` prefix, and passes them to the application client.

Parameters

Supported arguments include:

-CCsoapConnectorPort
The 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 not used. If you want to access classes that are not in the EAR file or part of the resource class paths, specify the appropriate class path here. Multiple paths can be concatenated.

-CCjar
The name of the client 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 of the local host if the `-CCBootstrapHost` parameter is not specified.

-CCadminConnectorPort
Indicates the port number that the administrative client function should use. The default value is `8880` for SOAP connections and `2809` for RMI connections.

-CCadminConnectorType
Specifies how the administrative client should connect 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.ofchoice.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 CORBA object URL or an 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. An example of a CORBA object URL specifying multiple systems follows:

```
-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 Service. The default is `false`.

-CCtracefile
The name of the file to write trace information. The default is to output to the console.

-CCpropfile
: 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
: 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
: 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 `=` 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 `System.exit()` after the client application completes. The default is `false`.

-CCdumpJavaNameSpace
: Prints out the Java portion of the WebSphere Application Server Java Naming and Directory Interface (JNDI) name space. The `true` value uses the short format which prints out the binding name and the type of the object bound at that location. The `long` value uses the long format which 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.

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 launchClient is called, it extracts the EAR file to a random directory name in the temporary directory on your hard drive. Then it sets up the thread ClassLoader to use the extracted EAR file directory and JAR files included in the `Manifest.mf` client 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 launchClient 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, launchClient 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, make sure 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.

Example: Using a Java 2 security manager with a J2EE application client

The launchClient command provides several parameters to control the use of a Java 2 security manager. By default the launchClient command does not enable nor run with a Java 2 security manager. To enable the Java 2 security manager, add the following parameter to your launchClient command:

```
-CCsecurityManager=enable
```

For example:

```
launchClient myear.ear -CCsecurityManager=enable
```

- When the security manager is enabled, the launchClient uses by default the `java.lang.SecurityManager` class and the `<WAS_HOME> /properties/client.policy` policy file. This policy file is configured to provide the standard permissions as described in the J2EE specification for J2EE application clients and applets. If your application receives a `java.security.AccessControlException`, you must add additional permissions to the `client.policy` file. For more information on adding additional permissions, see configuring client.policy files and AccessControlException.
- You can override the default security manager class by specifying the `-CCsecurityMgrClass` parameter and the default policy file using the `-CCsecurityMgrPolicy` parameter. For more information, see launchClient tool.
- If you invoke Java to start the launchClient class, it is recommended that you do not use the `-Djava.security.manager` parameter to enable the Java 2 security manager. Using this parameter causes the Java 2 security manager to be enabled prior to initialization of the J2EE application client runtime. The necessary permissions are not granted and your application may receive `java.security.AccessControlExceptions`.
- When the J2EE application client runtime is initialized, the Enterprise Archive (EAR) file that you specified is extracted to a random subdirectory in your users temporary directory location.

  Note: If the EAR file is a set of directories and subdirectories, then it is used in place and not expanded.

  The J2EE application client runtime sets the `com.ibm.websphere.client.applicationclient.archivedir` system property to the directory location of the EAR file. The `client.policy` file uses this system property to inform the security manager of the location of your application client codebase and to assign the configured permissions to that codebase. This occurs when the security manager is enabled. If the security manager is enabled at the time Java is started, then this system property is not set, the codebase is unknown, and the permissions can not be granted.
- It is recommended that you enable the security manager with the J2EE application client runtime. Use the following parameter: `-CCsecurityManager=enable`.

Example: Enabling Java 2 security prior to J2EE application client runtime initialization

To enable the Java 2 security prior to the J2EE application client runtime initialization, set the `com.ibm.websphere.client.applicationclient.archivedir` system property. Perform the following steps:

1. Set the system property to directory where the Enterprise Archive (EAR) should be expanded to, for example:

   ```
   -Dcom.ibm.websphere.client.applicationclient.archivedir=c:\myear1 (windows)
   ```

   ```
   -Dcom.ibm.websphere.client.applicationclient.archivedir=/usr/mrear1 (Unix)
   ```

2. Set the `java.security.policy` system property to use the `<WAS_HOME>/properties/client.policy` file, for example:

   ```
   -Djava.security.policy=%WAS_HOME%\properties\client.policy (Windows)
   -Djava.security.policy=$WAS_HOME/properties/client.policy (Unix)
   ```

- Setting the `com.ibm.websphere.client.applicationclient.archivedir` has the following effects:
  - If the directory does not exist or it is empty, the EAR file is extracted to that directory.
  - The EAR file is reused if it was previously extracted. This occurs even if the EAR file specified on the command line is different.
  - The security manager will grant the permissions from the `client.policy` file to that directory and all its subdirectories.
- There are two types of EAR files. The first type of EAR file is a single file that contains all the enterprise application files. The second type is a set of directories and subdirectories. The following only applies if you are using the single file form:
  - If you need to update your EAR file, delete the directory first.
  - The new EAR file will be extracted to the directory the next time you run. If you do not delete the directory or change the system property value to point to a different temporary directory, the currently extracted EAR file will be reused, and your changed EAR file will not be used.
- When specifying the `com.ibm.websphere.client.applicationclient.archivedir` property, ensure that the directory you specify is unique for each EAR file that you use. For example, do not point `MyEar1.ear` and `MyEar2.ear` files to the same directory. You also must create all directories up to, but not including, the last directory. For example, if you set the following:

  ```
  com.ibm.websphere.client.applicationclient.archivedir=/usr/myears/myear1
  ```

  then *usr* and *myears* must exist, but *myear1* does not have to exist prior to running the launchClient class.

Application client troubleshooting tips

This section provides some debugging tips for resolving common 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; the actual error you receive can be slightly different than what is shown here. Also, it can be useful to rerun the **launchClient** command specifying the `-CCverbose=true` option. This option provides additional information when the J2EE application client run time is initializing

Error: java.lang.NoClassDefFoundError

Explanation	This exception is thrown when Java code cannot load the specified class.
Possible causes	• Invalid or non-existent class • Classpath problem • Manifest problem

Recommended response

Check to determine if the specified class exists in a JAR file within your 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
```

ensure the class HelloHome exists in one of the JAR files in your EAR file. If it exists, ensure the path for the class is WebSphereSamples.HelloEJB.

If both the class and path are correct, then it is a classpath 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 Assembly Tool and click on 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 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 is a very difficult situation 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 problem, review the classpaths specified with the -CCclasspath option and the classpaths configured with the Application Client Resource Configuration Tool. Look for classes that also exist in the EAR file. You must resolve the situation where one of the classes is found on the file system instead of in the `.ear` file. Remove entries from the classpaths, or include the `.jar` files and classes in the `.ear` file instead of referencing them from the file system.

If you use the -CCclasspath parameter or resource classpaths in the Application Client Resource Configuration Tool, and you have configured multiple JAR files or classes, verify they are separated with the correct character for your operating system. Unlike the classpath field in the Application Assembly Tool, these classpath fields use platform-specific separator characters, usually a colon (on UNIX platforms) or a semi-colon (on Windows systems).
Note: The system classpath is not used by the Application Client run time if you use the launchClient batch or shell files. In this case, the system classpath would not cause this problem. However, if you load the launchClient class directly, you do have to search through the system classpath 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 enterprise bean 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 WebSphere **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 Assembly Tool 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. They must match.

Error: javax.naming.ServiceUnavailableException: A communication failure occurred while attempting to obtain an initial context using the provider url: ″iiop://[invalidhostname]″. Make sure that the host and port information is correct and that the server identified by the provider URL is a running name server. If no port number is specified, the default port number 2809 is used. Other possible causes include the network environment or workstation network configuration. Root exception is org.omg.CORBA.INTERNAL: JORB0050E: In Profile.getIPAddress(), InetAddress.getByName[invalidhostname] threw an UnknownHostException. minor code: 4942F5B6 completed: Maybe

Explanation	This exception occurs when you specify an invalid host server name.
Possible causes	• Incorrect host server invoked • Invalid host server name
Recommended response	Run the **launchClient** command again and specify the correct name of your host server with the -CCBootstrapHost parameter.

Error: javax.naming.CommunicationException: Could not obtain an initial context due to a communication failure. Since no provider URL was specified, either the bootrap host and port of an existing ORB was used, or a new ORB instance was created and initialized with the default bootstrap host of ″localhost″ and the default bootstrap port of 2809. Make sure the ORB bootstrap host and port resolve to a running name server. Root exception is org.omg.CORBA.COMM_FAILURE: WRITE_ERROR_SEND_1 minor code: 49421050 completed: No

Explanation	This exception occurs when you run the **launchClient** command to a host server that does not have the Application Server started. You also receive this exception when you specify an invalid host server name. This situation might occur if you do not specify a host server name when you run launchClient. The default behavior is for launchClient to run to localhost, 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 computer 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 Assembly Tool 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 make sure the Resource Reference has client configuration information and the name of the Resource Reference exactly matches the JNDI name of the client configuration. If it is correct, you might have a class loader issue.

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 EJB client side bindings. These are class files whose names end in _Stub and _Tie. If these files are not present, then use the Application Assembly Tool to generate the binding classes. For more information, see article Generating deployment code for modules. If the binding classes are in the EJB `.jar` file, then you might have a class loader issue.

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 the 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 code. However, since there is an AWT thread, the Java Virtual Machine code will not end until `System.exit` is called.
Possible causes	The Java Virtual Machine 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 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.

Chapter 7. Using Web services based on Web Services for J2EE

Decide if a Web service implementation benefits your business process.

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 1.3 and non-J2EE platforms. Web services benefit your e-business solution by integrating these enterprise systems, especially systems that have developed over a long period of time.

Using Web services makes most sense if your application's 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.

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.

To use Web services applications with WebSphere Application Server:
1. Plan to use Web services. Review the Universal Description, Discovery, and Integration (UDDI), Web services gateway and Web Services Invocation Framework concepts to learn how these components can make your Web services plan more robust.
2. Migrate existing Web services.
3. Develop Web services.
4. Assemble Web services.
5. Deploy Web services.
6. Secure Web services.
7. Tune Web services.
8. Troubleshoot Web services.

The following is an example of how a business might use Web services.

The owner of a flower shop wants to start receiving orders from customers through the Web. She starts her venture by finding wholesale flower suppliers, pricing their product, and completing contracts for future flower orders.

Using Web services, the flower shop owner can find wholesale flower suppliers. One way she finds new suppliers is to use a UDDI registry to search for potential suppliers. She chooses the suppliers and the registry sends back information on how to contact the flower distributors that meet her criteria.

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 email, 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 helps her get price lists. The WSDL file also specifies the Universal Resource Locator (URL) where the request is to be sent.

The flower shop owner now has to compare the prices she received back from each supplier, decide which suppliers she is going to do business with, and make arrangements for future orders to be filled. The ground work has been laid for the flower shop to 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 to be published to her Web site and she needs to provide a mechanism for customers to order flowers.

The flower supplier's Web services clients 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. Web services can also be used by the supplier to send invoices for orders and by the flower shop to pay the supplier's invoices. Processes that previously required forms to be filled out manually, and faxed or mailed, 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. There is no merchandise maintenance overhead, but the flower shop can offer their customers products that they otherwise might not have. Selling flowers through the Web increases capital for the flower shop without overhead of another store or money invested into additional product.

For a more detailed scenario, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporated the Web services concept.

Web services

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 a given URL using SOAP messages over a Hypertext Transport Protocol (HTTP) or Java Messaging Service (JMS) transport. 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 or purchase orders.

A Web service can be the service itself or the client that accesses the service.

Web services reflect a new, 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 available applications to accomplish some task. Web services deliver interoperability, for example, the ability 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.

The key components of a Web service are:
- Web Services Description Language (WSDL)

WSDL is the XML-based file that describes the Web service and allows the Web service request to bind to the service.

- SOAP

  SOAP is the XML-based protocol that allows the Web service request to invoke the service.
- Universal Description, Discovery and Integration Protocol (UDDI)

  UDDI is the registry that hosts the service broker. UDDI is similar to the Yellow Pages in a phone book.

Web services are Web applications that allow you to be more flexible in your business processes by integrating with applications that otherwise would 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 exploded with the birth of the Internet. Before, you would visit your library, search the collections and check out your books. If you didn't find the book 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 could 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 creates a convenience for you, as well as freeing the librarian to do other business tasks. For a more detailed scenario, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporated the Web services concept.

SOAP

SOAP is a specification for 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. Then main goal of its design is to be simple and extensible. A SOAP message is used to request a Web service.

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.

SOAP is an XML-based protocol that consists of three parts: an *envelope* that defines a framework for describing message content and process instructions, a set of *encoding rules* for expressing instances of application-defined data types, and a *convention* for representing remote procedure calls and responses.

SOAP is transport protocol-independent and can be used in combination with a variety of protocols. In Web services that are developed and implemented for use with WebSphere Application Server, SOAP is used in combination with HyperText Transport Protocol (HTTP), HTTP extension framework, and Java Messaging Service (JMS). SOAP is also operating system independent and not tied to any programming language or component technology.

Due to these characteristics, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages. Also, both server and client sides can reside on any suitable platform.

For more information about SOAP, see Web services: Resources for learning.

Planning to use Web services based on Web Services for J2EE

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

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

To plan to use Web services based on Web Services for J2EE:

1. Design Web services to fit your e-business solution. Consider what you want to accomplish by using Web services, how Web services fit into your current topology, applications and programming model. Decide how the Web services process requests on the server and how the clients manage and use the Web service.

   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.

   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.
2. 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 enterprise JavaBean (EJB), you can choose different tasks respective to your resources. You can also use the the WebSphere Application Developer Studio to complete development and implementation tasks.

   See Developing Web services based on Web Services for J2EE for information about developing Web services based on the Java language through the WebSphere Application Server. To read more about the WebSphere Application Developer Studio see the topic Web services development in the WebSphere Application Server Developer Studio InfoCenter.
3. Install WebSphere Application Server.
4. Review Web services Samples.

Develop a Web service.

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 have longed to integrate existing systems in order 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) interactions 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 of resources and data, there is a need 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, to be used by different groups of people in 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 a SOA

In order to efficiently use a SOA, you must abide by the following requirements:

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

  The most important basis for a simple integration between applications on different platforms is a communication protocol, which 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 allow a convenient integration at design time or even system run time, a search mechanism is required to retrieve suitable services. The services should be classified 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

Web services implement 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 play one or more of three roles:

- **Service provider**

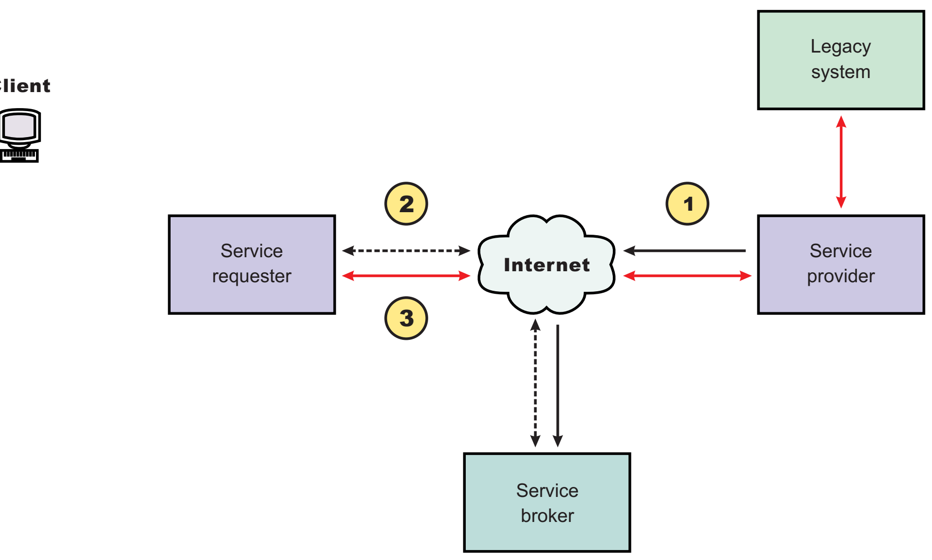
  The service provider creates a Web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or, if they are free, how to exploit them for other value. The provider also has to decide what category the service should be listed 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 about 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, the amount of the offered information has to be decided. 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. There are also brokers that catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, number of listings or accuracy of the listings.

- **Service requestor**

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



Characteristics of the Web service architecture

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

- A client is not coupled to a server, but 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, allowing new interfaces to 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 Hyper Text Transport Protocol (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 Web services-enable an existing application 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 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. It leverages the existing infrastructure. Some additional standards that are required to do so include SOAP, 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 be changed in order to be Web service enabled. In this redbook, however, we assume that Java is the implementation language for both the client and the server side of the Web service.

- **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. Therefore, vendor independence and interoperability are realistic goals this time.

- **Web services are dynamic.**

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

- **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 shortens development time and enables best-of-breed implementations.

- **Web services build on proven mature technology.**

  There are a lot of commonalities, as well as a few fundamental differences to other distributed computing frameworks. For example, the transport protocol is text based and not binary.

- **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 allows a more flexible re-configuration for an integration of the services in question.

- **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 be integrated 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 allow 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 can be achieved by dynamically linking internal applications to new partners and suppliers, to offer their services to complement internal services.

To see how these models are implemented using all aspects of Web services, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporate the Web services concept.

Migrating Apache SOAP Web services to Web Services for J2EE

If you have used Web services based on Apache SOAP in WebSphere Application Server Version 4.0.x through Version 5.0.2, and now want to develop and implement Web services based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification, you need to migrate your Version 4.0 and 5.0 client applications.

To migrate these client applications according to the Web Services for J2EE standards:

1. Plan your migration strategy. There are two ways you can port an Apache SOAP client to Java API for XML-based RPC (JAX-RPC) Web services client:
   - If you have, or can create, a Web Services Description Language (WSDL) document for the service, consider using the **WSDL2Java** command tool to generate bindings for the Web service. It is more work to adapt an Apache SOAP client to use the generated JAX-RPC bindings, but the resulting client code is more robust and easier to maintain. To follow this path, see Develop a Web services client based on Web Services for J2EE.
   - If you do not have a WSDL document for the service, do not expect the service to change, and you want to port the Apache SOAP client with a minimal work, you can convert the code to use the JAX-RPC dynamic invocation interface (DII), which is similar to the Apache SOAP APIs. The DII APIs do not use WSDL or generated bindings.

   You should be aware that since JAX-RPC does not specify a framework for user-written serializers, the JAX-RPC does not support the use of custom serializers. If your application cannot conform to the default mapping between Java, WSDL, and XML supported by WebSphere Application Server, you should not attempt to migrate the application. The remainder of this topic assumes that you have decided to use the JAX-RPC DII APIs.
2. Review the GetQuote sample. There is a Web services migration sample in the Samples Gallery. This sample is located in the `GetQuote.java` file, originally written for Apache SOAP, and includes an explanation about the changes needed to migrate to the JAX-RPC DII interfaces.
3. Convert the client application from Apache SOAP to JAX-RPC DII The Apache SOAP API and JAX-RPC DII API structures are similar. You can instantiate and configure a call object, set up the parameters, invoke the operation, and process the result in both. You can create a generic instance of a Service object with

   ```
   javax.xml.rpc.Service service =
      ServiceFactory.newInstance().createService(new QName(""));
   ```

   in JAX-RPC.

   a. Create the call object. An instance of the call object is created by

      ```
      org.apache.soap.rpc.Call call = new org.apache.soap.rpc.Call ()
      ```

      in Apache SOAP.

      An instance of the call object is created by

      ```
      java.xml.rpc.Call call = service.createCall();
      ```

      in JAX-RPC.

b. Set the endpoint URI. The target URI for the operation is passed as a parameter to
```
call.invoke:  call.invoke("http://...", "");
```

   in Apache SOAP.

   The setTargetEndpointAddress method is used as a parameter to
```
call.setTargetEndpointAddress("http://...");
```

   in JAX-RPC.

   Apache SOAP has a setTargetObjectURI method on the call object that contains routing information for the request. JAX-RPC has no equivalent method. The information in the targetObjectURI is included in the targetEndpoint URI for JAX-RPC.

c. Set the operation name. The operation name is configured on the call object by
```
call.setMethodName("opName");
```

   in Apache SOAP.

   The setOperationName method, which accepts a `QName` instead of a `String` parameter, is used in JAX-RPC as follows:
```
call.setOperationName(new javax.xml.namespace.Qname("namespace", "opName"));
```

d. Set the encoding style. The encoding style is configured on the call object by
```
call.setEncodingStyleURI(org.apache.soap.Constants.NS_URI_SOAP_ENC);
```

   in Apache SOAP.

   The encoding style is set by a property of the call object
```
call.setProperty(javax.xml.rpc.Call.ENCODINGSTYLE_URI_PROPERTY,
   "http://schemas.
xmlsoap.org/soap/encoding/");
```

   in JAX-RPC.

e. Declare the parameters and set the parameter values. Apache SOAP parameter types and values are described by parameter instances, which are collected into a Vector and set on the call object before the call, for example:
```
Vector params = new Vector ();
params.addElement (new org.apache.soap.rpc.
     Parameter(name, type, value, encodingURI));
// repeat for additional parameters...
call.setParams (params);
```

   For JAX-RPC, the call object is configured with parameter names and types without providing their values, for example:
```
call.addParameter(name, xmlType, mode);
// repeat for additional parameters
call.setReturnType(type);
```

   Where
   - *name* (type `java.lang.String`) is the name of the parameter
   - *xmlType* (type `javax.xml.namespace.QName`) is the XML type of the parameter
   - *mode* (type `javax.xml.rpc.ParameterMode`) the mode of the parameter, for example, IN, OUT, or INOUT

f. Make the call. The operation is invoked on the call object by
```
org.apache.soap.Response resp = call.invoke(endpointURI, "");
```

in Apache SOAP.

The parameter values are collected into an array and passed to `call.invoke` as follows:

```
Object resp = call.invoke(new Object[] {parm1, parm2,...});
```

in JAX-RPC.

g. Check for faults. You can check for a SOAP fault on the invocation by checking the response:

```
if resp.generatedFault then {
org.apache.soap.Fault f = resp.getFault;
f.getFaultCode();
f.getFaultString();
}
```

in Apache SOAP.

A `java.rmi.RemoteException` is thrown in JAX-RPC if a SOAP fault occurs on the invocation.

```
try {
... call.invoke(...)
} catch (java.rmi.RemoteException) ...
```

h. Retrieve the result. In Apache SOAP, if the invocation was successful and returns a result, it can be retrieved from the Response object:

```
Parameter result = resp.getReturnValue(); return result.getValue();
```

In JAX-RPC, the result of invoke is the returned object when no exception is thrown:

```
Object result = call.invoke(...);
 ...
return result;
```

Developing a Web services client based on Web Services for J2EE.

Test the Web services-enabled clients.

Developing Web services based on Web Services for J2EE

This topic explains how to develop a Web service using the Web Services for Java 2, 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.

For more information about when and how you should to use Web services see Using Web services based on Web Services for J2EE. You can read about several concepts, including what is Web services, SOAP, WSDL, Web Services for J2EE and Java API for XML-based remote procedure call (JAX-RPC). If you would like to review a scenario where Web services are used, see Web services scenario: Overview.

WebSphere Application Server uses Web services standards developed for the Java language under the Java Community Process (JCP). These standards include the Web Services for J2EE and JAX-RPC specifications.

You can also use the WebSphere Studio Application Developer Version 5.1 graphical user interface development tools to develop Web services that integrate with WebSphere Application Server.

Before you develop a Web service you need to Set up a Web services development and unmanaged client execution environment .

Follow the Example: Developing Web services based on Web Services for J2EE for a step-by-step look at this task.

You can develop a Web service based on Web Services for J2EE in one of four ways:

1. Develop a Web service using a Java bean.
2. Develop a Web service using a stateless session enterprise bean.
3. Develop a Web service with an existing WSDL file using a Java bean.
4. Develop a Web service with an existing WSDL file using a stateless session enterprise bean.

Assemble the Web service.

Example: Developing Web services based on Web Services for J2EE

This example takes you through the steps to develop a Web service from an enterprise JavaBean (EJB) implementation. The development process is based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

1. **Select the EJB or Java bean 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 or a Java bean without client-specific state, since 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 EJB must not have a transaction attribute of Mandatory, because there is no standard for Web services transactions at this time.

     A Java bean in a Web container requires the following:

     – 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 Java bean implementation, develop a Service Endpoint Interface for a Java bean implementation.
3. Develop a 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 Java bean implementation, develop Web services deployment descriptor templates for a Java bean implementation.
5. **Configure the deployment descriptors.**

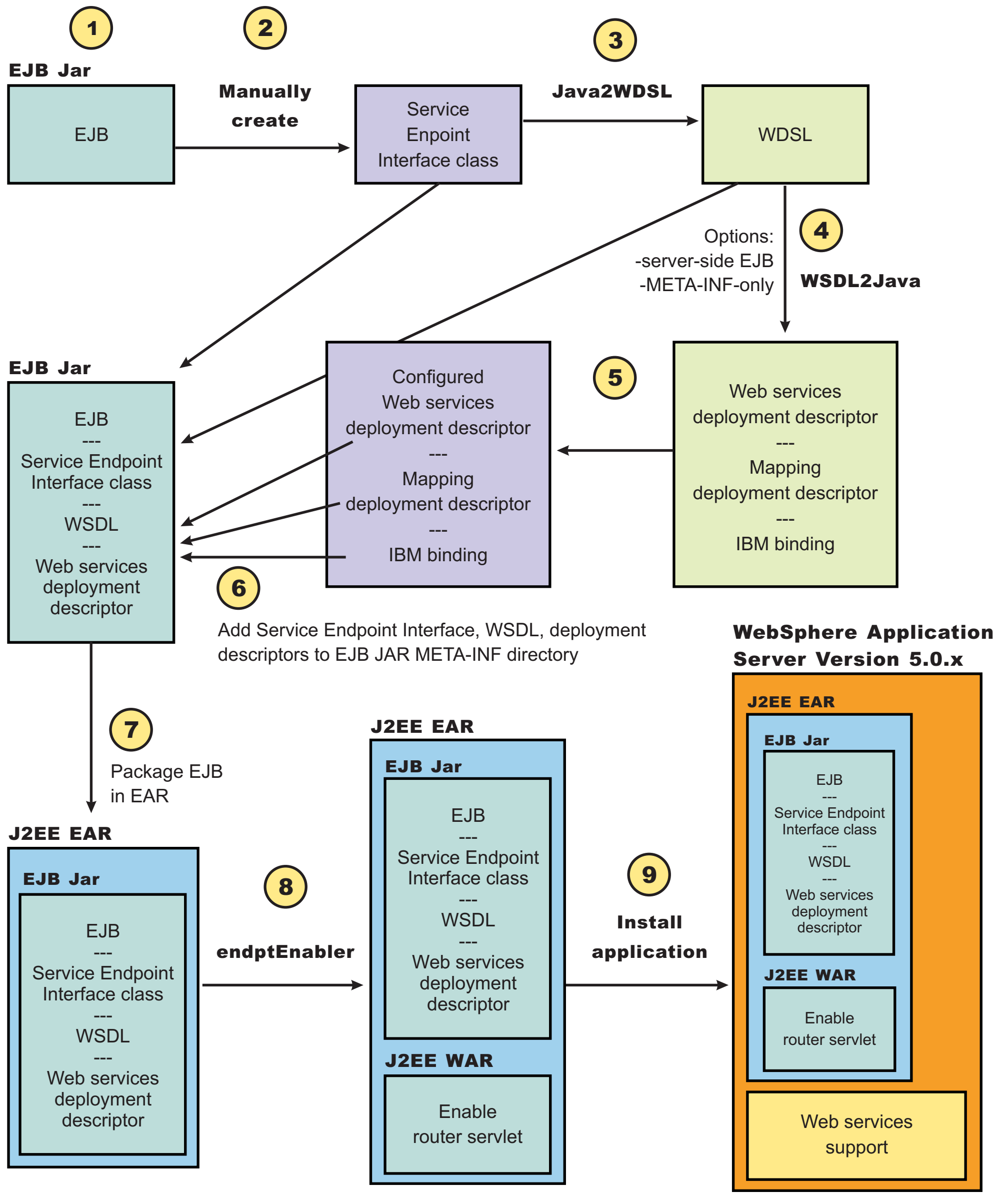
   By setting the `ejb-link` or `servlet-link` values of the `service-impl-bean` elements you can link to the EJB or Java bean that implements the service.

   Configure the webservices.xml deployment descriptor.

   Configure the ibm-webservices-bnd.xmi deployment descriptor.
6. Assemble a JAR file or Assemble a WAR file.
7. Assemble an EAR file from a Jar file or Assemble an EAR file from a WAR file.
8. Enable the Web service-enabled EAR file.

   This step only applies if you are using an EJB implementation.

9. Deploy the Web service application.
10. Publish the WSDL file.



1
EJB Jar
EJB
2
Manually create
Service Enpoint Interface class
3
Java2WDSL
WDSL
Options:
-server-side EJB
-META-INF-only
4
WSDL2Java
EJB Jar
EJB --- Service Endpoint Interface class --- WSDL --- Web services deployment descriptor
Configured Web services deployment descriptor --- Mapping deployment descriptor --- IBM binding
5
Web services deployment descriptor --- Mapping deployment descriptor --- IBM binding
6
Add Service Endpoint Interface, WSDL, deployment descriptors to EJB JAR META-INF directory
WebSphere Application Server Version 5.0.x
7
Package EJB in EAR
J2EE EAR
EJB Jar
EJB --- Service Endpoint Interface class --- WSDL --- Web services deployment descriptor
8
endptEnabler
J2EE EAR
EJB Jar
EJB --- Service Endpoint Interface class --- WSDL --- Web services deployment descriptor
J2EE WAR
Enable router servlet
9
Install application
J2EE EAR
EJB Jar
EJB --- Service Endpoint Interface class --- WSDL --- Web services deployment descriptor
J2EE WAR
Enable router servlet
Web services support

Web Services for J2EE

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 Web Services for J2EE is the Java Specification Requirements (JSR) 109. The specification includes open standards for developing and implementing Web services.

WebSphere Application Server Versions 5.0.2 and later use Web Services for J2EE as the standard for developing and implementing Web services. Before Version 5.0.2, WebSphere Application Server developed and implemented Web services based on Apache SOAP.

Web Services for J2EE 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.

Web Services for J2EE 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.

In order to achieve the benefits of using Web Services for J2EE, the Web services that you want to communicate with (provided by other sources), must also be based on the Java language. These other Web services can use other operating systems and languages, but the Web service itself must be based on the Java language.

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's 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.

Using Web Services for J2EE in WebSphere Application Server is based on J2EE 1.3. The same standards are included in J2EE 1.4.

To review the entire Web Services for J2EE specification, see Web services: Resources for learning.

Java API for XML-based remote procedure call (JAX-RPC)

The *Java API for XML-based RPC (JAX-RPC)* specification enables Java language developers to develop SOAP-based interoperable and portable Web services. JAX-RPC provides core APIs for developing and deploying Web services on a Java platform and is a required part of the J2EE 1.4 platform. JAX-RPC Web services can also be developed and deployed on J2EE 1.3 containers.

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 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.

To learn more about JAX-RPC see Web services: Resources for learning.

Artifacts used to develop Web services based on Web Services for J2EE

Development artifacts enable an enterprise bean or a Java bean module to be 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) 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 being 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 W3C Note.
- **webservices.xml**

  The `webservices.xml` file contains the J2EE Web service deployment descriptor 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 JavaBean (EJB), 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.
- **webservicesclient.xml**

  The `webservicesclient.xml` file is the client-side deployment descriptor describing the services being accessed. The `webservicesclient.xml` file is defined in the Web Services for J2EE specification, available through Web services: Resources for learning.
- **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 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.

There are references to the JAX-RPC specification through 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 namespaces 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 tool 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. 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 WSDL. For example, a `java.lang.String` class can be mapped to either an `xsd:string` or `soapenc:string`. The **Java2WSDL** command chooses one of these mappings, but the WSDL file must be edited if a different mapping is desired.
- There are multiple ways to generate WSDL constructs. For example, the part element in the `wsdl:message` can be generated with a `type` or `element` attribute. The **Java2WSDL** command makes an informed choice based on the settings of the -style and -use options.
- 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, write a bean property value as an attribute instead of an element.
- The WSDL file requires editing if header or attachment support is desired.
- The WSDL file requires editing if a multipart WSDL, one using `wsdl:import`, 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 specify 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 namespace `http://webservice.ibm.com`.

  The default mapping between XML namespaces and Java package names can be overridden using the -NStoPkg and -PkgtoNS options of the **WSDL2Java** and **Java2WSDL** commands.
- **Identifier mapping**

  Java identifiers are mapped directly to WSDL file and XML identifiers.

  Java bean property names are mapped to the WSDL file and 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 XML identifiers. See the WSDL2Java command -implClass option for more details.
- **WSDL construction summary**

  The following table summarizes the mapping from a Java construct to the related WSDL and XML construct.

Java construct	WSDL and XML construct
Service Endpoint Interface	`wsdl:portType`
Method	`wsdl:operation`

Parameters	`wsdl:input, wsdl:message, wsdl:part (1)`
Return	`wsdl:output, wsdl:message, wsdl:part (1)`
Throws	`wsdl:fault, wsdl:message, wsdl:part (1)`
Primitive types	`xsd` and `soapenc` simple types
Java beans	`xsd:complexType`
Java bean properties	Nested `xsd:elements` of `xsd:complexType`
Arrays	JAX-RPC defined array `xsd:complexType`
User defined exceptions	`xsd:complexType`

Note: The generated WSDL file is affected by the -style and -use options. A `wsdl:binding` that conforms to the generated `wsdl:portType` is generated. The style and use constructs of the `wsdl:binding` are determined from the -style and -use options. A `wsdl:service` containing a port that references the generated `wsdl:binding` is generated. The names and values of the `wsdl:service` are controlled by the **Java2WSDL** command options.

- **Style** and **use**

  Use the -style and -use options to generate different kinds of WSDL files. The four supported combinations are:
  - -style RPC -use ENCODED
  - -style DOCUMENT -use LITERAL
  - -style RPC -use LITERAL
  - -style DOCUMENT -use LITERAL -wrapped false

  The -use LITERAL option affects the generated WSDL file in the following ways:
  - The `soap:body` elements in the `wsdl:binding` are specified as `use="literal"`.
  - The `soap:fault` elements in the `wsdl:binding` are specified as `use="literal"`.
  - The soap encoded types are not used.
  - The soap encoded array style is not used. The `maxOccurs` attribute is used to indicate arrays.

  The -use ENCODED option affects the generated WSDL file in the following ways:
  - The `soap:body` elements in the `wsdl:binding` are specified as `use="encoded"` and the `encodingStyle` is set.
  - The `soap:fault` elements in the `wsdl:binding` are specified as `use="encoded"` and the `encodingStyle` is set.

  The -style RPC option affects the generated WSDL file in the following ways:
  - The `wsdl:part` elements use the `type` attribute to reference XML types.
  - The `wsdl:binding` is specified as `style="rpc"`.

  The -style DOCUMENT -wrapped false option affects the generated WSDL file in the following ways:
  - The `wsdl:part` elements use the `type` attribute to reference simple types. The `element` attribute is used to reference the root `xsd:elements` for everything that is not a simple type.
  - The `wsdl:binding` is specified as `style="document"`.

  The -style DOCUMENT -wrapped true option generates a WSDL file that conforms to the .NET WSDL file format:
  - A request `xsd:element` is generated for each method in the Service Endpoint Interface as follows:
    - The name of the `xsd:element` is the same as the name of the `wsdl:operation`.
    - The `xsd:element` contains an `xsd:sequence` that contains `xsd:elements` defining each parameter.
    - The request `wsdl:message` references the wrapper `xsd:element` using a single part:
      - The name of the part is `parameters`.
      - The `element` attribute is used to reference the wrapper `xsd:element`.
  - A response `xsd:element` is generated for each method in the Service Endpoint Interface as follows:

- The name of the `xsd:element` is the same as the name of the `wsdl:operation` appended with `Response`
- The `xsd:element` contains an `xsd:sequence` that contains `xsd:elements` defining the return value.
- The request `wsdl:message` references this wrapper `xsd:element` using a single part.
  - The `element` attribute is used to reference the wrapper `xsd:element`.

– The `wsdl:binding` is specified as `style="document"`.

Mapping of standard XML types from Java types

Some Java types map directly to standard XML types. These types do not require additional XML definitions in the `wsdl:types` section.

- **JAX-RPC Java primitive type mapping**

  The following table describes the mapping from the Java primitive and standard types to XML standard types. For more information see the JAX-RPC specification.

Java type	XML type
`int`	`xsd:int`
`long`	`xsd:long`
`short`	`xsd:short`
`float`	`xsd:float`
`double`	`xsd:double`
`boolean`	`xsd:boolean`
`byte`	`xsd:byte`
`byte[]`	`xsd:base64Binary` **Note:** The default mapping for `byte[]` is `xsd:base64Binary`. The data in `byte[]` is passed over the wire as a text string encoded in the `base64` format. An alternative format is `xsd:hexBinary`. To use the `xsd:hexBinary` format: • Edit the WSDL file and change `xsd:base64Binary` to `xsd:hexBinary`.
`java.lang.String`	`xsd:string`
`java.math.BigInteger`	`xsd:integer`
`java.math.BigDecimal`	`xsd:decimal`
`java.util.Calendar`	`xsd:dateTime`
`java.util.Date` **Note:** This mapping is not covered by the JAX-RPC.	`xsd:date`
`java.lang.Boolean`	`xsd:boolean xsi:nillable=true`
`java.lang.Float`	`xsd:float xsi:nillable=true`
`java.lang.Double`	`xsd:double xsi:nillable=true`
`java.lang.Long`	`xsd:long xsi:nillable=true`
`java.lang.Integer`	`xsd:int xsi:nillable=true`
`java.lang.Short`	`xsd:short xsi:nillable=true`
`java.lang.Byte`	`xsd:byte xsi:nillable=true`

Note: The `java.lang` wrapper classes in the last seven lines of the table map to the same XML construct as the corresponding Java primitive type. In addition, the `xsi:nillable` attribute is generated to indicate that such elements can be null.

- **Additional Java class mappings**

In addition to the standard JAX-RPC mapping, the following classes are mapped directly to XML types:

Java type	XML type
`java.util.Map` **Note:** Any classes that implement `java.util.Map` are also mapped to `apache:Map`.	`apache:Map`
`java.util.Collection` **Note:** Each Java array, except `byte[]`, and every class that implements `java.util.Collection` is mapped to a JAX-RPC defined `soapenc:Array` type.	`soapenc:Array`
`org.w3c.dom.Element`	`apache:Element`
`java.util.Vector`	`apache:Vector`
`java.awt.Image` **Note:** Used for attachment support.	`apache:Image`
`javax.mail.internet.MimeMultiPart` **Note:** Used for attachment support.	`apache:Multipart`
`javax.xml.transform.Source` **Note:** Used for attachment support.	`apache:Source`
`javax.activation.DataHandler` **Note:** Used for attachment support.	`apache:DataHandler`

Generation of wsdl:types

Java types that cannot be mapped directly to standard XML types are generated in the `wsdl:types` section.

- **Java arrays**

  Java arrays for the -use ENCODED option, with the exception of `byte[]`, are generated using the following format. See the JAX-RPC specification for more details. Alternative mappings can be found in Table 18.1 of the JAX-RPC specification.

  Java:

  ```
    Item[]
  ```

  Mapped to:

  ```
  <xsd:complexType name="ArrayOfItem">

   <xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
     <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="ns:Item"/>
    </xsd:restriction>
    </xsd:complexContent
  </xsd:complexType>
  ```
- **JAX-RPC value type and bean mapping**

  A Java class that matches the Java value type or bean pattern is mapped to an `xsd:complexType`. In order for a Java class to be mapped to XML, follow these conditions:
  – The class must have a public default constructor.
  – The class must not implement, directly or indirectly, `java.rmi.Remote`.
  – Public, non-static, non-final, non-transient fields are mapped. The class can contain other fields and methods, but they are not mapped.
  – If the class follows the Java bean pattern and has public getter and setter methods, the property is mapped.

Additional mapping rules can be found in the JAX-RPC specification. The following example indicates the mapping for 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>
```

Inheritance and abstract classes

The example contains two optional JAX-RPC features that are supported by WebSphere Application Server:

1. An abstract class is mapped to an `xsd:complexType` with `abstract="true"`.
2. An indexed bean property (see the methods for d in Base) are mapped to a nested element specified with `maxOccurs="unbounded"`. This format is similar to an XML array, but the SOAP message is different. An XML array defines an element for the array and nested elements for each item in the array. An element defined with `maxOccurs` indicates a series of items without the surrounding array wrapper element. Both formats are popular.

- **JAX-RPC enumeration class mapping**

  Section 4.2.4 of the JAX-RPC specification defines the mapping from an XML enumeration to a Java class. Though not specifically required by the JAX-RPC, the **Java2WSDL** command performs the reverse mapping. If you have a class that has the same format as a JAX-RPC enumeration class, it is mapped to an XML enumeration.

- **Holder classes**

  The JAX-RPC specification defines Holder classes in section 4.3.5. A Holder class is used to support `in` and `out` parameter passing. Every Holder class implements the`javax.xml.rpc.holders.Holder` interface. The **Java2WSDL** command maps Holder classes to the same XML type as the held type. In addition, references to Holder classes affect the generation of `wsdl:messages`.
- **Exception classes**

  If a class extends the exception, `java.lang.Exception`, it is mapped to an `xsd:complexType` similar to the Java bean mapping. The getter methods of the exception are mapped as nested `xsd:elements` of the `xsd:complexType`. See section 5.5.5 of the JAX-RPC specification for more details.

  Note: You need to generate implementation specific exception classes by invoking the **WSDL2Java**command on the resulting WSDL file.
- **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.
- **Generation of root elements**

  If the **Java2WSDL** command generates an `xsd:complexType` or `xsd:simpleType` for a parameter reference, a corresponding `xsd:element` is also generated. The `xsd:element` has the same name as the `xsd:complexType/xsd:simpleType` and uses the `type` attribute to reference the `xsd:complexType/xsd:simpleType`. The `wsdl:message` part can use the `element` attribute or the `type` attribute to reference the `xsd:element` or type. This choice is determined by the -style and -use options.

Generation from the interface or implementation class

The class passed to the **Java2WSDL** command represents the interface of the `wsdl:service`. The `wsdl:portType` and `wsdl:message` elements generate from this interface or implementation class.

- **Generation of the wsdl:portType**

  The name of the `wsdl:portType` is the name of the class unless overridden by the -portTypeName option.
- **Generation of wsdl:operation**

  A `wsdl:operation` generates for each public method in the interface that throws the exception, `java.rmi.RemoteException`.
  – The name of the `wsdl:operation` is the name of the method.
  – The `wsdl:operation` has a `parameterOrder` attribute, which defines the order of the parameters in the signature. Specifically, the `parameterOrder` lists the order of the parts of the request or response `wsdl:messages`.
  – The `wsdl:operation` has a nested `wsdl:input` element that references the request `wsdl:message` using the `message` attribute.
  – The `wsdl:operation` has a nested `wsdl:output` element that references the response `wsdl:message` using the `message` attribute.
  – The `wsdl:operation` has a nested `wsdl:fault` element that references the default `wsdl:message` using the `message` attribute.

  See sections 5.5.4 and 5.5.5 of the JAX-RPC specification for more information.
- **Generation of wsdl:message**

  Generating the `wsdl:message` is directly related to the -style and -use options. The following is the default mapping (-style RPC -use ENCODED):
  – A `wsdl:message` is created to represent the request.
    - A `wsdl:part` representing each parameter is added to the `wsdl:message`.
      • The `wsdl:part` has the same name as the parameter.
      • The `wsdl:part` uses the `type` attribute to locate the XML type of the parameter.
  – A `wsdl:message` is created to represent the response:
    - A `wsdl:part` representing the method return is created.
      • The `wsdl:part` has the same name as the method with Return appended.

Note:
The name of the part is not specified by the JAX-RPC and is typically not checked by SOAP engines.

- The `wsdl:part` has the same name as the parameter.
- The `wsdl:part` uses the `type` attribute to locate the XML type of the parameter.
- A `wsdl:part` is created for each parameter that is a Holder.
- The `wsdl:part` has the same name as the parameter.
- A `wsdl:message` is created to represent the fault if the operation has a `wsdl:fault`.
- A `wsdl:part` representing the fault is created.
- The `wsdl:part` has the same name as the exception.
- The `wsdl:part` uses the `type` attribute to locate the complexType representing the exception.

The same mapping is used as described if you use the -style RPC and -use LITERAL options. It is also valid to use the `wsdl:part element attribute` instead of the `type` attribute to reference the XML schema. If you use the -style DOCUMENT -wrapped false and -use LITERAL options, the same mapping is used as described except the `wsdl:part element` attribute is used to reference the XML schema. If the XML schema is a primitive type, like `xsd:string`, the `type` attribute is used to reference the XML type. The -style DOCUMENT, -wrapped true and -use LITERAL options result in completely different mappings for the request and response messages. For example:

– A request `xsd:element` is generated for each method in the Service Endpoint Interface.
  - The name of the `xsd:element` is the same as the name of the `wsdl:operation`.
  - The `xsd:element` contains an `xsd:sequence` that contains `xsd:elements` defining each parameter.
  - The request `wsdl:message` references the wrapper `xsd:element` using a single part.
    - The name of the part is `parameters`.
    - The `element` attribute is used to reference the wrapper `xsd:element`.

– A response `xsd:element` is generated for each method in the Service Endpoint Interface.
  - The name of the`xsd:element` is the same as the name of the `wsdl:operation` appended with Response.
  - The `xsd:element` contains an `xsd:sequence` that contains `xsd:elements` defining the return value.
  - The request `wsdl:message` references this wrapper `xsd:element` using a single part.
    - The element `attribute` is used to reference the wrapper `xsd:element`.

- **Generation of wsdl:binding**

  Generate a `wsdl:binding` with a name defined by the **Java2WSDL -bindingName** command.
  – The `wsdlsoap:binding style` attribute is set to `rpc` if you use the -style RPC option; otherwise it is set to `document`.
  – A `wsdl:operation` generates for each `wsdl:operation` defined in the `wsdl:portType`.
  – Each `wsdl:operation` has corresponding `wsdl:input`, `wsdl:output`and `wsdl:fault` elements.
  – The `wsdl:input`, `wsdl:output` and `wsdl:fault` elements each contain a `wsdlsoap:body` element.
  – The `wsdlsoap:body use` attribute is set to `literal` or `encoded` according to the **-use** argument. Set the `encodingStyle` attribute to `http://schemas.xmlsoap.org/soap/encoding/` when **use** is encoded.
- **Generation of the wsdl:service**

  Generate a `wsdl:service` with a name defined by the **Java2WSDL -serviceElement** command. For example:
  – The `wsdl:service` contains a port with a name defined by the **Java2WSL -servicePortName** command.
  – The port references the generated `wsdl:binding` with the `binding` attribute.
  – The port contains a `wsdlsoap:address` element with a
  – The `location` attribute is set to the value of the **Java2WSDL -location** command.

WSDL-to-Java mapping

The **WSDL2Java** command tool uses the following rules to generate Java classes when developing your Web services client and server. In addition, implementation specific Java classes are generated that assist in the serialization and deserialization, and invocation of the Web service.

General issues

- **Mapping of namespace to package**

  The JAX-RPC does not specify the mapping of XML namespaces to Java package names. The 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 XML namespace to Java package disregards the context-root. If two namespaces are the same up until 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 Java package `com.ibm.websphere`. The default mapping between XML namespaces and Java package names can be overridden using the -NStoPkg and -PkgtoNS options of **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 section 20 of the JAX-RPC specification for the rules to map an XML name to a Java identifier.

  The mapping rules attempt to follow accepted Java coding conventions. Class names always begin with an uppercase letter. Method names begin with a lowercase letter. The **WSDL2Java**command generates metadata in the `_Helper` class so that the values are serialized or deserialized using the XML names specified in the WSDL file.

- **Java construction summary**

WSDL and XML	Java
`xsd:complexType (struct)` **Note:** The `xsd:complexType` can also represent a Java exception if referenced by a `wsdl:message` for a `wsdl:fault` .	`Java Bean Class` **Note:** The classes, `_Helper`, `_Ser`, and `_Deser`, generate for each Java bean class. These implementation classes aid serialization and deserialization.
`nested xsd:element/xsd:attribute`	Java bean property
`xsd:complexType (array)`	Java array
`xsd:simpleType` (enumeration)	JAX-RPC enumeration class
`xsd:complexType (wrapper)`The method parameter signature typically is determined by the `wsdl:message`. However, if the WSDL file is a .NET wrapped style, the method parameter signature is determined by the wrapper `xsd:element`	Service Endpoint Interface method parameter signature **Note:** If a parameter is `out` or `inout`, a `Holder` class generates.
--	--
`wsdl:message`The method parameter signature typically is determined by the `wsdl:message`. However, if the WSDL file is a .NET wrapped style, the method parameter signature is determined by the wrapper `xsd:element`	Service Endpoint Interface method signature **Note:** If a parameter is `out` or `inout`, a `Holder` class generates.
`wsdl:portType`	Service Endpoint Interface
`wsdl:operation`	Service Endpoint Interface method
`wsdl:binding`	Stub **Note:** The Stub and ServiceLocator classes are implementation specific.
`wsdl:service`	Service Interface and ServiceLocator **Note:** The Stub and ServiceLocator classes are implementation specific.
`wsdl:port`	Port accessor method in Service Interface

Mapping standard XML types

- **JAX-RPC simple XML types mapping**

  The following mappings are XML types to Java types. For more information about these mappings, see section 4.2.1 of the JAX-RPC specification.

XML type	Java type
`xsd:string`	`java.lang.String`
`xsd:integer`	`java.math.BigInteger`
`xsd:int` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`int`
`xsd:long` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`long`
`xsd:short` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`short`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:float` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`float`
`xsd:double` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`double`
`xsd:boolean` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`boolean`
`xsd:byte` **Note:** If an element with this type has the `xsi:nillable` attribute set to `true`, it is mapped to the Java wrapper class of the primitive type.	`byte`
`xsd:dateTime`	`java.util.Calendar`
`xsd:date` **Note:** This mapping is not supported by the JAX-RPC.	`java.util.Date`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
--	--
`soapenc:base64`	`byte[]`
`soapenc:base64Binary`	`byte[]`
`soapenc:string`	`java.lang.String`
`soapenc:boolean`	`java.lang.Boolean`
`soapenc:float`	`java.lang.Float`
`soapenc:double`	`java.lang.Double`
`soapenc:decimal`	`java.math.BigDecimal`

soapenc:int	java.lang.Integer
soapenc:integer **Note:** This mapping is not supported by the JAX-RPC.	java.math.BigInteger
soapenc:short	java.lang.Short
soapenc:long **Note:** This mapping is not supported by the JAX-RPC.	java.lang.Long
soapenc:byte	java.lang.Byte

- **JAX-RPC optional simple XML type mapping**

  The **WSDL2Java** command supports the following JAX-RPC optional simple XML types.

XML type	Java type
xsd:qname	javax.xml.namespace.QName

- **JAX-RPC xsd:anyType mapping**

  The **WSDL2Java** command maps an `xsd:anyType` to a `java.lang.Object`. This is an optional feature of the JAX-RPC specification. The `xsd:anyType` can be used to store any XML type other than the XML primitive type. An `xsd:anyType` is always serialized along with an `xsi:type` that specifies the actual type.
- **Additional supported mappings**

  The following mappings are also supported by the **WSDL2Java** command. These mappings are not defined by the JAX-RPC specification.

XML type	Java type
apache:PlainText **Note:** For MIME attachments.	java.lang.String
apache:Map	java.util.Map
apache:Element	org.w3c.dom.Element
apache:Vector	java.util.Vector
apache:Image **Note:** For MIME attachments.	java.awt.Image
apache:Multipart **Note:** For MIME attachments.	javax.mail.internet.MimeMultipart
apache:Source **Note:** For MIME attachments.	javax.xml.transform.Source
apache:octetStream **Note:** For MIME attachments.	javax.activation.DataHandler
apache:DataHandler **Note:** For MIME attachments.	javax.activation.DataHandler

Mapping XML defined in the `wsdl:types` section

The **WSDL2Java** command generates Java types for the XML schema constructs defined in the `wsdl:types` section. The XML schema language is broader than the required or optional subset defined by the JAX-RPC specification. The **WSDL2Java** command supports all required mappings and most optional mappings. In addition, the command supports some XML schema mappings that are outside the JAX-RPC specification. In general, the **WSDL2Java** command ignores constructs that it does not support. For example, the **WSDL2Java** 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 it maps unsupported constructs to `wasws:SOAPElement`.
- **Mapping of xsd:complexType to Java bean**

The most common mapping is from an `xsd:complexType` to a Java bean class.

- ***Standard Java bean mapping***

  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) {...}

    }
  ```

- ***Methods equals() and hashCode()***

  The generated Java bean classes contain an implementation of the equals() method. The generation of this method is outside the JAX-RPC specification. The equals() method returns `true` if equals() is `true` for each contained bean property. The implementation accounts for self-referencing loops. This version of the equals() method is typically more useful than the ″identity″ equals provided by `java.lang.Object`.

  A corresponding hashCode() method is also generated in the Java bean class.

- ***Properties and indexed properties***

  In the standard Java bean mapping example, the nested `xsd:element` for property `a` is mapped to a Java bean property. In addition, the **WSDL2Java** command maps a nested `xsd:element` with `maxOccurs` > 1 to a Java bean indexed property.

- ***Attributes***

  The **WSDL2Java** command also supports the `xsd:attribute` element, as shown in the following example.

  Attribute `a` is mapped as a Java bean property, which is exactly the same mapping as a nested `xsd:element`. Implementation specific metadata is generated in the `Sample2_Helper` class to ensure that property `a` is serialized and deserialized as an attribute, and not as a nested element. For example:

  XML:

  ```
  <xsd:complexType name="Sample2">
    <xsd:sequence>
     <xsd:attribute name="a" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  ```

  Java:

```
public class Sample2 {
     // ..
     public Sample2() {}

     // Bean Property a
     public String getA()             {...}
     public void   setA(String value) {...}
  }
```

– ***Qualified versus unqualified names***

The **WSDL2Java** command supports the `elementForm` and `attributeForm` schema attributes.

This support is not specified in the JAX-RPC specification. These attributes are used to indicate whether an element or attribute is serialized and deserialized with a qualified or unqualified name. The default setting for `elementForm` is `qualified` and the default setting for `attributeForm` is `unqualified`. These settings do not affect the Java bean class that is generated, but the information is captured in the `_Helper` class metadata.

– ***Extension and the abstract attribute***

The **WSDL2Java** command supports extension of an `xsd:complexType` through the `xsd:extension` element. This support is required by the JAX-RPC specification.

The **WSDL2Java** command supports the `abstract` attribute. This feature is listed as optional by the JAX-RPC specification.

The following example shows the accepted use of the extension and abstract constructs. WebSphere Application Server uses the extension and abstract constructs to support polymorphism.

```
XML:

<xsd:complexType name="Base" abstract="true">
 <xsd:sequence>
  <xsd:element name="a" type="xsd:int"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
 <xsd:complexContent>
  <xsd:extension base="ns:Base">
   <xsd:sequence>
    <xsd:element name="b" type="xsd:int"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

Java:
   public abstract class Base {
      // ...
      public Base() {}

      public int getA() {...}
      public void setA(int a) {...}
   }

   public class Derived extends Base {
      // ...
      public Derived() {}

      public int getB() {...}
      public void setB(int b) {...}
   }
```

– ***Support for `xsd:any`***

The **WSDL2Java** command supports `xsd:anyelement`, which is different than `xsd:anyType`. This feature is not defined within the JAX-RPC specification and is subject to change.

If an `<xsd:any/>` element is defined within `xsd:sequence` or `xsd:all group`, SOAP values that do match one of the `xsd:elements` are stored in the Java bean. Values can be accessed from the Java bean using the get_any() and set_any() methods.

- **Mapping of xsd:element**

  An `xsd:element` is a construct that has a name or `name` attribute, and a type defined by a `complexType` or `primitive type`. There are two different kinds of `xsd:elements`:
  - **Root:** Defined directly underneath the schema elements and referenced by other constructs.
  - **Nested:** Nested underneath group elements and are not referenced by other constructs.

  Root elements are referenced by the WSDL file constructs, especially if the WSDL file is used to describe a literal service. Typically, root elements and types have the same names, which is allowed in the schema language. Under most circumstances the **WSDL2Java** command can produce Java artifacts without name collisions.
  - ***Four ways to reference a type***

    There are four ways that a nested or root `xsd:element` can reference a type:
    - **Use the `type` attribute:**

      This is the most common way to reference a type, for example:

      ```
      <xsd:element name="one" type="ns:myType"/>
      ```

      The **WSDL2Java** command recognizes the `type` attribute as a reference to a `complexType` or `simpleType`named, `myType`. The **WSDL2Java** command generates a Java type based on the characteristics of `myType`. Support for the `type` attribute is required by the JAX-RPC specification.
    - **Use the `ref` attribute:** For example:

      ```
      <xsd:element ref="ns:myElement"/>
      ```

      The **WSDL2Java** command recognizes the `ref` attribute as a reference to another root element named `myElement`. The name of the element is obtained from the referenced element, such as `myElement`. The type of the element is the type of the referenced element. The **WSDL2Java** command generates a Java type based on the characteristics of the referenced type. The `ref` attribute is an optional feature of the JAX-RPC specification.
    - **Use no attribute:**

      For example:

      ```
      <xsd:element name="three"/>
      ```

      When you do not use an attribute, the **WSDL2Java** command recognizes a reference to the `xsd:anyType` as defined by the XML schema specification. The `xsd:anyType` is an optional type of the JAX-RPC specification.
    - **Use an anonymous type:**

      For example:

      ```
      <xsd:element name="four">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="foo" type="xsd:string"/>
         </xsd:sequence>
        </xsd:complexType>
       </ xsd:element>
      ```

      When you use an anonymous type, the **WSDL2Java** command recognizes a reference to the type defined within the element.

      Note: The `complexType` does not have a name.

The **WSDL2Java** command generates a Java type based on the characteristics of this type. Since the anonymous type does not have a name, the **WSDL2Java** command uses the name of the `container` element, which can result in collisions with defined types and other anonymous types. The **WSDL2Java** command automatically detects and renames classes to avoid collisions. Support for anonymous types is not defined by the JAX-RPC specification, however using anonymous types is common.

Note: An `xsd:attribute` is like an `xsd:element`; it contains a name and refers to a type. An `xsd:attribute` can refer to its type with the `type`attribute or using an anonymous type.

– ***Element specific attributes***

Some attributes can be applied to `xsd:elements` and not to XML types.

The `maxOccurs` attribute indicates the maximum number of occurrences of the element in the SOAP message. The default value is 1. If the value is greater than 1, or unbounded, the **WSDL2Java** command maps the construct to a Java array or bean indexed property. Metadata is also generated to properly serialize and deserialize a series of elements versus a normal XML array. The `maxOccurs` attribute is an optional feature of the JAX-RPC specification.

The `minOccurs` attribute indicates the minimum number of occurrences of the element in the SOAP message. The default value is 1. The `xsi:nillable` attribute indicates whether the element can have a `nil` value. The `minOccurs` and `xsi:nillable` settings affect how a `null` value is serialized in a SOAP message. If `minOccurs=`0, the `null` value is not serialized. If `xsi:nillable=`
`true`, the value is serialized with the `xsi:nil=true` attribute.

- **Mapping of xsd:complexType to Java array**

The **WSDL2Java** command maps the following three kinds of XML formats to Java arrays:

XML:

```
<xsd:element name="array1" type="soapenc:Array"/>
```

Java:

```
Object[] array1;
```

XML:

```
<xsd:complexType name="arrayOfInt">
 <xsd:complexContent>
  <xsd:restriction base:"soapenc:Array">
   <xsd:attribute ref:"soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
  </xsd:restriction>
  </xsd:complexContext>
 </xsd:complexType>

 <xsd:element name="array2" type="ns:arrayOfInt"/>
```

Java:

```
  int[] array2;
```

XML:

```
<xsd:complexType name="arrayOfInt">
 <xsd:complexContent>
  <xsd:restriction base:"soapenc:Array">
   <xsd:sequence>
    <xsd:element name="item" type="xsd:int" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>

 <xsd:element name="array3" type="ns:arrayOfInt"/>
```

Java:

```
int[] array3;
```

- **Mapping of xsd:simpleType enumeration**

  The **WSDL2Java** command maps the following XML enumeration to a JAX-RPC specified enumeration class. See section 4.2.4 of the JAX-RPC specification for more details.

  ```
  <xsd:simpleType name="EyeColorType">

   <xsd:restriction base="xsd:string">
    <xsd:enumeration value="brown"/>
    <xsd:enumeration value="green"/>
    <xsd:enumeration value="blue"/>
   </xsd:restriction>
  </xsd:simpleType>
  ```
- **Mapping of xsd:complexType to exception class**

  If a `complexType` is referenced in a `wsdl:message` for a `wsdl:fault`, the `complexType` is mapped to a class that extends the exception, `java.lang.Exception`. This mapping is similar to the mapping of a `complexType` to a Java bean class, except a full constructor is generated, and only getter methods are generated. See section 4.3.6 of the JAX-RPC specification for more details.
- **Other mappings**

  The **WSDL2Java**command supports the mapping of `xsd:simpleType` and `xsd:complexTypes` that extend `xsd:simpleTypes`. These constructs are mapped to Java bean classes. The simple value is mapped to a Java bean property named, `value`. This is an optional feature of the JAX-RPC specification.

Mapping of `wsdl:portType`

The `wsdl:portType` construct is mapped to the Service Endpoint Interface. The name of the `wsdl:portType` is mapped to the name of the class of the Service Endpoint Interface.

Mapping of `wsdl:operation`

A `wsdl:operation` 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 .NET document and literal wrapped format, the method parameters are mapped from the wrapper `xsd:element`. The .NET document 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` constructs.
  - The WSDL file must have `use="literal"` in its `wsdl:binding` constructs.
  - 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:

  XML:

  ```
  <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 there are multiple `wsdl:parts` 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.

  The **WSDL2Java** command supports overloaded methods, but confirm that the part names of the overloaded methods are unique. For example:

  XML:

```
<wsdl:message name="request">
 <part name="param1" type="xsd:string"/>
 <part name="param2" type="xsd:int"/>
</wsdl:message name="response"/>

<wsdl:message name="response"/>
...
<wsdl:operation name="myMethod" parameterOrder="param1, param2">
 <input name="input" message="request"/>
 <output name="output" message="response"/>
</wsdl:operation>


Java:

void myMethod(String param1, int param2) ...
```

Mapping of `wsdl:binding`

The **WSDL2Java** command uses the `wsdl:binding` information to generate an implementation specific client side stub. WebSphere Application Server uses the `wsdl:binding` information on the server side to properly deserialize the request, invoke the Web service, and serialize the response. The information in the `wsdl:binding` should not affect the generation of the Service Endpoint Interface, but it can when the document and literal wrapped format is used or when there are MIME attachments.

- **MIME attachments**

  For a WSDL 1.1 compliant WSDL file, a part of an operation message, which is defined in the binding to be 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 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`

  There are a number of problems with MIME attachments as they are defined in WSDL 1.1, including:
  – The semantics of the `mime:multipartRelated` clause are not fully defined
  – The semantics do not allow for arrays of MIME attachments

Because of these problems, several types are not specified by the JAX-RPC for MIME attachments. These types are defined in the supported mappings previously discussed.

- **Headers**

  A `wsdl:binding` can also define SOAP headers, for example:

  XML:

  ```
  <wsdl:message name="request">
    <part name="param1" type="xsd:string"/>
   </wsdl:message/>

   <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>
     <soap:header message="request" part="param1" use="literal"/>
    </input>
    ...
   </wsdl:operation>
  ```

  Java:

  ```
  void myMethod(String param1) ...
  ```

  This is an example of an explicit header or a header with a value determined from a `method` parameter. Instead of appearing in the `soap:body` SOAP message, the value of `param1` now appears in the `soap:header` SOAP message. The **WSDL2Java** command supports explicit headers and does not support implicit headers. Implicit headers have a value not determined by a parameter. For example, you could replace the `soap:header` clause in the example with:

  ```
  <soap:header message="someOtherMsgNotAppearingInthePortType" part="someOtherPart"
  use="literal"/>
  ```

  Note: The **WSDL2Java** command supports explicit headers, but it is not considered good programming practice to use them. Headers are typically used for middleware logic, not business logic. Explicit headers place parameters used in business logic into the header.

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`. 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 is sent over the wire. The **WSDL2Java** command and the WebSphere Application Server run time use the information in the WSDL file to confirm that the SOAP message is properly serialized and deserialized.

Document versus RPC, literal versus encoded

If a `wsdl:binding` indicates a message is sent using an RPC format, the SOAP message contains an element defining the operation. If a `wsdl:binding` indicates the message is sent using a document format, the SOAP message does not contain the `operation` element.

If the `wsdl:part` is defined using the `type` attribute, the name and type of the part are used in the message. If the `wsdl:part` is defined using the `element` attribute, the name and type of the element are used in the message. The `element` attribute is not allowed by the JAX-RPC specification when `use="encoded"`.

If a `wsdl:binding` indicates a message is encoded, the values in the message are sent with `xsi:type` information. If a `wsdl:binding` indicates that a message is literal, the values in the message are typically not sent with `xsi:type` information. For example:

```
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"/>
 ...
RPC/ENCODED:
<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:
<soap:body>
 <a>ABC</a>
 <c>123</a>
</soap:body>
DOCUMENT/LITERAL  wrapped:
<soap:body>
 <ns:method_wrapper>
  <a>ABC</a>
  <c>123</a>
 <ns:method_wrapper>
</soap:body>
```

The document and literal wrapped mode is the same as the document and literal mode. However, in the document and literal wrapped mode, there is only a single element within the body, and the element has the same name as the operation.

Multi-ref processing

If `use=encoded`, XML types that are not simpleTypes are passed in the SOAP message using the `multi-ref` attributes, `href` and `id`. The following example assumes that parameters one and two reference the same Java bean named, `info` containing fields `a` and `b`:

Note:

Deserialization produces a single instance of the `info` class for the encoded case and two instances are created for the literal case.

```
RPC/ENCODED:
<soap:body>
 <ns:method>
  <param1 href="#id1"/>
  <param2 href="#id2"/>
 <ns:method>
 <multiref id="id1" xsi:type="ns:info">
  <a xsi:type="xsi:string">hello<a>
  <b xsi:type="xsi:string">world</b>
 </multiref>
</soap:body>
```

```
RPC/LITERAL:
<soap:body>
 <ns:method>
  <param1>
   <a>hello</a>
   <b>world</b>
  </param1>
  <param2>
   <a>hello</a>
   <b>world</b>
  </param2>
 <ns:method>
</soap:body>
```

XML arrays and the `maxOccurs` attribute

A SOAP message is affected by whether the element is defined by an XML array or using the `maxOccurs` attribute.

```
WSDL:
<element name="foo" type="ns:ArrayOfString"/>

Literal Instance:

 <foo>
  <item>A</item>
    <item>B</item>
  <item>C</item>
 </foo>
```

```
WSDL:
<element name="foo" maxOccurs="unbounded" type="xsd:string"/>

Literal Instance:

 <foo>A</foo>
  <foo>B</foo>
 <foo>C</foo>
```

`minOccurs` and `nillable` attributes

An element specified with `minOccurs=0` that has a `null` value is not serialized in the SOAP message. An element specifying `nillable="true"` has a null value and is serialized into a SOAP message with the `xsi:nil=true` attribute. For example:

```
<a xsi:nil="true"/>
```

Qualified versus unqualified

The XML schema `attributeForm` and `elementForm` attributes indicate whether the attributes and nested elements are serialized with qualified or unqualified names. If a part name is serialized, it is always serialized as an unqualified name.

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) specification. The **Java2WSDL** command accepts a Java class as input and produces a WSDL file representing the input class. If there is an existing file at the output location, it is overwritten. The WSDL file 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.

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 EJB remote interface that extends the javax.ejb.EJBObject class
  - Service Endpoint Interface that extends the java.rmi.Remote class
  - Java bean

  The **Java2WSDL** command locates the class in CLASSPATH.

Important arguments

- **-bindingName** ***name***

  Specifies the name to use for the binding element. If not specified, the binding name is the portTypeName.
- **-help**

  Displays the help message.
- **-helpX**

  Displays the help message for extended options.
- **HelpXoptions**
  - **-debug**

    Displays debug messages.
  - **-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 specified.
  - **-MIMEStyle**

    Specifies a style representing Multipurpose Internet Mail Extensions (MIME) information. Valid arguments are:
    - **Axis**
    - **WSDL11** (default)
  - **-soapAction**

    Valid arguments are:
    - **DEFAULT**

      Sets the `soapAction` field according to deployment information.
    - **NONE**

      Sets the `soapAction` field to `""`.
    - **OPERATION**

      Sets the `soapAction` field to the operation name.

- **-stopClasses** ***parent*** **[,** ***parent*****]**

  If the -all argument is specified, the **Java2WSDL** command searches inherited classes and interfaces to construct the list of methods for WSDL file operations. The **Java2WSDL** command searches inherited classes and interfaces when generating extended complexTypes. The search stops when a class or interface is found within a package that begins with `java` or `javax`. The -stopClasses argument can be used to define additional classes that cause the search to stop.
- **-namespaceImpl** ***namespace***

  Specifies the target namespace for the implementation WSDL if -outputImpl specified.
- **-voidReturn**

  Valid arguments are:
  - **ONEWAY**

    Methods with void returns are one-way. This is the default for JMS transport.
  - **TWOWAY**

    Methods with void returns are two-way. This the default for HTTP transport.
- **-wrapped** ***boolean***

  Specifies if the WSDL file should be generated according to wrapped rules. This is only valid if `use` is literal. The option defaults to `true`.

• **-extraClasses** ***classes***

Specifies other classes that should be represented in the WSDL file.

• **-input** ***wsdl-uri***

Specifies the input WSDL file used to build an output WSDL file. Information from an existing WSDL file, whose name is specified in this option, is used with the input Java class to generate the desired output.

• **-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 class.

• **-location** ***location***

Provides the location or Uniform Resource Locator (URL) of the service. Typically, this value fills automatically when the Web service deploys. Use this argument to specify the location if you want to generate a WSDL file containing a location URL without deploying. A warning displays to remind you that the generated WSDL file should not be published if the final location is not yet been determined. The name after the last slash or backslash is the name of the service port, unless the name is overridden by the -servicePortName argument. The service port address location attribute is assigned the specified value.

• **-namespace** ***targetNamespace***

Indicates the target namespace for the WSDL file being generated. See Mapping between Java, WSDL and XML for the algorithm used to obtain the default namespace.

• **-output** ***wsdl-uri***

Indicates the path and file name of the output WSDL file. If not specified, the default file, *`class`*`.wsdl`, is written into the current directory.

• **-PkgtoNS** ***package namespace***

Specifies the mapping of a Java package to a namespace. If there is a package without a namespace, the **Java2WSDL** command generates a namespace name. This argument can be repeated to specify mappings for multiple packages.

• **-portTypeName** ***name***

Specifies the name to use for the portType element. If not specified, the class name is used.

• **-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.
- **-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, or omitting use, a `style=rpc/use=encoded` WSDL file is generated. If RPC is specified with -use LITERAL, a `style=rpc/use=literal` WSDL file is generated. If DOCUMENT is specified with -use LITERAL or omitting use, a `style=document/use=literal` WSDL file is generated.
- **-transport http | jms**

  Generates SOAP bindings for either Hyper Text Transport Protocol (HTTP) (default) or Java Messaging Service (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 only be specified once.
- **-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. For more information, see the Mapping between Java language, WSDL and XML.
- **-verbose**

  Displays verbose messages.

WSDL2Java command

The **WSDL2Java** command tool creates Java classes and deployment descriptor templates from a Web Services Description Language (WSDL) file using the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification. See Mapping between Java language, WSDL and XML for more information.

Classes and files generated

The following kinds of classes and files are generated:
- **For each portType in the WSDL document (`<wsdl:portType>` element tag):**
  - Service Endpoint Interface
- **For each service in the WSDL document (`<wsdl:service>` element tag):**
  - Service Interface when the -role develop-client argument is specified.
  - ServiceLocator when the -role deploy-client argument is specified.

    This class is a WebSphere product-specific implementation of the service interface, and is not used directly.
  - `webservices.xml` deployment descriptor template when the -role develop-server argument is specified
  - `ibm-webservices-bnd.xmi` deployment descriptor template when the -role develop-server argument is specified.
  - `ibm-webservices-ext.xmi` deployment descriptor template when the -role develop-server argument is specified.
  - *wsdlfile*`_mapping.xml` JAX-RPC mapping file when the -role develop-client or -role develop-server is specified.
  - `webservicesclient.xml` deployment descriptor template when the -role develop-client argument is specified.
  - `ibm-webservicesclient-bnd.xmi` deployment descriptor template when the -role develop-client argument is specified.
  - `ibm-webservicesclient-ext.xmi` deployment descriptor template when the -role develop-client argument is specified.

When the role is a server role, the container argument specifies which J2EE container the implementation uses. When the -role develop-server -container ejb arguments are specified, the `webservices.xml`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-ext.xmi` and the mapping file are generated into the `META-INF` subdirectory. When the -role develop-server -container web arguments are specified, the files are generated into the `WEB-INF` directory.

- **For each binding in the WSDL file (<`wsdl:binding`> element tag):**
  - A stub that implements the Service Endpoint Interface (deploy-client role)
  - An implementation template for an enterprise bean and templates for the EJB remote interface and home interface generate when the -role develop-server and -container-ejb arguments are specified.
  - An implementation template for the Java bean when the -role develop-server and -container-web arguments are specified.
- **Other classes and files:**
  - A Java bean representing the structure of the type when the -role develop-server or -role develop-client arguments are specified for each complexType or simpleType.
  - Three classes, `*_Ser.java`, `*_Deser.java`, and `*_Helper.java`, generate for each complexType to assist in converting the bean to SOAP and back when the -role deploy-server or -role deploy-client argument is specified.
  - A `*Holder.java` class generates when the -role develop-server or -role develop-client arguments are specified for each `out` and `inout` parameter.

Command line syntax

The command line syntax is:

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

Required arguments

- ***WSDL-URI***

  Specifies the location of the input WSDL document 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

- **-container** *j2ee-container*

  Indicates the J2EE container to be used. Valid arguments are:
  - **client**

    Indicates client container.
  - **ejb**

    Indicates enterprise JavaBean (EJB) container.
  - **none**

    Indicates no container.
  - **web**

    Indicates Web container.

  If client is role, the default argument is **none**. If server is role, the container must be **ejb** or **web**. The same container option must be used for both development and deployment.
- **-deployScope** *argument*

  Indicates how to deploy the server implementation. Valid arguments are:
  - **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.
- **-genResolver**

  Generates an `absolute-import resolver` class. The purpose of this class is to record the contents of the imported WSDL files used by the WSDL URI. This class is used by the runtime. It can also be used for future **WSDL2Java** command runs. This is desirable when the imported WSDL files are remote and can be inaccessible or slow to access. It also eliminates the possibility that a remote WSDL file might have different contents at run time than it did at development time. The generated class is named `_AbsoluteImportResolver.java`. You should compile and package this class with the other Java classes generated by the **WSDL2Java** command.

- **-help**

  Displays a help message and exits.
- **-helpX**

  Displays a help message for extended options and exits. The options are:
  - **-all**

    Generates Java files for all types, even those that are not referenced.
  - **-debug**

    Prints debugging information.
  - **-fileNStoPkg** *filename*

    Specifies the file of namespace to package mappings. The default is `NStoPKG.properties`.
  - **-genJava** *argument*

    Generates Java files. Valid arguments are:
    - **IfNotExists**, default
    - **Overwrite**
    - **No**
  - **-genXML** *argument*

    Generates the `.xml` and `.xmi` files. Valid arguments are:
    - **IfNotExists**, default
    - **Overwrite**
    - **No**
  - **-password** *password*

    Specifies the login user password to access the WSDL URI.
  - **-testCase**

    Generates the template for a JUnit test case for testing a Web service.
  - **-user** *id*

    Specifies the login user name to access the WSDL URI.
- **-inputMappingFile** mapping file

  Specifies the file name of the Java to WSDL mapping file.
- **-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 there is a namespace in the WSDL file called `urn:AddressFetcher2`, and you want files generated from the objects in this namespace to reside in the package `samples.addr`, provide the -NStoPkg urn:AddressFetcher2=samples.addr argument to the **WSDL2Java** command.
- **-output** directory

  Sets the root directory for emitted files.
- **-role** *j2ee role*

  Specifies the J2EE development role that identifies which files to generate. Valid arguments are:
  - **client**

    Combination of develop-client and deploy-client.
  - **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**

    Combination of develop-server and deploy-server.
- **-timeout** *seconds*

  Specifies how long the **WSDL2Java**command should wait, in seconds, for the WSDL-URI to respond before giving up. The default is 45 seconds, -1 disables the timeout.
- **-useResolver** *resolver-class*

  Specifies an `absolute-import resolver` class to use during parsing. This class must have been created during a previous execution of the **WSDL2Java** command using the -genResolver option. The class must be available in CLASSPATH.
- **-verbose**

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

Setting up a development and unmanaged client execution environment for Web services based on Web Services for J2EE

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. WebSphere Application Server also includes the Assembly Toolkit that can be downloaded from the Web site

```
http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid=
swg24005125&loc=en_US&cs=utf-8&lang=en+en
```

. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

Websphere Studio Application Developer Version 5.1 has GUI-based development tools to develop Web services that integrate with Websphere Application Server 5.0.2.

Before you can set up a Web services development and unmanaged client execution environment within WebSphere Application Server, you must Install WebSphere Application Server.

To set up a Web services development and unmanaged client execution environment:
1. Develop thin application client code on a server machine and run the setupCmdLine script.
2. 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
   ```

Develop Web services based on Web Services for J2EE.

Developing a Web service from a Java bean

Set up a Web services development and unmanaged client execution environment.

To develop a Web service from a Java bean:
1. Access an existing Java bean Web archive (WAR) file.
2. Develop a Java bean Service Endpoint Interface.

3. Develop a Web Services Description Language (WSDL) file.
4. Develop Web services deployment descriptor templates for a Java bean implementation.
5. Configure the `webservices.xml` deployment descriptor.
6. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
7. Assemble a Web services-enabled WAR file when starting from Java.
8. Assemble a Web services-enabled WAR into an EAR file.
9. Deploy the EAR file into WebSphere Application Server.

Test the Web service.

Developing a WSDL file

Develop a Service Endpoint Interface.

You need a Web Services Description Language (WSDL) file to use Web services. You can develop your own WSDL file or get one from a Web service provider through E-mail, downloading or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own.

To develop a WSDL file:

1. Configure the Service Endpoint Interface class and referenced classes into your CLASSPATH.
   - On Windows, set `CLASSPATH="%CLASSPATH%;`*<list your application JAR files and classes>*`"`.
   - On UNIX, 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 an enterprise JavaBean (EJB).
   - Move the WSDL file to the `WEB-INF/wsdl` subdirectory if you are using a Java bean.
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 on (**javac -g**). Parameter names are stored in the `.class` file with the debug information. If your implementation class was compiled with debug on, you can use the **Java2WSDL -implClass *seiImpl seiInterface*** command to generate a WSDL file having the proper part names.

A WSDL file that defines the Web service described by the Service Endpoint Interface.

This example uses a JAR file named `AddressBook.jar` containing a class named `AddressBook.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 was compiled with debugging information on. Run the **Java2WSDL -implClass addr.AddressBookBean addr.AddressBook** command to create a WSDL file named `AddressBook.wsdl`.

Develop Web services deployment descriptor templates from a WSDL file.

WSDL:

Web Services Description Language (WSDL) is an eXtensible Markup Language (XML)-based description language that has been submitted to the World Wide Web Consortium (W3C) as the industry standard for describing Web services. The power of WSDL is derived from two main architectural principles: the ability to describe a set of business operations and the ability to separate the description into two basic units, a description of the operations and the details of how the operation and the information associated with it are packaged.

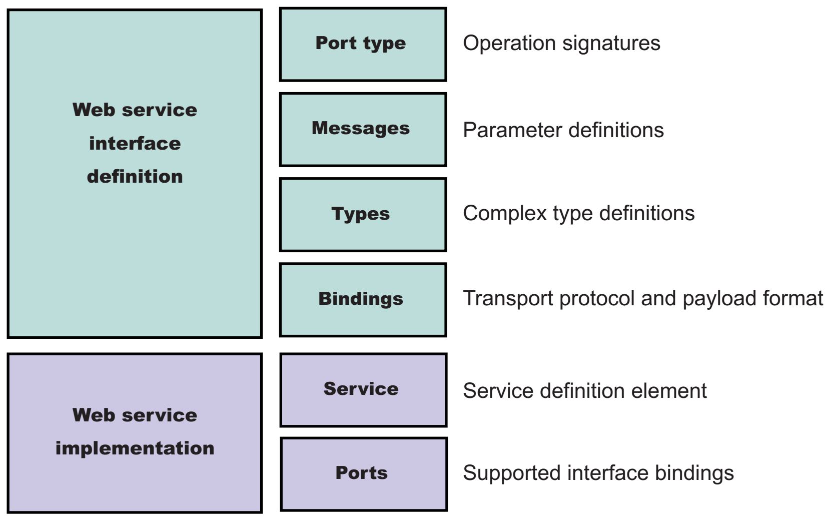
A WSDL document allows a service provider to specify the name and address of the Web service; protocol and encoding style used when accessing the public operations of the Web service; and the type information, including name, operations, parameters and data comprising the interface of the Web service.

The WSDL document is the engine of a Java 2 platform, Enterprise Edition (J2EE) Web service; without it there is no service. The information within a WSDL file maps to the Java application to create a Web service.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Therefore, a WSDL document is composed of several elements. See WSDL anatomy for more information and examples of the WSDL elements.

When creating a Web service for WebSphere Application Server, you must first have an implementation bean that includes a Service Endpoint Interface. Then, you use the **Java2WSDL** command-line tool to create a WSDL that defines the Web service. To learn more about how the WSDL file is used in the development process, see Developing Web services based on Web Services for J2EE.

WSDL anatomy: Web Services Description Language (WSDL) files are written in eXtensible Markup Language (XML). To learn more about XML, see Web services: Resources for learning.


Web service interface definition
Port type
Operation signatures
Messages
Parameter definitions
Types
Complex type definitions
Bindings
Transport protocol and payload format
Web service implementation
Service
Service definition element
Ports
Supported interface bindings

A WSDL contains the following parts:

- **Web service interface definition**

  This is where the elements are contained, as well as the namespaces.
- **Web service implementation**

  This is where you find the definition of the service and ports.

A WSDL file describes a Web service with the following elements:

portType

The description of the operations and their associated messages. PortTypes define abstract operations.

```
<portType name="EightBall">
 <operation name="getAnswer">
  <input message="ebs:IngetAnswerRequest"/>
  <output message="ebs:OutgetAnswerResponse"/>
 </operation>
</portType>
```

message

The description of parameters (input and output) and return values.

```
<message name="IngetAnswerRequest">
 <part name="meth1_inType" type="ebs:questionType"/>
</message>
<message name="OutgetAnswerResponse">
 <part name="meth1_outType" type="ebs:answerType"/>
</message>
```

types

The schema for describing XML complex types used in the messages.

```
<types>
 <xsd:schema targetNamespace="...">
  <xsd:complexType name="questionType">
   <xsd:element name="question" type="string"/>
  </xsd:complexType>
  <xsd:complexType name="answerType">
  ...
</types>
```

binding

Bindings describe the protocol used to access a service, as well as the data formats for the messages defined by a particular portType.

```
<binding name="EightBallBinding" type="ebs:EightBall">
 <soap:binding style="rpc" transport="schemas.xmlsoap.org/soap/http">
 <operation name="ebs:getAnswer">
 <soap:operation soapAction="urn:EightBall"/>
  <input>
   <soap:body namespace="urn:EightBall" ... />
  ...
```

The remaining parts, services and ports, indicate where you can find the WSDL.

Service

Contains the Web service name and a list of the ports.

Ports

Contains the location of the Web service and the binding to used to access the service.

```
<service name="EightBall">
 <port binding="ebs:EightBallBinding" name="EightBallPort">
  <soap:address location="localhost:8080/axis/EightBall"/>
 </port>
</service>
```

Publishing WSDL files:

To publish a Web Services Description Language (WSDL) file you need an enterprise application, also known as an enterprise archive (EAR) file, that contains a Web services-enabled module and has been deployed into WebSphere Application Server. See Deploying Web services based on Web Services for Java 2 platform, Enterprise Edition (J2EE).

The WSDL files for each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files to clients that want to invoke your Web services.

You can publish WSDL files for the deployed EAR file in one of three ways:
1. Publish a WSDL file with the administrative console.
2. Publish a WSDL file with the **wsadmin** command tool.
3. Publish a WSDL file through a URL.

Publishing WSDL files with the administrative console:

When publishing Web Services Description Language (WSDL) files with the administrative console, you can specify default or custom HTTP URL prefixes. You can also specify a Java Message Service (JMS) URL prefix.

To publish a WSDL file with the administrative console:
1. Open the administrative console.
2. Click **Applications**> **Enterprise Applications** > *application*. Under Additional Properties, click **Publish WSDL** which brings you to the **Publish WSDL files for Web Services** panel.
3. Specify the default URL prefixes for the Web service.
   a. Select **HTTP URL prefix**.
   b. Select an entry from the drop down list. If you have multiple application modules, select the application module's checkbox on the module table.
   c. Click **Apply**. The URL prefix is copied to the selected module HTTP URL prefix field.
   d. Click **OK**.
   e. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** panel.
   f. Download the zip file. Follow your browser's instructions to download the zip file.
4. Specify custom URL prefixes for the Web service.
   a. Select **Custom HTTP URL prefix**.
   b. Type the name of the URL prefix in the **Custom HTTP URL prefix** field. The entry must be of the form `http|https://<host_name>:<port_number>`. For example:
      ```
      http://myHost:999
      ```
      If you have multiple application modules, select the application module's checkbox on the module table.
   c. Click **Apply**. The URL prefix is copied to the selected module HTTP URL prefix field.
   d. Click **OK**.
   e. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** panel.
   f. Download the zip file. Follow your browser's instructions to download the zip file.
5. Specify a JMS URL prefix.
   a. Select the application module.
   b. Type the JMS URL prefix into the **JMS URL prefix** field. The entry must be of the form: `jms:/[queue|topic]?destination=<queue or topic_jndi_name>&connectionFactory=<connection_factory_jndi_name>`. For example:
      ```
      jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF1
      ```
   c. Click **OK**.

d. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** panel.
e. Download the zip file. Follow your browser's instructions to download the zip file.

Publishing WSDL files using the wsadmin command:

The Web Services Description Language (WSDL) files in each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files to the clients that want to invoke your Web services.

The scripting client, **wsadmin**, can publish the WSDL files in either local, for example, -conntype NONE, or remote mode. However, in local mode, the target application should be located at the same node where the **wsadmin** command is invoked.

The steps below assume that the application has been deployed and that the application server is running.

To publish a WSDL file with the **wsadmin** command:
1. From a command prompt, start `install_root\bin\wsadmin` if you are using Windows or `install_root/bin/wsadmin` if you are using UNIX.
2. At the **wsadmin** command prompt, enter one of the two commands:
   - **$AdminApp publishWSDL *app_Name path_Name***
   - **$AdminApp publishWSDL *app_Name path_Name* {{module {{*binding url-prefix*}}}}**

   Where
   - *app_Name* is the application name
   - *path_Name* is the absolute path to the zip file that will contain the published WSDL files. The zip file is saved on the machine running WebSphere Application Server, therefore, if the server is running on a different machine, you need to obtain the zip file from that machine. The directory structure of the resulting zip file is based on the following information:
     ```
     Application file name
             module file name
                 META-INF/ or WEB-INF/
                     wsdl/
                           WSDL file name
     ```
     See the usage scenario for an example of this directory structure.
   - *binding* is either http or jms (both are in lower case)
   - *url-prefix* is the partial SOAP address for the associated SOAP binding. For an HTTP binding the form is `http://host:port/` or `https://host:port`.

     For Java Message Service (JMS) bindings, the form is `jms:/queue?destination=dest&connectionFactory=cf` or `jms:/topic?destination=dest&connectionFactory=cf`

   The **$AdminApp publishWSDL *app_Name path_Name*** command updates the WSDL SOAP address prefixes using the default values. If you do not want to update the WSDL SOAP address prefixes, use the other command, instead of the default values.

   The **$AdminApp publishWSDL *app_Name path_Name* {{module {{*binding url-prefix*}}}}** command allows you to customize the WSDL SOAP address for each module. You can specify a different address prefix for each SOAP binding.

The WSDL files from Web services are published to a specified zip file. You can hand the zip file to the client and the client can use the published WSDL files to create a Web services client that accesses the deployed service.

The command to publish WSDL files for a Web service named WebServicesSamples could be
$AdminApp publishWSDL WebServicesSamples c:/temp/samplesWsdl.zip

or

$AdminApp publishWSDL WebServicesSamples c:/temp/sampleswsdl.zip { {AddressBookJ2WB.war {{http http://localhost:9080}}} {StockQuote.jar {{http https://localhost:9443}}} }

The directory structure for this created zip files is

```
WebServicesSamples.ear/StockQuote.jar/META-INF/wsdl/StockQuoteFetcher.wsdl
WebServicesSamples.ear/AddressBookW2JE.jar/META-INF/wsdl/AddressBookW2JE.wsdl
WebServicesSamples.ear/AddressBookJ2WE.jar/META-INF/wsdl/AddressBookJ2WE.wsdl
WebServicesSamples.ear/AddressBookJ2WB.war/WEB-INF/wsdl/AddressBookJ2WB.wsdl
WebServicesSamples.ear/AddressBookW2JB.war/WEB-INF/wsdl/AddressBookW2JB.wsdl
```

Publishing WSDL files using a URL:

Before you can publish a Web Services Description Language (WSDL) file using a URL, the Web services-enabled application should be installed and running.

The files referenced by the `<wsdl-file>` element in the `webservices.xml` file can or cannot import other WSDL or XSD files. Typically, all WSDL or XSD files are originally placed into the `META-INF/wsdl` directory when using enterprise JavaBeans (EJBs) or the `WEB-INF/wsdl` directory when using Java beans. If your WSDL or XSD files are not placed in one of these directories, the file referenced by the `<wsdl-file>` and its imported files are located at the same directory and copied to the `wsdl/` directory for publishing purposes.

Note: EJB-based Web service applications must have an HTTP router or a Web module. Only HTTP URLs are supported for publishing.

To publish a WSDL file using a URL:

1. Retrieve the outer-most WSDL file. The outer-most WSDL file is the WSDL file defined by the `<wsdl-file>` element in the `webservices.xml` file.

   Each Web service has an endpoint address, like `http://example.com/services/stockquote`. You can retrieve the outer-most WSDL file (defined by the <wsdl-file> element within the webservices.xml file) by appending the string ″/wsdl″ or ″/wsdl/″ to the endpoint address, for example,`http://example.com/services/stockquote/wsdl`.
2. Retrieve the imported WSDL files. When the outer-most WSDL file imports other WSDL or XSD files, these imported files can be retrieved by appending the relative path to the URL, which is used to retrieve the outer-most WSDL file. This is also true for WSDL files that import other files. This process is similar to the use of relative hyperlinks in HTML documents. If an HTML document contains a hyperlink to other documents, the relative path is appended to create the URL to access the hyperlinked documents.

Suppose you have an application with the following directory structure:

```
<module-root>/
META-INF/
WEB-INF/
webservices.xml/* define Foo service, the <wsdl-file> element points to "/wsdl/fooImpl.wsdl"
*/ web.xml
ibm-webservices-bnd.xml
<jaxrpc-mapping-file>
wsdl/
fooImpl.wsdl/* importing foo.wsdl which is an interface wsdl */
foo.wsdl /* importing type definition for the interface */
fooTypes.xsd
```

If the SOAP address for the foo service is `http://examples.com:9080/services/foo`, the simple way to retrieve the foo's outer-most WSDL, is with the following form `http://examples.com:9090/services/foo/wsdl` or `http://examples.com:9090/services/foo/wsdl/`. The URL is redirected to `http://examples.com:9090/services/foo/wsdl/fooImpl.wsdl`, where `fooImpl.wsdl` is the name of the outer-most WSDL file.

Since the `fooImpl.wsdl` file has the import `<import namespace="http://examples.com/foo" location="a/b/foo.wsdl>`, use the URL `http://examples.com:9090/services/foo/wsdl/a/b/foo.wsdl` to obtain the `foo.wsdl` file.

Publish WSDL files settings:

Use this page to publish Web Services Description Language (WSDL) files.

To view this administrative console page, click **Applications** >**Enterprise Applications** > *application_instance* > **Publish WSDL**.

When you click **OK**, a zip file of all the Web services-enabled modules in the application is produced. The name of the published zip file is `application_name_WSDLFiles.zip`. In the published zip file, the directory structure is *application_name/module_name/[META-INF|WEB-INF]/wsdl/wsdl_file_name*.

In a published WSDL file, the location attribute of a service `soap:address` stanza contains the URL through which the Web service is accessed. You can specify the portion of the URL to be used for the Web services in each module. You can access the Web services in a module through a HTTP transport or JMS transport, or both. You can specify URL information for both types of transports.

Specify URL prefixes for Web Services:

Specifies the *protocol* (either http or https), *host_name*, and *port_number* to be used in the URL.

The URL prefix format is protocol://*host_name:port_number*, for example, http://*myHost*:9045. The actual URL that appears in a published WSDL file consists of the prefix prepended to the module's context-root and the Web service url-pattern, for example, http://*myHost*:9045/services/*myService*.

Select HTTP URL prefix:

Specifies the drop down list associated with a default list of URL prefixes. This list is the intersection of the set of ports for the module's virtual host and the set of ports for the module's application server. Use items from this list if the Web services application server is accessed directly.

To set an HTTP prefix, select either the **HTTP URL prefix** or **Custom HTTP URL prefix**, enter the value, select the check box of the modules that are to use the prefix, and click **Apply**. When you click **Apply**, the entry in the **Select HTTP URL prefix** or **Custom HTTP URL prefix** fields, depending on which is selected, is copied into the **HTTP URL prefix** field of any module whose check box (in the leftmost column) is selected. The HTTP prefix is not applied to the fields in the JMS URL prefix column.

Custom HTTP URL prefix:

Specifies the *protocol*, *host*, and *port_number* of the intermediate service if the Web services in a module are accessed through an intermediate node, for example the Web services gateway or an IHS server.

To set an HTTP prefix, select either the **HTTP URL prefix** or **Custom HTTP URL prefix**, enter the value, select the check box of the modules that are to use the prefix, and click **Apply**. When you click **Apply**, the entry in the **Select HTTP URL prefix** or **Custom HTTP URL prefix** fields, depending on which is selected, is copied into the **HTTP URL prefix** field of any module whose check box (in the leftmost column) is selected. The HTTP prefix is not applied to the fields in the JMS URL prefix column.

JMS URL prefix:

Specifies the JMS URL prefix string used for each module.

The URL prefix specified must contain the destination and connectionFactory properties. It can contain other property-value pairs, but it must not contain the targetService property, which is added by the system when the published WSDL files are created. The format of the JMS URL prefix is:

```
jms:/[queue&topic]?destination=target_queue_or_topic_jndi_name&connectionFactory=factory_jndi_name
```

For example, `jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF`.

The actual URL that appears in a published WSDL file consists of the prefix prepended to the Web service targetService. For example:

```
jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF&targetService=StockQuote
```

Multipart WSDL best practices: WebSphere Application Server supports deployment of Web services using a multipart Web Services Description Language (WSDL) file. That is, WSDL files import other WSDL files when the WSDL file listed in the `<wsdl-file>` element of the `webservices.xml` deployment descriptor contains all `<wsdl:service>` and `<wsdl:port>` elements. The WSDL file is divided into an implementation WSDL and an interface WSDL.

The `<wsdl:import>` element indicates a reference to another WSDL file. If the `<wsdl:import>` element `location` attribute does not contain a URL, that is, it contains only a file name, and does not begin with `http://`, `https://` or `file://`, the imported file must be located in the same directory and must not contain a relative path component. For example, if `META-INF/wsdl/A_Impl.wsdl` is in your module and contains the import statement `<wsdl:import="A.wsdl" namespace="..."/>` , the file, `A.wsdl` must also be located in the module `META-INF/wsdl` directory.

It is recommended that all WSDL files be placed in either the `META-INF/wsdl` directory, if you are using enterprise JavaBeans (EJBs), or the `WEB-INF/wsdl` directory, if you are using Java beans, even if there are relative imports within the WSDL files. Otherwise, there are implications with the WSDL publication when you use a path like the following `<location="../interfaces/A_Interface.wsdl"namespace="..."/>`. Using a path like this fails because the presence of the relative path, regardless of whether the file is located at that path or not. If the location is a URL, it must be readable at both deployment and server startup.

WSDL publication

The files located in the `META-INF/wsdl` or `WEB-INF/wsdl` directory can be published through either a URL or file, including WSDL or XSD files. For example, if the file referenced in the `<wsdl:file>` element of the `webservices.xml` deployment descriptor is located in the `META-INF/wsdl` or `WEB-INF/wsdl` directory, it is publishable. If the files imported by the `<wsdl:file>` are located in the `wsdl/` directory or its subdirectory, they are publishable.

If the WSDL file referenced by the `<wsdl:file>` element is located in a directory other than `wsdl`, or its subdirectories, the file and its imported files, either WSDL or XSD files, which are in the same directory, are copied to the `wsdl` directory without modification when the application is installed. These types of files can also be published.

If the `<wsdl:file>` imports a file located in a different directory, the file is not copied to the `wsdl` directory and not available for publishing.

Developing a Service Endpoint Interface for a Java bean implementation

Set up a Web services development and unmanaged client execution environment.

The Service Endpoint Interface defines the methods for a particular Web service. The Java bean implementation must implement methods having the same signature as the methods on the Service Endpoint Interface. There are a number of restrictions 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 Toolkit, which is a component of the Application Assembly Toolkit. The steps are similar except the Assembly Toolkit automatically compiles the interface when you save it.

To develop a Service Endpoint Interface for a Java bean implementation:
1. Create a Java interface containing the methods to include in the Service Endpoint Interface. The interface should extend the `java.rmi.Remote` interface. Each method throws the exception, `java.rmi.RemoteException`. If you start with an existing Java interface, remove any methods that do not conform to JAX-RPC.
2. Compile the interface. Use the **javac** commands for Windows and UNIX platforms listed in the topic Developing thin application client code to compile the interface. In the **javac** command, use the name of the Service Endpoint Interface class for the class to be compiled.

A Service Endpoint Interface which you can use to develop a Web service.

This example uses a Java interface called AddressBook. 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);
}
```

You use the AddressBook Java interface to create the Service Endpoint Interface:
1. Begin with the remote interface, `AddressBook.java`.
2. Make a copy of the remote interface named `AddressBook_SEI.java` and use it as a template for the Service Endpoint Interface.
3. Change the interface to extend the `java.rmi.Remote` interface.
4. Modify each method declaration to add a throws clause for `java.rmi.RemoteException`.
5. Compile the interface.

Use the Service Endpoint Interface to Develop a Web Services Description Language (WSDL) file.

Developing Web services deployment descriptor templates for a Java bean implementation

To develop the deployment descriptor templates from a Web Services Description Language (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 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 is a Java bean 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.

To develop deployment descriptor templates:
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 generated files displays when the command runs.

Deployment descriptor templates that are required to implement or use a Web service.

The following example uses a WSDL file named `AddressBookJ2WB.wsdl`:
1. 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 as follows:
   ```
   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
   ```

Developing a Web service using a stateless session enterprise bean

Set up a Web services development and unmanaged client execution 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 serializable and cannot be object references.
- 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 as follows:
1. Access an existing Java archive (JAR) file to be used as a Web service. Make sure that the enterprise bean meets the requirements.
2. Develop an 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.
4. Develop Web services deployment descriptor templates from an EJB implementation.
5. Assemble a Web services-enabled JAR file.
6. Configure the `webservices.xml` deployment descriptor.
7. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
8. Assemble a Web services-enabled enterprise archive (EAR) file.
9. 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.
10. Deploy the EAR file into WebSphere Application Server.

A Web service from a stateless session enterprise bean.

Developing a Service Endpoint Interface from an EJB remote interface

Set up a Web services development and unmanaged client execution environment.

The Service Endpoint Interface defines the Web services methods. The enterprise JavaBean (EJB) that implements the Web service must implement methods having the same signature as the methods of the Service Endpoint Interface. There are a number of restrictions 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 Toolkit, which is a component of the Application Server Toolkit. The steps are similar except the Assembly Toolkit automatically compiles the interface when you save it.

To develop a Service Endpoint Interface:
1. Create a Java interface containing the methods to include in the Service Endpoint Interface. The interface should extend the `java.rmi.Remote` interface. Each method throws the exception, `java.rmi.RemoteException`. If you start with an existing Java interface, remove any methods that do not conform to JAX-RPC.
2. Compile the interface. Use the **javac** commands for Windows and UNIX platforms listed in the topic Developing thin application client code to compile the interface. In the **javac** command, use the name of the Service Endpoint Interface class for the class to be compiled.

A Service Endpoint Interface which you can use to develop a Web service.

This example uses an EJB remote interface called AddressBook_RI.

```
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;
}
```

You use the `AddressBook_RI` remote interface to create the Service Endpoint Interface:
1. Begin with the remote interface, `AddressBook_RI.java`:
2. Make a copy of the remote interface named `AddressBook.java` and use it as a template for the Service Endpoint Interface.
3. Change the interface to extend the `java.rmi.Remote` interface, instead of the `javax.ejb.EJBObject` Service Endpoint Interface.
4. Compile the `AddressBook.java` Service Endpoint Interface.

Use the Service Endpoint Interface to Develop a WSDL file.

Developing Web services deployment descriptor templates for an EJB implementation

To develop the deployment descriptor templates from a Web Services Description Language (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 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 is an enterprise Java bean (EJB) 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.

To develop deployment descriptor templates:
Run the **WSDL2Java -verbose -role develop-server -container ejb -genJava no *wsdlURL*** command to generate the server deployment descriptor templates and mapping file into the `META-INF` subdirectory. If the **-verbose** option is specified, a list of all generated files displays when the command runs.

Deployment descriptor templates that are required to implement a Web service.

The following example uses a WSDL file named `AddressBookJ2WE.wsdl`:
1. Generate the template files:
   - `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
   ```

Completing the EJB implementation

Develop EJB implementation templates and bindings from a Web Services Description (WSDL) file.

To complete the EJB implementation:
1. Inspect the enterprise EJB remote interface template, `*portType*_RI.java`. If necessary, modify the template. *portType* is the name of the `<wsdl:portType>` element in the WSDL file.
2. Inspect the EJB home interface template, `*portType*Home.java`. If necessary, modify the template.
3. Edit the EJB implementation template, `*binding*Impl.java`. *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 EJB 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.

An EJB JAR file containing an EJB and supporting classes created from a WSDL file.

Configure the webservices.xml deployment descriptor .

Configuring the webservices.xml deployment descriptor

Create an enterprise JavaBean (EJB) Java archive (JAR) file or Web archive (WAR) file containing `webservices.xml`:

- Assemble a Web services-enabled EJB JAR file when starting from Java code.
- Assemble a Web services-enabled EJB JAR file from WSDL.
- Assemble a Web services-enabled WAR file when starting from Java code.
- Assemble a Web services-enabled WAR file when starting from WSDL.

Do one of the following based on whether your implementation is an EJB JAR file or Web module WAR file:

- Develop Web services Java bean deployment descriptor templates from a WSDL file.
- Develop Web services EJB deployment descriptor templates from a WSDL file.

This topic explains how to configure the `webservices.xml` deployment descriptor with the Assembly Toolkit which replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

To configure the `webservices.xml` deployment descriptor:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file or WAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** >**J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project containing the `webservicesclient.xml` file in the **Project Navigator** pane.
6. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents appear.
7. Right-click the `webservices.xml` file.
8. Select **Open**. The **Web Services** editor opens.
9. Expand the **Web service descriptions** section.
   a. Select the service you want to configure.
10. Expand the **Web service description implementation details** section.
    a. Verify the **Web service description name** field is unique among all the Web service descriptions in the editor.
    b. Verify that the **WSDL file** field indicates there is an existing WSDL file in the module. This file, by convention, should be located in the `META-INF/wsdl` directory for an enterprise bean JAR file and in the `WEB-INf/wsdl` directory for a WAR file.
    c. Verify the **JAX-RPC mapping file** field indicates an existing mapping file within the module. This file, by convention, should be located in the `META-INF` directory for an enterprise bean JAR file and in the `WEB-INF` directory for a WAR file.
11. Expand the **Port components** section.
    a. Verify there are port component entries corresponding to the used WSDL ports in the **Port components** section.
12. Select a *port_component* to open the editor for that port component. The **Port Components** editor opens.
13. Expand the **Port component implementation details** section.
    a. Verify the **WSDL Port Namespace URL** and **WSDL Port Local part** fields are set to the namespace and local name of the corresponding port in the WSDL file. These fields are configured by the **WSDL2Java** command tool when the `webservices.xml` file is generated.

14. Verify the **Service endpoint interface** field names the fully qualified Service Endpoint Interface class. This field is configured by the **WSDL2Java** command when the `webservices.xml` file is generated.
15. Locate the **Service implementation bean** field.
    a. Configure this field to indicate the EJB or servlet that implements the Web service. Configure by selecting **EJB link** for an enterprise bean module or **Servlet link** for a Web module. Use the drop down list in the **Service implementation bean** field to select the enterprise bean or servlet used to implement the Web service. The choices in the drop down menu come from the enterprise beans defined in the `ejb-jar.xml` file for an enterprise bean module or the servlets defined in the `web.xml` file for a Web module.

Configuring the ibm-webservices-bnd.xmi deployment descriptor

Develop implementation templates and bindings for the `ibm-webservices-bnd.xmi` from the Web Services Description Language (WSDL) file.

Do one of the following based on whether your implementation is an EJB Java archive (JAR) file or Web module Web archive (WAR) file:
- Assemble a Web services-enabled JAR file when starting from Java code.
- Assemble a Web services-enabled WAR file when starting from Java code.
- Assemble a Web services-enabled JAR file when starting from WSDL.
- Assemble a Web services-enabled WAR file when starting from WSDL.

This topic explains how to configure bindings using the Assembly Toolkit which replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface. .

To configure the `ibm-webservices-bnd.xmi` deployment descriptor with the Assembly Toolkit:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file or WAR file into the Assembly Toolkit.
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 containing the `webservices.xml` file in the **Project Navigator** pane.
6. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents appear.
7. Right-click the `webservices.xml` file.
8. Select **Open**. The **Web Services** editor opens.
9. Click the **Bindings** tab located at the bottom of the editor pane. The **Web Services Bindings** editor opens.
10. Verify the `wsdescNameLink` element settings.
    a. Expand the **Web services description bindings** section. Verify that the value of the `<webservice-description-name>` element in the `webservices.xml` deployment descriptor is listed in the section. If the value is not listed:
    b. (Optional) Click **Add**, choose the correct Web services name and click **OK**. You do not need to complete this step is you have verified that the correct Web services name is listed in the **Web Services Description Bindings** tab.
11. Verify the `pcnameLink` attribute settings.
    a. Expand the **Web Service Description Bindings** section. Verify that the correct service is selected. If the correct service is not listed:
    b. (Optional) Expand **Port Component Binding**. Verify the correct Web services name is selected in the **Web Service Description Bindings** section.

This selection is a prerequisite to creating a `pcnameLink` attribute.

c. In the **Port Component Binding** section, click **Add**. You need to make a selection in the **Web Service Description Bindings** section before you can create the port component binding in the **Port Component Binding** section. The **Port Component Bindings Dialog** opens.

d. Select the desired port from the drop down list in the **PC Name Link** field.

e. Click **OK**.

f. Click the **Binding Configurations** tab to view the bindings for your port.

g. **Save** the bindings file.

12. Click **File** > **Export** to export the JAR file, or continue using the Assembly Toolkit for configuration and assembly tasks.
13. Click **ctrl-s** to save your changes.

The `ibm-webservices-bnd.xmi` deployment descriptor is configured for the Web service implementation module.

ibm-webservices-bnd.xmi assembly properties

ibm-webservices-bnd.xmi properties

The `ibm-webservices-bnd.xmi` file is a deployment descriptor for a Web Services-enabled Web module or enterprise JavaBean (EJB) module. It contains information for the Web services runtime that is either WebSphere product-specific or was not specified by the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can edit these properties using the Assembly Toolkit. See Configuring the ibm-webservices-bnd.xmi deployment descriptor for instructions.

The following user-definable assembly properties are supported:

- **`wsDescNameLink`**

  Attribute of the `wsdescBindings` element that specifies the link to the corresponding `<webservice-description-name>` in `webservices.xml`.
- **pc-name-link**

  Attribute of the `pcBindings` element that specifies the link to the `<port-component-name>` in the `webservices.xml` file.
- **scope**

  Attribute of the `pcBindings` element that specifies when new instances of implementation beans are created. Possible values are Request, Session, and Application.

The value of scope for a deployed Web service can be changed using the administrative console. Using application management, navigate to the Web module of the Web service application and select Web Services Implementation Scope.

Example bindings file

The following examples demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute whose value is a unique string. Review the template `xmi` files generated by the **WSDL2Java** command for examples of ID strings.

```
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0" xmlns:xmi=
"http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
 <wsdescBindings wsDescNameLink="AddressBookService">
  <pcBindings pcNameLink="AddressBook" scope="Application"/>
 </wsdescBindings>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Configuring the webservices.xml deployment descriptor for Handler classes

This topic explains how to use the Assembly Toolkit to configure the `webservices.xml` deployment descriptor for user-provided Handler classes. The Assembly Toolkit is a component of the Application Server Toolkit. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

You should have an enterprise archive (EAR) file for the applications you want to configure. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file should contain the handler classes being configured. These classes implement the `javax.xml.rpc.handler.Handler` interface. For more information on writing handler classes, see Chapter 6 of the Web Services for J2EE 1.0 specification and chapter 12 of the JAX-RPC 1.0 specification available through Web services: Resources for learning. The application modules must contain the `webservices.xml`(for server) and `webservicesclient.xml` (for client) deployment descriptors.

To configure a handler in the `webservices.xml` deployment descriptor:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** and import the EAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** >**J2EE**.
4. Click the **Project Navigator** tab to switch to the **Project Navigator** pane.
5. Locate the project that contains the `webservices.xml` deployment descriptor. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents, including the `webservices.xml` file, are visible.
6. Right-click the `webservices.xml` file.
7. Click **Open**. The **Web Services** editor opens.
8. Expand the **Web services descriptions** section.
   a. Select the *service* for which you want to configure the handler.
9. Expand the **Port components** section.
10. Select a *port_component* for which you want the editor to open. The **Port Components** editor opens.
11. Expand the **Port component handlers** section.
12. Click **Add** at the bottom of the **Port component handlers** section. A **Class browser** opens.
13. Browse for the name of the Handler class in the module. When it displays in the **Matching types** field, select the class and click **OK**. The Class browser window closes after you click OK and the **Handlers** pane of the **Web Services Editor** opens.
14. (Optional) Configure properties in the **Handlers** pane. See Handler class properties for a list of the properties you can configure in this step.
15. Type **ctrl-s** to save the changes.

Developing a new Web service with an existing WSDL file using a Java bean

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).

To develop a new Web service with an existing WSDL file using a Java bean:

1. Develop Java bean implementation templates and bindings from a WSDL file.
2. Complete the Java bean implementation.
3. Assemble a Web services-enabled Web archive (WAR) file when starting from a WSDL file.
4. Configure the `webservices.xml` deployment descriptor.
5. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
6. Assemble a Web services-enabled WAR into an EAR file.
7. Deploy the EAR file into WebSphere Application Server.

Develop Web services deployment descriptor templates from a WSDL file.

You can either 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 Java bean implementation

To develop the Java bean implementation templates and bindings from a Web Services Description (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 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.

Implementation templates are generated using the -role develop-server option of the **WSDL2Java** command. The **WSDL2Java** command also generates bindings and deployment descriptors.

To develop Java bean implementation templates and bindings from a WSDL file:
Run the **WSDL2Java -verbose -role develop-server -container web *wsdlURL*** command. Since the **verbose** option is specified, a list of all generated files is displayed when the command runs.

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 an Java bean named AddressBook and a WSDL file named `AddressBook.wsdl`. 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 generated file named `AddressBookSOAPBindingImpl.java` 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.

Completing the Java bean implementation

Develop Java bean implementation templates and bindings from a Web Services Description Language (WSDL) file.

1. Edit the Java bean implementation template, *binding*Impl.java. *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 typical 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.

A Java archive (JAR) file containing a Java bean and supported classes created from the WSDL file.

Configure the `webservices.xml` deployment descriptor.

Developing a new Web service from an existing WSDL file using a stateless session enterprise bean

Set up a Web services development and unmanaged client execution 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.

Create the enterprise bean and artifacts that enable the enterprise bean to be a Web service and assemble those artifacts into the enterprise application as follows:

1. Develop implementation templates and bindings from a WSDL file.
2. Complete the enterprise bean implementation.
3. Assemble a Web services-enabled enterprise EJB Java archive (JAR) file.
4. Configure the `webservices.xml` deployment descriptor.
5. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
6. Assemble a Web services-enabled EJB JAR into an EAR file.
7. 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** command or Assembly Toolkit before deployment.
8. Deploy the EAR file into WebSphere Application Server.

An EJB implementation of a Web service defined in the WSDL file.

Developing EJB implementation templates and bindings from a WSDL file

To develop enterprise JavaBean (EJB) implementation templates and bindings from a Web Services Description (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 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.

Implementation templates are generated using the -role develop-server option of the **WSDL2Java** command.

Templates are generated for an EJB implementation for the following:
- EJB
- EJB remote interface
- EJB Home

The **WSDL2Java** command also generates bindings and deployment descriptors.

To develop implementation templates and bindings from a WSDL file:
Run the **WSDL2Java -verbose -role develop-server -container ejb *wsdlURL*** command. Since the **verbose** option is specified, a list of all generated files is displayed when the command runs.

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 an enterprise bean named AddressBook and a WSDL file named `AddressBook.wsdl`. 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.

Web services implementation scope

Use this page to view and manage the scope of the ports of a Web Service bean.

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 scope determines when a new instance of a service implementation is created for the Web service ports in a module. An application scope causes the same instance of the implementation to be used for all requests on the application. A session scope causes the same instance to be used for all requests on each session. A request scope causes a new instance to be used on every request.

Default Port Mapping Definitions collection

Use this page to view and manage a default port type mapping for a Web service.

To view this page of the Administrative Console, click **Applications** >**Enterprise Application** > *application_instance* > **Web Modules** > *module_instance*>**Web Services Client Bindings** > **Edit** > *default_port_instance*.

For EJB modules, click **Applications** >**Enterprise Application** > *application_instance* > **EJB Modules** > *module_instance*>**Web Services Client Bindings** > **Edit** > *default_port_instance*.

Specify the default port of a service when a particular port type is requested. The port type is described by its local name and namespace. A getPort method specifying only the port type gets the port named by the default port local name and namespace.

Port Type Local Name
Specifies the name of this Web service.

Port Type Namespace
Specifies the local name describing the port type to be mapped.

Default Port Local Name
Specifies the namespace describing the port type to be mapped.

Default Port Namespace
Specifies the namespace of the port to map to.

Default Port Type Mapping Properties settings

Use this page to view and manage a default port type mapping for a Web service.

To view this page of the Administrative Console, click **Applications** >**Enterprise Application** > *application_instance* > **Web Modules** > *module_instance*>**Web Services Client Bindings** > **Edit** > *default_port_instance*.

For EJB modules, click **Applications** >**Enterprise Application** > *application_instance* > **EJB Modules** > *module_instance*>**Web Services Client Bindings** > **Edit**> *default_port_instance*.

Specify the default port of a service when a particular port type is requested. The port type is described by its local name and namespace. A getPort method specifying only the port type gets the port named by the default port local name and namespace.

Port Type Local Name
Specifies the local name of the port type to be mapped.

Port Type Namespace
Specifies the namespace of port type to be mapped.

Default Port Local Name
Specifies the local name of the port to map to.

Default Port Namespace
Specifies the namespace of the port to map to.

Developing Web services clients based on Web Services for J2EE

This topic explains how to develop a Web services client based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

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's 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.

Before you begin this task, locate the Web Services Description Language (WSDL) file that defines the Web service to access.

To create the client code and artifacts that enable the application client to access a Web service:

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 `webservicesclient.xml` 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.

You have created and tested a Web services client application. For step-by-step information see Example: Developing Web services clients based on Web Services for J2EE.

Example: Developing Web services clients based on Web Services for J2EE

This example takes you through the steps to develop a Web service 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. For a Java or J2EE application to act as a client of a Web service, you must map the WSDL file to the Java code. The JAX-RPC specification defines the mapping between a WSDL file, Java code and XML Schema types.

Steps for this example task

1. Obtain the Web Services Description Language (WSDL) document for the Web service that you want to access.

   You can obtain the WSDL document from the service provider by e-mail or by looking it up in a Universal Description, Discovery and Integration (UDDI) registry.
2. Develop client bindings from your WSDL file.

   The WSDL document is used to generate all the information needed to invoke the Web service, including the Service Endpoint Interface and implementations; generated service interface; `webservicesclient.xml` and `ibm-webservicesclient-bnd.xmi` and `ibm-webservicesclient-ext.xmi` deployment descriptors.

   The **WSDL2Java** command-line tool is run against your WSDL file to develop client bindings.
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 sample 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 webservicesclient.xml deployment descriptor.

   Configure the ibm-webservicesclient-bnd.xmi deployment descriptor.
6.
7. 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

To develop the client bindings from a Web Services Description (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 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.

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 the application client container.
- **-container ejb**

  Generates bindings and deployment descriptors for a client that is an EJB in the EJB module.
- **-container web**

  Generates bindings and deployment descriptors for a client residing in the Web container.

To develop client bindings from a WSDL file:
Run the **WSDL2Java -verbose -role develop-client -container *type wsdlURL*** command.

Where *type* is **ejb** for an enterprise JavaBean (EJB) client, **web** for a Java bean client, or **client** for an application client.

Note: You can have:
- -container web
- -container ejb
- -container client

Since the **verbose** option is specified, a list of all generated files is displayed when the command runs.

The bindings and deployment descriptors needed by a client to use a Web service.

The following example uses an enterprise bean named AddressBook and a WSDL file named `AddressBook.wsdl`. 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\webservicesclient.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-ext.xmi.
```

Complete the client implementation.

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

Assembling a Web services-enabled client JAR file into an EAR file

You need the following artifacts:
- Assembled client module, containing the implementation, all classes generated by the **WSDL2Java** command-line tool, `MANIFEST.MF` and deployment descriptor. This module can be:
  – An application client module containing `META-INF/application-client.xml`
  – An enterprise JavaBean (EJB) module containing `META-INF/ejb-jar.xml`
- Web Services Description Language (WSDL) file used to develop the client
- Templates for `webservicesclient.xml` and `ibm-webservicesclient-ext.xmi` deployment descriptors, if used.
- Generated JAX-RPC mapping deployment descriptor

You can use the Assembly Toolkit to assemble Web service-enabled client applications.

To assemble the client code and artifacts that enable the application client to access a Web service:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file, App Client JAR file, or WAR file into the Assembly Toolkit.
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 file you just imported in the **Project Navigator** pane.
6. Expand the `ejbModule` (for an EJB JAR file) or the `appClientModule` (for the application client JAR file) entry so 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.
   a. Copy the WSDL file to the `META-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
   b. Copy the `webservicesclient.xml` and the JAX-RPC mapping file in the `META-INF` subdirectory in the same manner you copied the WSDL file. The JAX-RPC mapping file is indicated by the `<jaxrpc-mapping-file>` element in the `webservicesclient.xml` file.
   c. (Optional) Place the `ibm-webservicesclient-ext.xmi` and the `ibm-webservicesclient-bnd.xmi` file in the `META-INF` subdirectory, if used.
8. Assemble the JAR file into an EAR file using typical assembly techniques if the client runs in a container.
9. Right-click on the `WEB-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `WEB-INF` directory.
   a. Copy the WSDL file to the `WEB-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
   b. Copy the `webservicesclient.xml` and the JAX-RPC mapping file in the `WEB-INF` subdirectory in the same manner you copied the WSDL file. The JAX-RPC mapping file is indicated by the `<jaxrpc-mapping-file>` element in the `webservicesclient.xml` file.
   c. (Optional) Place the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file in the `WEB-INF` subdirectory, if used.

The artifacts required to enable the client module to use Web services are added to the module.

This example uses a JAR file named `AddressBookClient.jar` and an EAR file named `AddressBookClient.ear`:

```
META-INF/MANIFEST.MF
META-INF/application-client.xml
META-INF/wsdl/AddressBook.wsdl
META-INF/webservicesclient.xml
META-INF/AddressBook_mapping.xml
com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookClient.jar` file into the `AddressBookClient.ear` file, the `AddressBookClient.ear` file contains the following files:

```
META-INF/MANIFEST.MF
AddressBookClient.jar
META-INF/application.xml
```

Configure the webservicesclient.xml deployment descriptor .

Assembling a Web services-enabled client WAR file into an EAR file

You need the following artifacts:
- Assembled client Web archive (WAR) module, containing the implementation, all classes generated by the **WSDL2Java** command-line tool, `MANIFEST.MF` and deployment descriptor.
- Web Services Description Language (WSDL) file used to develop the client
- Templates for `webservicesclient.xml`, `ibm-webservicesclient-bnd.xmi` and `ibm-webservicesclient-ext.xmi` deployment descriptors, if used.
- Generated JAX-RPC mapping deployment descriptor

You can use the Assembly Toolkit to assemble Web service-enabled client applications.

To assemble the client code and artifacts that enable the application client to access a Web service:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the WAR file into the Assembly Toolkit.
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 file you just imported in the **Project Navigator** pane.
6. Expand the `webContemt` entry so the `WEB-INF` directory is displayed. Expand the `WEB-INF` directory.
7. Right-click on the `WEB-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `WEB-INF` directory.
   a. Copy the WSDL file to the `WEB-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
   b. Copy the `webservicesclient.xml` and the JAX-RPC mapping file in the `WEB-INF` subdirectory in the same manner you copied the WSDL file. The JAX-RPC mapping file is indicated by the `<jaxrpc-mapping-file>` element in the `webservicesclient.xml` file.
   c. (Optional) Place the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file in the `WEB-INF` subdirectory, if used.
8. Assemble the WAR file into an EAR file using typical assembly techniques.

The artifacts required to enable the client module to use Web services are added to the module.

This example uses a WAR file named `AddressBookWeb.war` and an EAR file named `AddressBook.ear`:

```
WEB-INF/MANIFEST.MF
WEB-INF/web.xml
WEB-INF/wsdl/AddressBook.wsdl
```

```
WEB-INF/webservicesclient.xml
WEB-INF/AddressBook_mapping.xml
WEB-INF/ibm-webservicesclient-ext.xmi (optional)
WEB-INF/ibm-webservicesclient-bnd.xmi
com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookWeb.war` file into the `AddressBook.ear` file, the `AddressBook.ear` file contains the following files:

```
WEB-INF/MANIFEST.MF
AddressBookWeb.war
WEB-INF/application.xml
```

Configure the webservicesclient.xml deployment descriptor .

Configuring the ibm-webservicesclient-bnd.xmi deployment descriptor

This topic explains how to configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file using the Assembly Toolkit, which replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To configure the`ibm-webservicesclient-bnd.xmi` deployment descriptor file:

1. Start the Assembly Toolkit
2. Locate the `webservicesclient.xml` file in the module.
3. Double-click the `webservices.xml` file to open the **Web Services Client** editor.
4. Access the **Web Services Client Bindings** editor through the **Client Binding** tab at the bottom of the editor pane.
5. Verify the `componentNameLink` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Expand the **Component scoped references** section.
   c. Click **Add**.
   d. Select the component scoped references defined in the `webservicesclient.xml` file from the list.
6. Verify the `serviceRefLink` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Click the **Services References** tab.
   c. Click **Add**.
   d. Select the service references defined in the `webservicesclient.xml` file from the list.
7. Verify the `deploydWSDLFile` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Select the service references or component scoped reference desired.
   c. Expand the **Service reference details** section.
   d. Click **Browse** on the **Deployed WSDL file** field.
   e. Select the new WSDL file.
   f. Click **OK**.

   The `deployedWSDLFile` element of a deployed Web service can also be changed using the administrative console. Using application management, navigate to the Web module or EJB module of the Web service application and select **Web Services Client Bindings**.
8. Verify the `defaultMappings` element settings.

a. Open the **Web Services Client Bindings** editor.
b. Click **Default Mappings**.
c. Click **Add**.
d. Edit the entries in the newly added row to establish a mapping between a *portType* and *port* in the WSDL file. There can only be one entry for each *portType*.
e. Select the new WSDL file.
f. Click **OK**.

The `defaultMappings` of a deployed Web service can also be changed using the administrative console. Using application management, navigate to the Web module or EJB module of the Web service application and select **Web Services Client Bindings**.

9. Access the **Web Services Client Port Bindings** editor through the **Port Bindings** tab at the bottom of the editor pane.
10. Verify the `syncTimeout` element settings.
    a. Create a **Port Qualified Name Bindings** for the port.
    b. Open the **Web Services Client Bindings** editor.
    c. Confirm that a service reference is selected in either the **Component scoped references** or **Service references** section.
    d. Expand the **Port qualified name bindings** section.
    e. Click **Add**. A new entry is now added to the **Port qualified name bindings** list.
    f. Click the new **Port qualified name bindings** entry. The **Web Services Client Port Bindings** editor opens.
    g. Expand the **Port qualified name bindings details** section.
    h. Type the *namespace* of the WSDL file port you want to configure, in the **Port Namespace Link** field.
    i. Type the *local_name* of the WSDL file port you want to configure in the **Port Local Name Link**field. The name displayed in the **Port qualified name bindings** list is the local name of the WSDL file port.
    j. Click **OK**.
    a. Configure the `syncTimeout` property by locating the **Synchronization timeout** field and enter the desired value.
11. Verify the `basicAuth` element settings.
    a. Open the **Web Services Client Bindings** editor.
    b. Expand the **Basic authentication** section.
    c. Type the desired value in the **User ID** and **Password** fields.
    d. Click **OK**.
12. Verify the `sslConfig` element settings.
    a. Open the **Web Services Client Bindings** editor.
    b. Expand the **SSL Configuration** section.
    c. Type the desired value in the **Name** field.
    d. Click **OK**.
13. After editing the properties, type **ctrl-s** on your keyboard to save the changes.

ibm-webservicesclient-bnd.xmi assembly properties

The `ibm-webservicesclient-bnd.xmi` file contains information for the Web services runtime that is WebSphere product-specific.

Assembly properties

The following user-definable assembly properties are supported:

- **componentNameLink**

  Attribute of the componentScopedRefs element that specifies the link to the corresponding <component-scoped-refs> element in webservicesclient.xml file. This property is used only when the Web service client is an EJB.
- **serviceRefLink**

  Attribute of the serviceRefs element that specifies the link to the <service-ref-name> in the webservicesclient.xml file.

  You can edit this property in the Assembly Toolkit:
- **deployedWSDLFile**

  Attribute of the serviceRefs element is optional and permits an alternate WSDL file to be used other than that specified in the <wsdl-file> element of webservicesclient.xml file. If this attribute is specified, the alternate WSDL file must be packaged in the same module and must be compatible with the development WSDL file. The deployedWSDLFile property is used to supply a new WSDL file containing a different endpoint URL than the original WSDL file.
- **defaultMappings** element

  Identifies which port should be used for a given portType when none is explicitly selected by the client. This element has the following attributes: portTypeNamespace, portTypeLocalName, portNamespace, portLocalName. These attributes identify which wsdl:port should be used for a wsdl:portType.
- **syncTimeout**

  Attribute of the portQnameBindings element that specifies how long, in seconds, to wait for a response from a synchronous call.
- **basicAuth**

  Element of the portQnameBindings element that can be used to authenticate a service client to the service endpoint, independent of the underlying transport that includes, HTTP, HTTPS, and JMS. Set the user ID and password attributes as needed.
- **sslConfig**

  Element of the portQnameBindings element that specifies the Secure Sockets Layer (SSL) configuration of an HTTPS outbound request. The name attribute is the name of a SSL configuration entry or alias defined in the SSL Configuration Repertoire.

  Note: This attribute is only used when the client is running in the WebSphere Application Server.

Example bindings file

The following example demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute whose value is a unique string. Review the template xmi files generated by the **WSDL2Java** command for examples of ID strings.

```
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">

 <componentScopedRefs componentNameLink="myComponent ref"/>

 <serviceRefs serviceRefLink="myService ref" deployedWSDLFile="META-INF/wsdl/alternate.wsdl">
  <defaultMappings portTypeLocalName="AddressBook" portTypeNamespace="http://www.com.ibm"
portLocalName="AddressBookPort" portNamespace="http://www.com.ibm"/>
   <portQnameBindings portQnameNamespaceLink="http://www.com.ibm"
portQnameLocalNameLink="AddressBookPort" syncTimeout="99">
    <basicAuth userid="myId" password="myPassword"/>
    <sslConfig name="mynode/DefaultSSLSettings"/>
   </portQnameBindings>
  </serviceRefs>
 </com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Configuring the webservicesclient.xml deployment descriptor

You should have an enterprise JavaBean (EJB) Java archive (JAR) file, Web archive (WAR) file or an application client file that you can import into the Assembly Toolkit.

This topic explains how to configure the `webservicesclient.xml` deployment descriptor with the Assembly Toolkit. It is one of the tools available with the Application Server Toolkit product. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

To configure the `webservicesclient.xml` deployment descriptor with the Assembly Toolkit:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file, WAR file or application client file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** >**J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project containing the `webservicesclient.xml` file in the **Project Navigator** pane.
6. Expand the directories under the project until the META-INF or WEB-INF directory and its contents appear.
7. Right-click on the `webservicesclient.xml` file.
8. Select **Open**. **The Web Services Client** editor opens.
9. Expand the **Service references** section.
10. Select the *service_reference* that you want to configure.
11. Expand the **Service reference overview** section.
12. Type the name of the service for which the client accesses in the **Description** field.
13. Expand the **Service reference implementation details** section.
    a. Type the name that the Java Naming Directory Interface (JNDI) uses to locate the service in the **Service references name** field. The JNDI lookup string for this service is `java:comp/env/service-ref-name`. By convention, the service reference name always begins with `service/`.
    b. Type the class name, including package, of the generated Java interface that is the Service Interface for this Web service in the **Service interface name** field.
    c. Type the WSDL file name used by the client, relative to the root of the module, in the **WSDL file** field.
    d. Type the file name of the Java mapping file, relative to the root of the module, in the **JAX RPC mapping file** field.
14. Click **ctrl-s** to save the changes.

The `webservicesclient.xml` deployment descriptor is configured.

Configuring the webservicesclient.xml deployment descriptor for Handler classes

This topic explains how to use the Assembly Toolkit to configure the `webservicesclient.xml` deployment descriptor for user-provided Handler classes. The Assembly Toolkit is a component of the Application Server Toolkit. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

You should have an Enterprise archive (EAR) file for the applications you want to configure. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file should contain the handler classes being configured. These classes implement the `javax.xml.rpc.handler.Handler` interface. For more information on writing handler classes, see Chapter 6 of the Web Services for Java 2 platform, Enterprise Edition (J2EE) 1.0 specification and chapter 12 of the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification available through Web services: Resources for learning. The application modules must contain the `webservices.xml`(for server) and `webservicesclient.xml` (for client) deployment descriptors.

To configure a handler in the `webservicesclient.xml` deployment descriptor:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** and import the EAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** >**J2EE**.
4. Click the **Project Navigator** tab to switch to the **Project Navigator** pane.
5. Locate the project that contains the `webservicesclient.xml` deployment descriptor. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents, including the `webservicesclient.xml` file, are visible.
6. Right-click the `webservicesclient.xml` file.
7. Click **Open**. The **Service References** pane of the **Web Services Client** editor opens.
8. Expand the **Service references** section.
   a. Select the *service_reference* for which you want to configure the handler.
9. Expand the **Handlers** section.
10. Click **Add** at the bottom of the **Handlers** section. A **Class browser** opens.
11. Browse for the name of the Handler class in the module. When it displays in the **Matching types** field, select the class and click **OK**. The Class browser window closes after you click OK and the **Handlers** pane of the **Web Services Editor** opens.
12. (Optional) Configure properties in the **Handlers** pane. See Handler class properties for a list of the properties you can configure in this step.
13. Type **ctrl-s** to save the changes.

Handler class properties

You can configure the following Handler class properties through the Assembly Toolkit. See Configuring the webservices.xml deployment descriptor for Handler classes or Configuring the webservicesclient.xml deployment descriptors for Handler classes for instructions on how to configure the properties.

Description

Standard Java 2 platform, Enterprise Editioin (J2EE) technology descriptor field.

Display name

Standard J2EE technology descriptor field.

Small icon

Standard J2EE technology descriptor field.

Large icon

Standard J2EE technology descriptor field.

Handler name

The name of the handler. This name must be unique within the module.

Handler class

The fully qualified name of the Handler class. Initially, it is set by the Assembly Toolkit's class browser.

Initial parameters

Property names and values to be made available to the handler.

SOAP headers

Qnames of the SOAP headers that are processed by this handler. See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification, available through Web services: Resources for learning, for more information about setting this property.

SOAP roles

URIs containing the SOAP actor names for which the handler acts in the role of. See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification, available through Web services: Resources for learning, for more information about setting this property.

Example: Configuring Handler classes for Web services deployment descriptors

This scenario explains how to add trivial client and server Handler classes to a sample application named `WebServicesSamples.ear`. The Handler classes display messages when given a request or response to handle.

The code for the client Handler class is:

```
 package samples;

public class ClientHandler implements javax.xml.rpc.handler.Handler {
     public ClientHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleRequest");
   return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleResponse");
   return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleFault");
   return true;  }

public void init(javax.xml.rpc.handler.HandlerInfo config) { }

public void destroy() {
  }

public javax.xml.namespace.QName[] getHeaders() {
   return null; }
}
```

The code for the server Handler class is:

```
 package sample;
public class ServerHandler implements javax.xml.rpc.handler.Handler {
  public ServerHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleRequest");
   return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleResponse");
   return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleFault");
   return true; }

public void init(javax.xml.rpc.handler.HandlerInfo config) { }

public void destroy() {  }

public javax.xml.namespace.QName[] getHeaders() {
   return null;  }
}
```

1. Compile these classes using

   %JAVA_HOME%\bin\java -extdirs %WAS_EXT_DIRS% ClientHandler.java ServerHandler.java (on Windows)

   $JAVA_HOME/bin/java -extdirs $WAS_EXT_DIRS ClientHandler.java ServerHandler.java (on Unix)
2. Open the Assembly Toolkit and import the two sample EAR files:
   - %WAS_HOME%\samples\lib\WebServicesSamples\WebServicesSamples.ear on Windows or $WAS_HOME/samples/lib/WebServicesSamples/WebServicesSamples.ear on Unix.
   - %WAS_HOME%\samples\lib\WebServicesSamples\ApplicationClients.ear on Windows or $WAS_HOME/samples/lib/WebServicesSamples/ApplicationClients.ear on Unix..
3. Import the compiled handler classes into the projects for the sample modules:
   - Import sample.ClientHandler into the **appClientModule** directory of the **AddressBookClient** project.
   - Import sample.ServerHandler into the **ejbModule** directory of the **AddressBookW2JE** project.
4. Configure the webservicesclient.xml deployment descriptor for Handler classes.
5. Configure the webservices.xml deployment descriptor for Handler classes.
6. Save your changes and export the EAR files.
7. Uninstall the `WebServicesSamples.ear` application from your server if it is already installed.
8. Install the new `WebServicesSamples.ear` application.
9. Start the server.
10. Run the client:

    launchClient ApplicationClients.ear -CCjar=AddressBookClient.jar

    When the client executes, the console output is as shown below. The messages from the handlers are shown in bold.

    ```
    IBM WebSphere Application Server, Release 5.1
    J2EE Application Client Tool
    Copyright IBM Corp., 1997-2003
    WSCL0012I: Processing command line arguments.
    WSCL0013I: Initializing the J2EE Application Client
    Environment.
    WSCL0035I: Initialization of the J2EE Application Client
    Environment has completed.
    ```

```
WSCL0014I: Invoking the Application Client class
com.ibm.websphere.samples.webservices.addr.AddressBookClient
>> Querying address for 'Purdue Boilermaker' using port
AddressBookW2JE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       1 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       2 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       3 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port AddressBookW2JB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       4 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
```

For the client, the Handler class is configured for each service reference, not for each port. The `AddressBook` sample has four ports, but only one service reference, therefore the `ClientHandler` handles requests and responses on all ports.

When the server log file is examined, it contains:

```
[9/24/03 16:39:22:661 CDT] 4deec1c6 WebGroup      I SRVE0180I:
[HTTP router for AddressBookW2JE.jar] [/AddressBookW2JE] [Servlet.LOG]:
AddressBook: init
[9/24/03 16:39:23:161 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleRequest
[9/24/03 16:39:23:211 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleResponse
```

What to do next

Install and test the application.

Testing Web services-enabled clients

Before testing your Java client, confirm that the server endpoint specified in the client Web Services Description Language (WSDL) file is running and available.

The following steps and examples assume that you are testing a system that has WebSphere Application Server installed, and that you have configured your environment as described in Setting up a Web services development and unmanaged client execution environment.

Tests are run differently depending on whether the client module has client container deployment information, which consists of the `application-client.xml` and `webservicesclient.xml` files, as well as the JAX-RPC mapping file and WSDL file. The client enterprise archive (EAR) files discussed in this topic are referred to as managed because they contain the deployment information. The client Java archive (JAR) files discussed are referred to as unmanaged because they that do not contain the deployment information.

To test Web services-enabled clients:

1. Test an unmanaged client JAR file.
   a. Execute your application with the **java** command. On Windows platforms:
      ```
      "%JAVA_HOME%\bin\java" "-Xbootclasspath/p:%WAS_BOOTCLASSPATH%"
      -Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"
      -Djava.ext.dirs="%WAS_EXT_DIRS%"
      -classpath "%WAS_CLASSPATH%;<list your application JAR files and classes>"
      <fully qualified class name to run><your application parameters>
      ```
      On UNIX:
      ```
      "$JAVA_HOME/bin/java" "-Xbootclasspath/p:$WAS_BOOTCLASSPATH"
      -Djava.security.auth.login.config="$WAS_HOME/properties/wsjaas_client.conf"
      -Djava.ext.dirs="$WAS_EXT_DIRS"
      -classpath "$WAS_CLASSPATH;<list of your application JAR files and classes>
      <fully qualified class name to run><your application parameters>
      ```
   The unmanaged client application runs.
2. Test a managed client EAR file.
   a. Execute your client application with the **launchClient** command. An example of using the command is as follows:
      ```
      launchClient clientEar
      ```

Web services-enabled clients that have been tested.

Troubleshoot your Web services application.

Web services client bindings

Use this page to specify the Web Service Description Language (WSDL) file name and default port type mappings for the Web services in a module.

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

Specifies the name of this 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 module.

WSDL Filename

Specifies the WSDL file name, which is relative to the 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. This is needed only 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 default port mapping specifies the port to use for a port type.

Default Port Mappings

Specifies and manages the default port type mapping for a Web service when a particular port type is requested.

Assembling Web services applications based on Web Services for J2EE

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 the Assembly Toolkit which replaces the Application Assembly Tool (AAT). The Assembly Toolkit is one of the tools available with the Application Server Toolkit product. .

To assemble Web services applications:
1. Start the Assembly Toolkit.
2. Assemble a Web services-enabled EJB JAR file.
3. Assemble a Web services-enabled EJB JAR file into an EAR file.
4. (Optional) Enable the EAR file. When the EAR file contains EJB modules, it must have the Web services endpoint WAR file added with the **endptEnabler** command-line tool or Assembly Toolkit before deployment.
5. Assemble a Web services-enabled WAR file.
6. Assemble a Web services-enabled WAR file into an EAR file.

A Web services-enabled EAR file that you can deploy into WebSphere Application Server.

Deploy the Web services-enabled EAR file into WebSphere Application Server.

Assembling a Web services-enabled EJB JAR file

You can assemble a Web services-enabled enterprise JavaBean (EJB) Java archive (JAR) file in one of two ways:
1. Assemble a Web services-enabled EJB JAR file when starting from Java code.
2. Assemble a Web services-enabled EJB JAR file when starting from Web Services Description Language (WSDL).

An assembled Web services-enabled EJB JAR file.

Configure the webservices.xml deployment descriptor .

Assembling a Web services-enabled EJB JAR file when starting from Java code

You need the following artifacts:
- Assembled Enterprise JavaBean (EJB) Java archive (JAR) file (not enabled for Web services)
- Compiled Java class for the Service Endpoint Interface
- Web Services Description Language (WSDL) file
- Complete `webservices.xml`,`ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi` and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

This topic explains how to assemble a Web service-enabled EJB JAR file with the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Tool (AAT) and is one of the tools available with the Application Server Toolkit product.

To assemble an Web services-enabled EJB JAR file when starting from Java code:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file into the Assembly Toolkit.

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 containing the JAR file you just imported in the **Project Navigator** pane.
6. Expand the `ejbModule` entry until the `META-INF` directory displays. 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 on 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, `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 its package begins in the `ejbModule` directory. You can import either the source file or compiled class file. If you import the source file it automatically compiles.

The artifacts required to Web service-enable an EJB module for Web services are added to the JAR file.

After assembling a JAR file named `AddressBook.jar`, the JAR file contains the following files. The files added in this task are in bold:

```
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
```

Configure the webservices.xml deployment descriptor .

Assembling Web services-enabled EJB JAR file when starting from WSDL

You need the following artifacts:
- An assembled Enterprise JavaBean (EJB) Java archive (JAR) file containing the EJB implementation and all classes generated by the **WSDL2Java** command tool when the **role** argument is develop-server and the **container** argument is EJB.
- A Web Services Description Language (WSDL) file
- Complete `webservices.xml`, `ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi`, and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

You can use the Assembly Toolkit to assemble Web services-enabled JAR files. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled EJB JAR file when starting from WSDL:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file into the Assembly Toolkit.
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 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 on 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 `webservices.xml`.
10. Copy `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` into the `META-INF` subdirectory in the same manner.

The artifacts required to enable an EJB module for Web services are added to the JAR file.

After assembling a JAR file named `AddressBook.jar` contains the following files. The files added in this task are in bold:

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookSoapBindingImpl.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

Configure the webservices.xml deployment descriptor .

Assembling a Web services-enabled WAR file

You can assemble a Web services-enabled Web archive (WAR) file in one of two ways:
1. Assemble a Web services-enabled WAR file when starting from Java code.
2. Assemble a Web services-enabled WAR file when starting from WSDL.

A Web services-enabled WAR file is assembled.

Assembling a Web services-enabled WAR file when starting from Java code

You need the following artifacts:
- An assembled Web archive (WAR) file containing `web.xml`, but not Web services-enabled
- The Java class for the Service Endpoint Interface
- A Web Services Description Language (WSDL) file
- Complete `webservices.xml`,`ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi`, and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

You can use the Assembly Toolkit to assemble a Web services-enabled WAR file. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled WAR file when starting from Java code:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the WAR file into the Assembly Toolkit.
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 WAR file you just imported in the **Project Navigator** pane.
6. Expand the `WebContent` directory so the `WEB-INF` directory is displayed. Expand the `WEB-INF` directory
7. Confirm that the `WEB-INF/web.xml` descriptor for the Web module contains a `<servlet-class>` element indicating the Java bean class that is implementing the service. Confirm by:
   a. Double-click **Web Deployment Descriptor**.
   b. In the **Web Deployment Descriptor** editor, click the **Servlets**.
   c. Enter the full path name of the Java bean class implementing 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 on the `wsdl` directory and click **File** > **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 `webservices.xml`.
11. Copy `webservices.xml`,`ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` into the `WEB-INF` subdirectory in the same manner.
12. Import the Service Endpoint Interface class so that its package begins in the `JavaSource` directory. When you import the source file it is automatically compiled.

The artifacts required to Web service-enable the Web module are added to the WAR file.

Assemble a Web services-enabled WAR into an EAR file.

Assembling a Web services-enabled WAR file when starting from WSDL

You need the following artifacts:
- Assembled Web archive (WAR) file containing the enterprise JavaBean (EJB) implementation, all classes generated by the **WSDL2Java** command tool and a Web deployment descriptor, `web.xml`.
- A Web Services Description Language (WSDL) file
- Complete `webservices.xml`, `ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi`, and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

You can use the Assembly Toolkit to assemble Web services-enabled WAR files. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled WAR file when starting from WSDL:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the WAR file into the Assembly Toolkit.
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 WAR file you just imported in the **Project Navigator** pane.
6. Expand the `WebContent` directory so 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. To confirm:
   a. Double-click **Web Deployment Descriptor**.
   b. In the **Web Deployment Descriptor** editor click the **Servlets** tab.
   c. Enter the full path name of the Java bean class implementing the Web service 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 on the `wsdl` directory and click **File** > **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 `webservices.xml`.
11. Copy the `webservices.xml`, `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi` deployment descriptors in the `WEB-INF` subdirectory.

The artifacts required to Web service-enable the Web module is added to the WAR file.

Assemble a Web services-enabled WAR into an EAR file.

Assembling a Web services-enabled EJB JAR into an EAR file

Before assembling a Web services-enabled enterprise archive (EAR) file Assemble a Web services-enabled EJB Java archive (JAR) file.

You can assemble a Web services-enabled EAR file with the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled EAR file:
1. Start the Assembly Toolkit.
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

Before assembling a Web services-enabled enterprise archive (EAR) fileAssemble a Web services-enabled Web archive (WAR) file.

This topic explains how to assemble a Web services-enabled WAR file into and EAR file using the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Tool (AAT) and is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled WAR file into an EAR file:

1. Start the Assembly Toolkit.
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 based on Web Services for J2EE.

Enabling a Web services-enabled EAR file

Before doing this task, you need to Assemble a Web services-enabled EJB JAR into an enterprise archive (EAR) file.

You can add router modules to your Web services-enabled application, also known as an EAR file with the **endptEnabler** command or the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Toolkit (AAT) and is a component of the Application Server Toolkit (ASTK) product.

These tools add one or more router modules to the EAR file for each EJB JAR module within the EAR file. A router module provides an endpoint for the Web services in a particular enterprise JavaBean (EJB) Java archive (JAR) module.

Each router module supports a specific transport such as HyperText Transport Protocol (HTTP) or Java Messaging Service (JMS). If there are no EJB JAR modules in the EAR file, it is not necessary to use these tools.

1. Enable an EAR file with the **endptEnabler** command-line tool.
2. Enable an EAR file with the Assembly Toolkit.

Deploy the EAR file into WebSphere Application Server.

Enabling a Web services-enabled EAR file with the endptEnabler command

Before doing this task, you need to assemble a Web services-enabled EJB JAR into an enterprise archive (EAR) file.

The **endptEnabler** command-line tool adds one or more router modules to the EAR file for each EJB JAR module within the EAR file. A router module provides an endpoint for the Web services in a particular Enterprise JavaBean (EJB) Java archive (JAR) module.

Each router module supports a specific transport such as HyperText Transport Protocol (HTTP) or Java Messaging Service (JMS). If there are no EJB JAR modules in the EAR file, it is not necessary to use these tools.

To enable an EAR file with the **endptEnabler** command:

1. Invoke the **endptEnabler** command from the `*install_root*\bin` directory. If you are using UNIX, 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 they are requested by the **endptEnabler** command. You are prompted for various input values for each Web services-enabled EJB JAR module in the EAR file. Typically, you should 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 Web services-enabled EJB JAR module contained in the EAR file. For HTTP, a context-root is configured for the application so the Web service can be invoked through a URL. The URL used to invoke the Web service is:

```
http://host[:port]/context-root/services/port-component-name
```

Deploy the EAR file into WebSphere Application Server.

endptEnabler command: The **endptEnabler** command enables a set of Web services within an enterprise archive (EAR) file. You can add one or more router modules to the EAR file that include a Web service-enabled EJB JAR file.

Each router module provides a Web service endpoint for a particular transport. For example, an HyperText Transport Protocol (HTTP) router module can be added so that the Web service can receive requests over the HTTP transport, and a Java Messaging Service (JMS) router module can be added 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 a Web service-enabled EJB JAR file, 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 as follows:

```
endptEnabler
  [-verbose|-v]
  [-quiet|-q]
  [-help|-h|-?]
  [-properties|-p properties-filename]
  [-transport|-t default-transports]
  [-enableHttpRouterSecurity]
  [ear-filename]
```

All parameters are optional and described as follows:

- **-verbose, -v**

  Detailed progress messages are displayed as the tool processes the EAR file. This command-line option is mapped to the verbose global property.
- **-quiet, -q**

  No per-module progress messages are displayed as the tool processes the EAR file. This command-line option is mapped to the quiet global property.
- **-help, -h, -?**

  A brief help message is displayed explaining the various options.
- **-properties, -p *properties-filename***

  Properties from the file <properties-filename> are read and used to control the behavior of the tool.
- **-transport, -t *default-transports***

  Specifies the default list of transports for which router modules should be created for each EJB JAR file contained in the EAR file. This command-line option is mapped to the defaultTransports global property. Examples are:

  ```
  -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 EJB's are secured in the EJB 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 interactive mode, you are prompted for the EAR file name, router module names and other important values as the processing occurs. The following dialog is an example of the endptEnabler interactive mode:

  Note: In this dialog, user input is in fixed width font, and 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 non-interactive mode, router module names and other important values are determined from user-specified properties or default values.

endptEnabler properties

The **endptEnabler** command allows you to control its run time behavior by specifying a set of properties with the -properties command-line option. These properties fall into two categories: global and per-module. Global properties affect the overall behavior of the tool as it processes multiple EJB JAR modules within the EAR file. Per-module properties affect the processing of a particular EJB JAR module.

Global properties

The following table describes the global properties supported by the **endptEnabler** command:

Property name	Description	Default value
verbose	Displays detailed progress messages.	False
quiet	Displays only brief progress messages.	False
http.enableRouter Security	Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB's are secured in the EJB JAR file.	False
http.router ModuleNameSuffix	Specifies the suffix used to construct default HTTP router module names. The `.war` extension is added by the **endptEnabler** command.	_HTTPRouter
jms.routerModule NameSuffix	Specifies the suffix used to construct default JMS router module names. The `.jar` extension is added by the **endptEnabler** command.	_JMSRouter
jms.listenerInput PortNameSuffix	Specifies the suffix used to construct default Listener Input Port names.	_ListenerPort
jms.default DestinationType	Specifies the default destination type to use for all JMS router modules added to the EAR file. This should be either queue or topic.	queue
defaultTransports	Specifies the default list of transports for which router modules should be 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. *`ejbJarName`* refers to the name of an EJB JAR module within the EAR file, without the `.jar` extension.

Property name	Description	Default value
ejbJarName .transports	Lists the transports for which router modules should be created for a particular EJB 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 would otherwise be 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 EJB JAR file.	*ejbJarName*_HTTPRouter
ejbJarName .http.contextRoot	Specifies the context root associated with the HTTP router module for a particular EJB JAR file.	/*ejbJarName*
ejbJarName.jms.skip	Specifies the Flag which bypasses the addition of an HTTP router module even if it would otherwise be 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 EJB JAR file.	*ejbJarName*_JMSRouter
ejbJarName. jms.listenerInputPort Name	Specifies the name of the Listener Input Port to be associated with the JMS router module.	*ejbJarName*_ListenerPort
ejbJarName.ejb JarName.jms. destinationType	Specifies the JMS destination type associated with the JMS router. Valid values are queue and topic.	queue

Properties example

Suppose an EAR file contains an EJB JAR file named, `StockQuoteEJB.jar` that contains Web services. The following set of properties might be used to control the **endptEnabler** command runtime 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.listenerInputPortName=StockQuote_LP

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
```

Enabling a Web services-enabled EAR file with the Assembly Toolkit

Before doing this task, you need to Assemble a Web services-enabled EJB JAR into an enterprise archive (EAR) file.

You can add one or more router modules to your Web services-enabled application, also known as an EAR file with the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Toolkit (AAT) and is a component of the Application Server Toolkit (ASTK) product.

A router module provides an endpoint for the Web services in a particular Enterprise JavaBean (EJB) Java archive (JAR) module.

Each router module supports a specific transport such as HyperText Transport Protocol (HTTP) or Java Messaging Service (JMS). If there are no EJB JAR modules in the EAR file, it is not necessary to use these tools.

To enable a Web services-enabled EAR file with the Assembly Toolkit:
1. Start the Assembly Toolkit.
2. Right-click on the EJB project to be enabled.
3. Click **Web Services** > **Endpoint Enabler**.
4. Specify the transport and router module names in the corresponding fields.
5. Click **OK**.

An HTTP or JMS router module is added to the EAR file for each Web services-enabled EJB JAR module contained in the EAR file. For HTTP, a context-root is configured for the application so the Web service can be invoked through a URL. The URL used to invoke the Web service is:

```
http://host[:port]/context-root/services/port-component-name
```

Deploy the EAR file into WebSphere Application Server.

Deploying Web services based on Web Services for J2EE

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 has been configured and enabled for Web services. You can use either the administrative console or the **wsadmin** scripting interface 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** during step 1 of the Install New Application wizard. For more information about installing applications using the administrative console see Installing a new application.

Note:

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

Use the following steps to deploy the EAR file with the **wsadmin** command:

1. Start *install_root*\bin\wsadmin from a command prompt. If you are using UNIX start *install_root*/bin/wsadmin.
2. Enter the **$AdminApp install *EARfile* ″-usedefaultbindings -deployws″** command at the **wsadmin** prompt.

The Web service is installed into the application server.

Secure Web services.

wsdeploy command

This topic explains how to use the **wsdeploy** command-line tool with Web services that are based on the Web Services for 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 than once. Deployment can be performed separately using the **wsdeploy** command, the Assembly Toolkit, or when the application is installed. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product. When using the **wsadmin** command for installation, specify the -deployws option. When using the administrative console for installation, select the **Deploy Web services** check box. When using the Assembly Toolkit, **Right-click** the module and select **Web Services** >**Deploy Web Services** from the pop-up menu. You can download the Assembly Toolkit from the Web site

```
http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=
ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en
```

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

wsdeploy command syntax

The command syntax is as follows:

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

Required options:
- ***Input_filename***

  Specifies the path to the EAR or JAR file to be deployed.
- ***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 containing 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 be added to CLASSPATH when the generated classes are compiled. Multiple entries are separated the same as they would be in the CLASSPATH environment variable, with a semicolon on Windows platforms and a colon for UNIX platforms.
- **-codegen**

  Specifies that deployment code is to be generated, but not compiled. 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

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

Messages

- Flag *-f* is not valid

  Option *f* was not recognized as being a valid option.
- Flag *-c* is ambiguous

  Options may be abbreviated, but the abbreviation must be unique. In this case, the **wsdeploy** command can not determine which option was intended.
- Flag *-c* is missing parameter *-p*

  A required parameter for an option was omitted.
- Missing *p* parameter

  A required option was omitted.

Using the Java Messaging Service to transport Web services requests

WebSphere Application Server offers support for using a Java Messaging Service (JMS) transport layer, in addition to the existing HTTP transport. Using JMS transport allows your Web service clients and servers to communicate through JMS queues and topics instead of HTTP connections. One-way and synchronous two-way requests are supported.

Note: A Web service must be implemented as an enterprise JavaBean (EJB) to be accessed through the JMS transport.

The benefits of using JMS as an alternative to HTTP, include:
- Request and response messages are sent through reliable messaging.
- One-way requests allow clients and servers to be more loosely-coupled. For example, the server does not have to be active when the client sends the one-way request.
- One-way requests can be sent to multiple servers simultaneously through the use of a topic.

To use JMS as a transport for Web services requests:
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, in order to be accessible on the JMS transport. A JMS binding is a `wsdl:binding` element containing 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 should contain a `wsdlsoap:address` element whose `location` attribute specifies a JMS endpoint URL string.

   Note: The specification of the actual JMS endpoint URL string can be deferred until you publish the WSDL file. When you develop the Web service, a placeholder such as `file:/unspecified_location` can be used for the endpoint URL string.
2. Decide on the names and types of JMS objects that your application uses Before your application can be installed, you need to:
   a. Decide whether your Web service receives its requests from a queue or a topic.
   b. Decide whether to use a secure destination (queue or topic) or a nonsecure destination.
   c. Decide on the names for your destination, connection factory and listener port. The following list provides examples of the names that might be used for the mythical StockQuote Web service:
      - **Queue:** StockQuote_Q (JNDI name: jms/StockQuote_Q)
      - **Connection factory:** StockQuote_CF (JNDI name: jms/StockQuote_CF)
      - **Listener port:** StockQuoteEJB_ListenerPort
3. Define the JMS administered objects. Once you have decided on the names and types of the JMS objects, use the administrative console or the **wsadmin** scripting interface to define the JMS objects.
4. Add the JMS endpoints to your EAR file using the **endptEnabler** command tool. You must run the **endptEnabler** command to add a JMS endpoint or router module for each Web service-enabled EJB JAR file contained in the EAR file. By default, the **endptEnabler** command adds only HTTP endpoints, but the -transport jms option can be used to request the addition of JMS endpoints.
5. Deploy the Web services application. During the install process you are prompted for two types of information for each Web service-enabled EJB JAR contained in your EAR file:
   - The Java Naming and Directory Interface (JNDI) name of the connection factory to be used by the EJB JAR file message driven bean (MDB) listener for sending reply messages.

     If your Web service contains two-way operations, the MDB listener, defined inside the JMS endpoint added by **endptEnabler** command, needs to access a queue connection factory in order 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 install process, you must provide the actual JNDI name of the queue connection factory that should be used by the MDB listener 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 to be used by the MDB listener.

  A listener port is an object 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 desired queue or topic are properly delivered to the MDB. During deployment, you can modify the name of the listener port associated with each MDB listener. The listener port name contained in the input EAR file displays as a default value. If you specify the correct listener port name to the **endptEnabler** command, perhaps through the use of properties, during step 3, 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.

6. Publish the WSDL file. In this step, you enter the JMS endpoint URL string to use for each Web service-enabled EJB JAR file belonging to the application. The JMS endpoint URLs are then written to the published WSDL files for use by clients.

   For example, suppose that an application called StockQuoteService contains an EJB JAR file named `StockQuoteEJB`, which contains one or more Web services accessible on the JMS transport. Suppose that, in step 2, you defined a queue with the JNDI name `jms/StockQuote_Q` and a connection factory with the JNDI name `jms/StockQuote_CF` to be used by your application. In this example, you would specify the following string as the JMS URL prefix within the **Publish WSDL** user interface:

   ```
   jms:/queue?destination=jms/StockQuote_Q&connectionFactory=jms/StockQuote_CF
   ```

   The WSDL publisher uses this partial URL string to produce the actual JMS URL for each port component defined in the EJB JAR file. The published WSDL file can be used by clients needing to invoke the Web service.

Java Messaging Service endpoint URL syntax

A Java Messaging Service (JMS) endpoint URL is used to access a Web service 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 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 transport type, `jms:`, 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 used to specify the JMS endpoint information.

The properties supported in the URL string are described as follows:

Destination-related properties (required)

Property name	Description
destination	Specifies the Java Naming and Directory Interface (JNDI) name of the destination queue or topic.
connectionFactory	Specifies the JNDI name of the connection factory.
targetService	Specifies the name of the port component to which the request is dispatched.

JNDI-related properties (optional)

Property name	Description
initialContextFactory	Specifies the name of the initial context factory to use which is mapped to the java.naming.factory.initial property.
jndiProviderURL	Specifies the JNDI provider URL which is mapped to the java.naming.provider.url property.

JMS-related properties (optional)

Property name	Description
deliveryMode	Indicates whether the request message should be 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`.

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. This means that 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 specified as part of a JMS endpoint URL string.

Securing Web services based on WS-Security

Web services security for WebSphere Application Server is based on standards included in the Web services security (WS-Security) specification. These standards address how to provide protection for messages exchanged in a Web service environment. The specification defines the core facilities for protecting the integrity and confidentiality of a message and provides mechanisms for associating security-related claims with the message. Web services security is a message-level standard based on securing SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens.

Use the deprecated ″Securing Apache SOAP Web services″ topics in the WebSphere Application Server, Version 5 documentation if you are still using Apache SOAP Version 2.3.

To secure Web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, federation, delegation, and auditing across a spectrum of application and business topologies. One of the key requirements for the security model in today’s business environment is the ability to interoperate between formerly incompatible security technologies (such as public key infrastructure, Kerberos and so on.) in heterogeneous environments (such as Microsoft .NET and Java 2 Platform, Enterprise Edition (J2EE)). The complete Web services security protocol stack and technology roadmap is described in Security in a Web Services World: A Proposed Architecture and Roadmap.

Specification: Web Services Security (WS-Security) proposes a standard set of SOAP extensions that you can use to build secure Web services. These standards confirm integrity and confidentiality, which are generally provided with digital signature and encryption technologies. In addition, Web services security provides a general purpose mechanism for associating security tokens with messages. A typical example of the security token is a user name and password token, in which a user name and password are included as text. Web services security defines how to encode binary security tokens using methods such as X.509 certificates and Kerberos tickets.

An administrator can use any of the following methods to integrate message-level security into a WebSphere Application Server environment:

- Secure Web services using XML digital signature
- Secure Web services using XML encryption
- Secure Web services using basicauth authentication
- Secure Web services using identity assertion authentication
- Secure Web services using signature authentication
- Secure Web services using a pluggable token

Web services security specification- a chronology

This document describes the process used to develop the Web services security specifications.

Non-OASIS activities

In April 2002, IBM, Microsoft, and VeriSign proposed the *Web Services Security (WS-Security) specification* on their Web sites. This specification included the basic ideas of security token, XML signature, and XML encryption. The specification also defined the format for username tokens and encoded binary security tokens. After some discussion and an interoperability test based on the specification, the following issues were noted:

- The specification requires that the Web services security processors understand the schema correctly so that the processor distinguishes between the ID attribute for XML signature and XML encryption.
- The freshness of the message, which indicates whether the message complies with predefined time constraints, cannot be determined.
- Digested password strings do not strengthen security.

In August 2002, IBM, Microsoft, and VeriSign published the *Web Services Security Addendum*, which attempted to address the previously listed issues. The following solutions were put in the addendum:

- Require a global ID attribute for XML signature and XML encryption
- Use time stamp header elements that indicate the time of the creation, receipt, or expiration of the message
- Use password strings that are digested with a time stamp and nonce (randomly generated token)

OASIS activities

In June 2002, the Organization for the Advancement of Structured Information Standards (OASIS) received a proposed Web services security specification from IBM, Microsoft, and Verisign. The Web Services Security Technical Committee (WSS TC) was organized at OASIS soon after the submission. The technical committee included many companies including IBM, Microsoft, VeriSign, Sun Microsystems, and BEA Systems.

In September 2002, WSS TC published its first specification, *Web Services Security Core Specification, Working Draft 01*. This specification included the contents of both the original Web services security specification and its addendum.

The coverage of the technical committee became larger as the discussion proceeded. Since the Web Services Security Core Specification allows arbitrary types of security tokens, proposals were published as profiles. The profiles described the method for embedding tokens, including Security Assertion Markup

Language (SAML) tokens and Kerberos tokens imbedded into the Web services security messages. Subsequently, the definitions of the usage for user name tokens and X.509 binary security tokens, which were defined in the original Web Services Security Specification, were divided into the profiles.

WebSphere Application Server supports the following specifications:

- Web Services Security: SOAP Message Security Draft 13 (formerly Web Services Security Core Specification)
- Web Services Security: Username Token Profile Draft 2

The following figure shows the various Web services security-related specifications. As indicated in the figure, the current support level for Web services security: SOAP message security is based on Draft 13 from May 2003. The current support level for Web services security User name token profiles, is based on Draft 2 from February 2003.

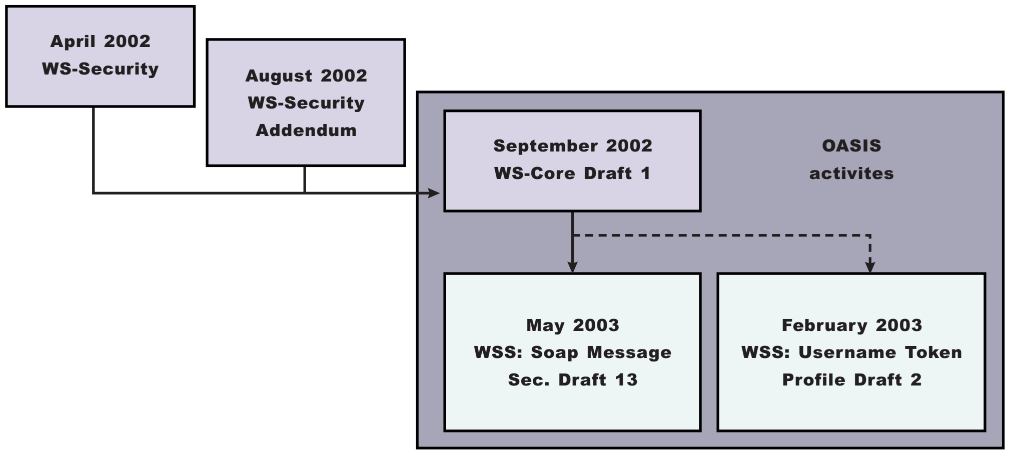


Figure 4. Web services security specification support

Web services security support

WebSphere Application Server, Versions 4.x, 5, and 5.0.1 support digital signature for Apache Simple Object Access Protocol (SOAP) Version 2.x. Beginning with WebSphere Application Server, Version 5.0.2, IBM supports *Web services security*, which is an extension of the IBM Web services engine to provide a quality of service. The IBM implementation is based on the Web services security specification, ″Web Services Security (WS-Security)″, originally proposed by IBM, Microsoft, and VeriSign in April 2002. Early versions of the proposed draft specification can be found in Web Services Security (WS-Security) Version 1.0 05 April 2002 and Web Services Security Addendum 18 August 2002. The WebSphere Application Server implementation is based on the Organization for the Advancement of Structured Information Standards (OASIS) working Draft 13 specification. (See the OASIS Web Services Security TC Web site for the latest working specification.) However, not all the features in the OASIS working Draft 13 specification are implemented.

WebSphere Application Server security infrastructure fully integrates Web services security with Java 2 Platform, Enterprise Edition (J2EE) security. When a user ID and password are embedded in a request message, authentication is performed with the user ID and password. If authentication is successful, a user identity is established and further resource access is authorized based on that identity. After the user ID and password are authenticated by the Web services security run time, a J2EE container performs authorization.

WebSphere Application Server provides an implementation of the key features of Web services security based on the following specifications:

- Specification: Web Services Security (WS-Security) Version 1.0 05 April 2002
- Web Services Security Addendum 18 August 2002
- Web Services Security: SOAP Message Security Working 13 May 2003
- Web Services Security: Username Token Profile Draft

The following table provides a summary of Web services security elements supported by WebSphere Application Server:

Table 2. Web services security elements

Element	Notes
`UsernameToken`	Both the user name and password for the `BasicAuth` authentication method and the user name for the identity assertion authentication method are supported. WebSphere Application Server, Version 5.1 supports nonce, a randomly generated value.
`BinarySecurityToken`	X.509 certificates and Lightweight Third Party Authentication (LTPA) can be embedded, but there is no implementation to embed Kerberos tickets. However, the binary token generation and validation are pluggable and are based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). You can extend this implementation to generate and validate other types of binary security tokens.
`Signature`	The X.509 certificate is embedded as a binary security token and can be referenced by the `SecurityTokenReference`. WebSphere Application Server does not support shared, key-based signature.
`Encryption`	Both the `EncryptedKey` and `ReferenceList` XML tags are supported. `KeyIdentifier` specifies public keys and `KeyName` identifies the secret keys. WebSphere Application Server has the capability to map an authenticated identity to a key for encryption or use the signer certificate to encrypt the response message.
`Timestamp`	WebSphere Application Server supports the Created and Expires attributes. The freshness of the message, which indicates whether the message complies with predefined time constraints, is checked only if the Expires attribute is present in the message. WebSphere Application Server does not support the Received attribute, which is defined in the addendum. Instead, WebSphere Application Server uses the TimestampTrace Received attribute, which is defined in the OASIS specification.
`XML based token`	You can insert and validate an arbitrary format of XML tokens into a message. This format mechanism is based on the JAAS APIs.

Signing and encrypting attachments is not supported by WebSphere Application Server. The namespaces used for sending a message were published by OASIS in draft 13 (http://schemas.xmlsoap.org/ws/2003/06/secext). However, the Web services security run time in WebSphere Application Server can accept any of the following namespaces:

April 2002 specification
http://schemas.xmlsoap.org/ws/2002/04/secext

August 2002 addendum
http://schemas.xmlsoap.org/ws/2002/07/secext

http://schemas.xmlsoap.org/ws/2002/07/utility

OASIS draft published on draft 13 May 2003
http://schemas.xmlsoap.org/ws/2003/06/secext

http://schemas.xmlsoap.org/ws/2003/06/utility

Note: 5.1 + WebSphere Application Server only uses the previously mentioned two name spaces for sending out requests and responses. However, the product can process all other mentioned name spaces for incoming requests and responses.

WebSphere Application Server provides the following capabilities for Web services security:
- Integrity of the message
- Authenticity of the message
- Confidentiality of the message
- Privacy of the message
- Transport level security: provided by Secure Sockets Layer (SSL)
- Security token propagation (pluggable)
- Identity assertion

See the previous table titled, ″Web services security elements,″ for a description of capabilities that are not supported.

Web services security and Java 2 Platform, Enterprise Edition security relationship

This document describes the relationship between Web services security (message level security) and Java 2 Platform, Enterprise Edition (J2EE) platform security.

WebSphere Application Server supports Java Specification Requests (JSR) 101 and JSR 109 (see Developing Web services for more information). These JSRs define Web services for the Java 2 Platform, Enterprise Edition (J2EE) architecture so that you can develop and run Web services on the J2EE component architecture. ″Web services security″ refers to the ″Web services security: SOAP Message Security″ specification (see Web services security support for more information).

Important

Note: ″Web services security″ refers to the ″Web services security: SOAP Message Security″ specification (see Web services security support for more information).

Securing Web services with WebSphere Application Server security (J2EE role-based security)

You can secure Web services using the existing security infrastructure of WebSphere Application Server, J2EE role-based security, and Secure Sockets Layer (SSL) transport level security.

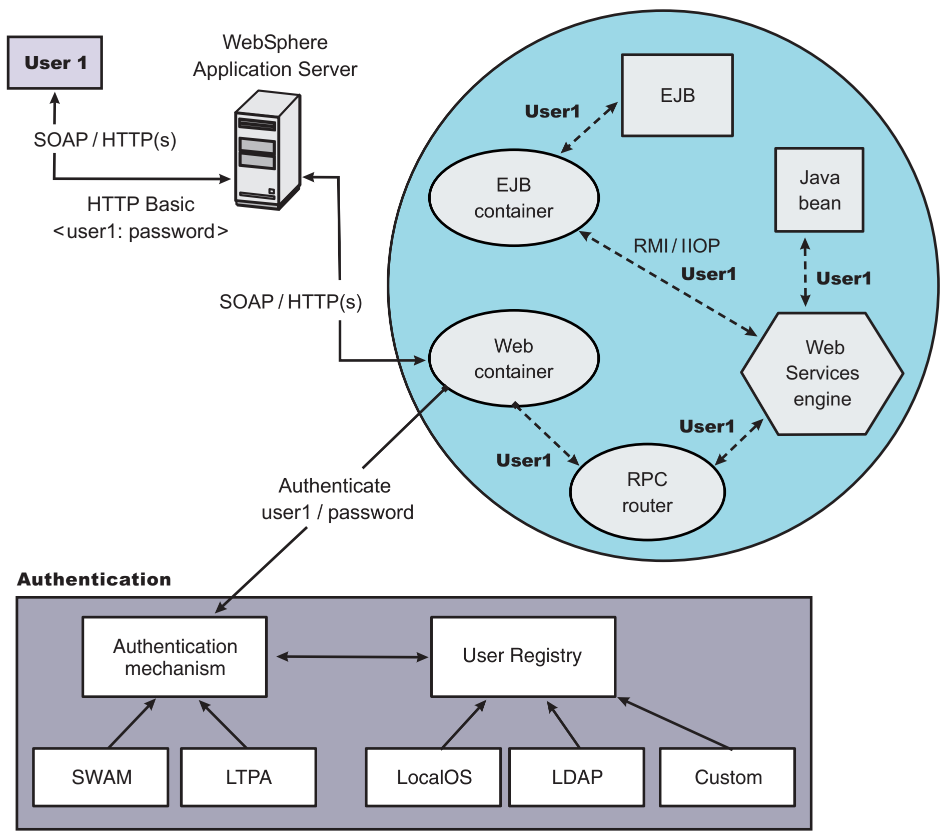


Figure 5. Simple object access protocol message flow using existing security infrastructure of WebSphere Application Server

The Web services endpoint can be secured using J2EE role-based security. The Web services sender sends the basic authentication data using the HTTP header. SSL (HTTPS) can be used to secure the transport. When the WebSphere Application Server receives the SOAP message, the Web container authenticates the user (in this example, `user1`) and sets the security context for the call. After the security context is set, the SOAP router servlet sends the request to the implementation of the Web services (the implementation can be JavaBeans or enterprise bean files). For enterprise bean implementations, the EJB container performs an authorization check against the identity of `user1`.

The Web services endpoint also can be secured using the J2EE role. Then, authorization is performed by the Web container before the SOAP request is dispatched to the Web services implementation.

Securing Web services with Web services security at the message level

You can also secure Web services using Web services security at the message level. In this case, you can digitally sign or encrypt a certain part of the message. Web services security also supports security token propagation within the SOAP message. The following scenario assumes that the Web services endpoint is not secured with J2EE role-based security and the enterprise bean is secured with J2EE role-based security.

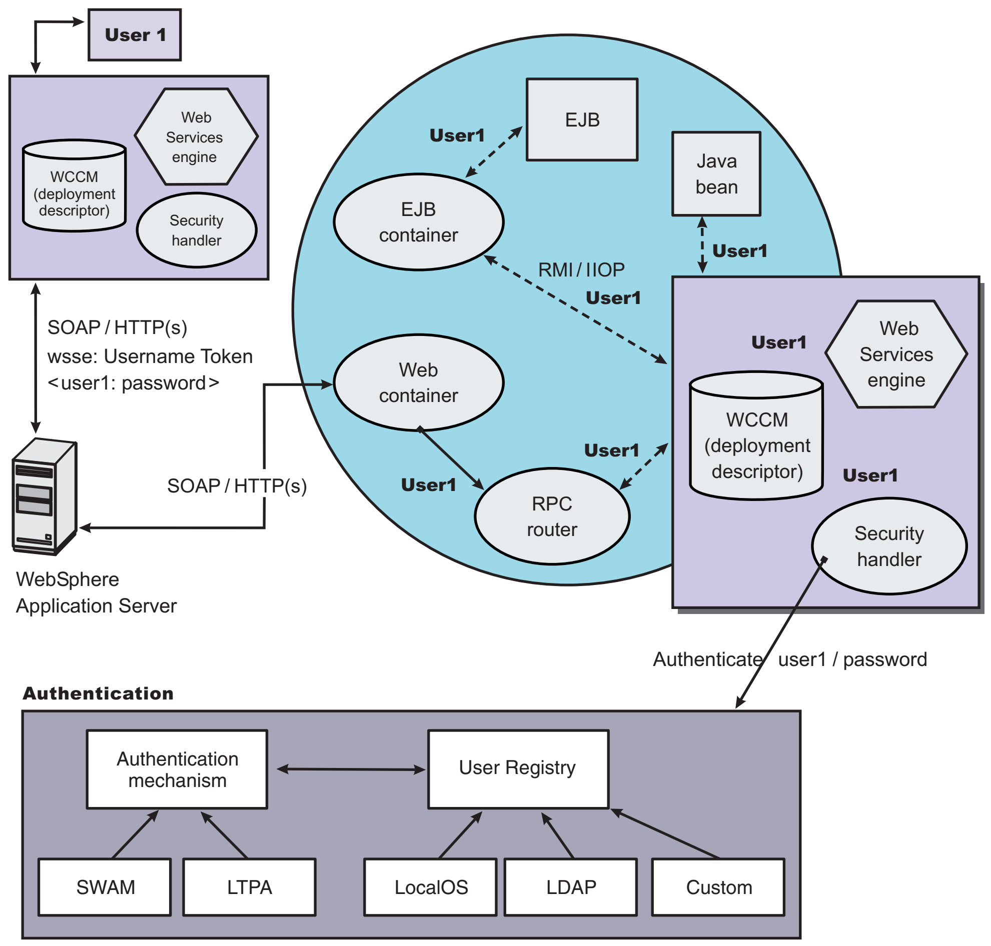


Figure 6. Simple Object Access Protocol message flow using Web services security

In this case, the Web services endpoint is not secured with J2EE role-based security. The Web services engine processes the SOAP message before the client sends the message to the Web services endpoint. The Web services security run time acts on the security constraints, such as digitally signing, encrypting, or generating (and inserting) a security token in the SOAP header. In this case `<wsse:UsernameToken>` is generated using `user1` and the password. On the server-side (receiving), the Web services process the incoming message and Web services security enforces security constraints. This enforcement includes making sure that messages are properly signed, properly encrypted, and decrypted, authenticating the security token, and setting up the security context with the authenticated identity. (In this case, `user1` is the authenticated identity.) Finally, the SOAP message is dispatched to the Web services implementation (if the implementation is an enterprise beans file, the EJB container performs an authorization check against `user1`). SSL also might be used in this scenario.

Mixing the two

The second scenario shows that Web services security can complement J2EE role-based security. For example, SSL can be enabled at the transport level to provide a secure channel. Furthermore, if the Web services implementation is an enterprise beans file, you can leverage the EJB role-based authorization by performing authorization checks. Web services security run time leverages the security infrastructure to set the authenticated identity in the security context. The authenticated identity can be used in the downstream call to J2EE resources (or other resource types).

There are subtle consequences of combining the two scenarios. For example, if the HTTP transport is sending basic authentication data with `user1` and password in the HTTP header, but `<wsse:UsernameToken>` with `user99` and `letmein` also is inserted into the SOAP header. In the previous scenarios, there are two authentications performed. One authentication is performed by the Web container for authenticating `user1`, and the other is performed by Web services security for authenticating `user99`. The Web services security run time runs after the Web container runs and `user99` is the authenticated identity that is set in the security context.

Web services security can also propagate security tokens from the sender to the receiver for SOAP over a Java Message Service (JMS) transport.

Web services security model in WebSphere Application Server

The Web services security model used by WebSphere Application Server is the declarative model. WebSphere Application Server does not include any application programming interfaces (APIs) for programmatically interacting with Web services security. However, a few Server Provider Interfaces (SPIs) are available for extending some security-related behaviors.

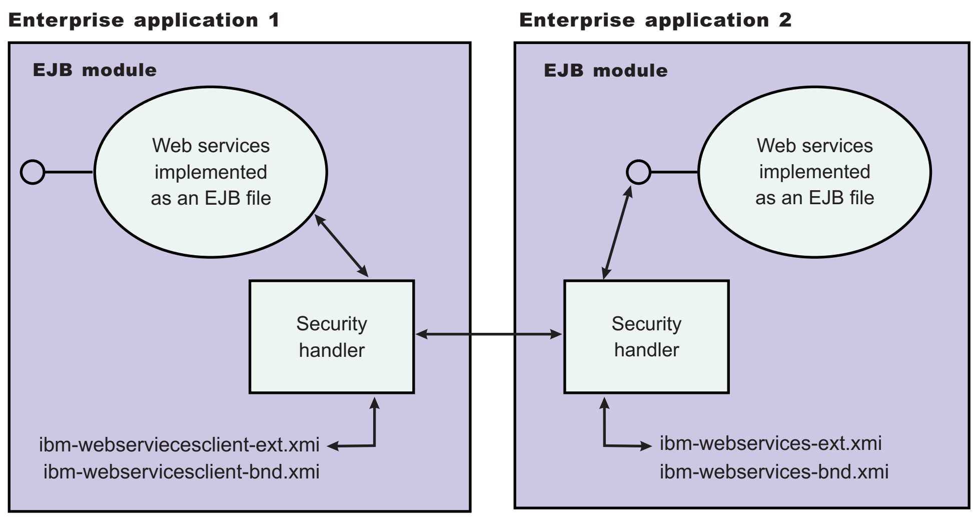


Figure 7. Web services security model

The security constraints for Web services security are specified in IBM deployment descriptor extensions for Web services. The Web services security run time acts on the constraints to enforce Web services security for the Simple Object Access Protocol (SOAP) message. The scope of the IBM deployment descriptor extension is at the enterprise bean (EJB) or Web module level. Bindings are associated with each of the following IBM deployment descriptor extensions:

Client (Might be either a J2EE Client (Application Client Container) or Web services acting as a client)

`ibm-webservicesclient-ext.xmi`

`ibm-webservicesclient-bnd.xmi`

Server

`ibm-webservices-ext.xmi`

`ibm-webservices-bnd.xmi`

It is recommended that you use the tools provided by IBM (the Assembly Toolkit and WebSphere Studio Application Developer) to create the IBM deployment descriptor extension and bindings. After the bindings are created, you can use the administrative console, the Assembly Toolkit, or the WebSphere Studio Application Developer to specify the bindings.

Important

Note: The binding information is collected after application deployment rather than during application deployment. The alternative is to specify the required binding information before deploying your application.

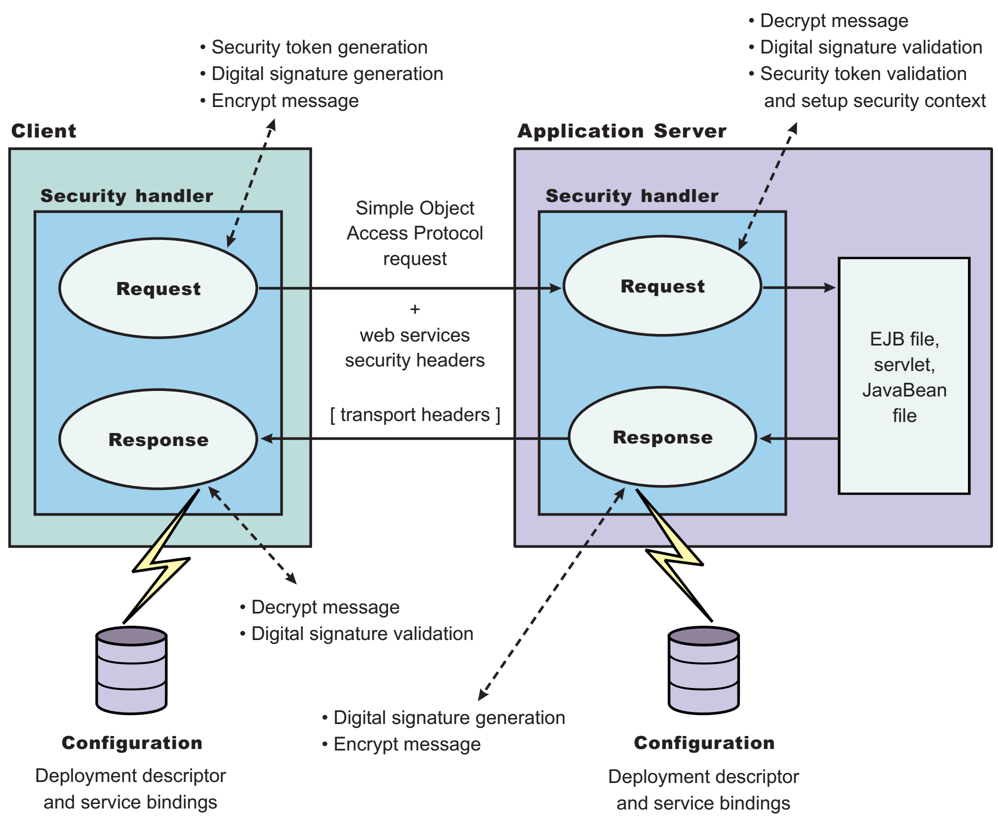

• Security token generation
• Digital signature generation
• Encrypt message
• Decrypt message
• Digital signature validation
• Security token validation and setup security context
Client
Application Server
Security handler
Security handler
Simple Object Access Protocol request
+
web services security headers
[transport headers]
Request
Response
Request
Response
EJB file, servlet, JavaBean file
• Decrypt message
• Digital signature validation
• Digital signature generation
• Encrypt message
Configuration
Deployment descriptor and service bindings
Configuration
Deployment descriptor and service bindings

Figure 8. Web services security message interpretation

The Web services security run time enforces Web services security based on the defined security constraints in the deployment descriptor and binding files. Web services security has the following four points where it intercepts the message and acts on the security constraints defined:

Message points	Description
Request sender (defined in the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files)	• Applies the appropriate security constraints to the SOAP message (such as signing or encryption) before the message is sent, generating the time stamp or the required security token.

Message points	Description
Request receiver (defined in the `ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` files)	• Verifies that the Web services security constraints are met. • Verifies the freshness of the message based on the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints. • Verifies the required signature. • Verifies that the message is encrypted and decrypts the message if encrypted. • Validates the security tokens and sets up the security context for the downstream call.
Response sender (defined in the `ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` files)	• Applies the appropriate security constraints to the SOAP message response, like signing the message, encrypting the message, or generating the time stamp.
Response receiver (defined in the `ibm-webservicesclient-ext.xmi` or `ibm-webservicesclient-bnd.xmi` files)	• Verifies that the Web services security constraints are met. • Verifies the freshness of the message based on the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints. • Verifies the required signature. • Verifies that the message is encrypted and decrypts the message, if encrypted.

Web services security property collection

Use this page to a view a list of additional properties for the configuration.

There are several ways to view a Web services security property collection panel. Complete the following steps to view one of these administrative console pages:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Key Locators** > *key_locator_name*.
3. Under Additional Properties, click **Properties**.
4. Click **New** to create a new property.
5. Click **Delete** to a delete a property that you specified previously.

Name

Specifies the name of the property.

Value

Specifies the value for the property.

Web services security property configuration settings

Use this page to configure additional properties.

There are several ways to view a Web services security property configuration settings panel. Complete the following steps to view one of these administrative console pages:

1. Click **Servers** > **Application Servers** >*server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Key Locators** > *key_locator_name*.
3. Under Additional Properties, click **Properties** > **New**.

Property Name
Specifies the name of the property.

Data type:	String

Property Value
Specifies the value for the property.

Data type:	String

Usage scenario for propagating security tokens

A sample scenario

This document describes a usage scenario for Web services security.

In scenario 1, Client 1 invokes Web services 1. Then Web services 1 calls EJB file 2. EJB file 2 calls Web services 3 and Web services 3 calls Web services 4.

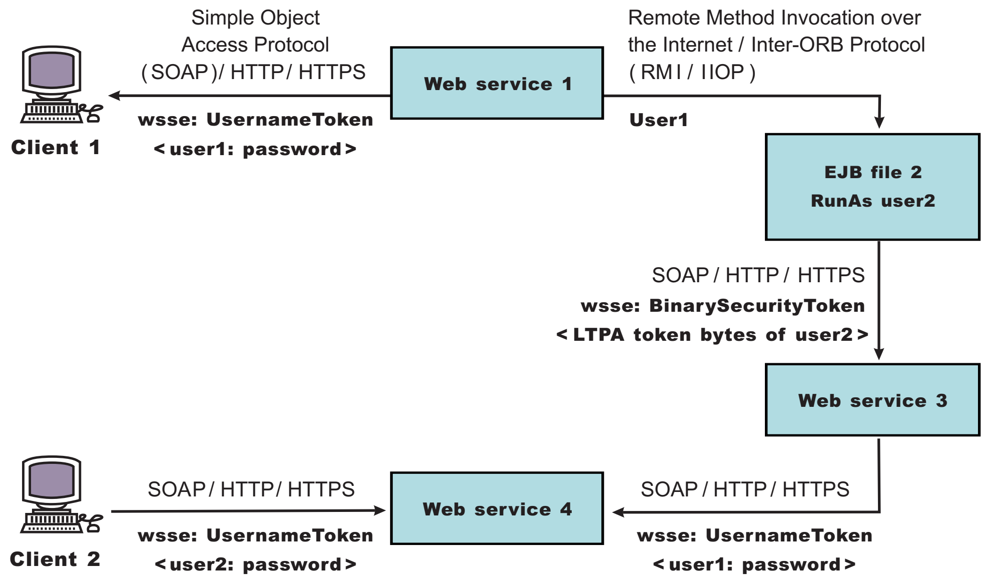


Figure 9. Propagating security tokens

The previous scenario shows how to propagate security tokens using Web services security, the security infrastructure of the WebSphere Application Server, and Java 2 Platform, Enterprise Edition (J2EE) security. Web services 1 is configured to accept `<wsse:UsernameToken>` only and use the BasicAuth authentication method. However, Web services 4 is configured to accept either `<wsse:UsernameToken>` using the BasicAuth authentication method or Lightweight Third Party Authentication (LTPA) as `<wsse:BinarySecurityToken>`. The following steps describe the scenario shown in the previous figure:

1. Client 1 sends a SOAP message to Web services 1 with `user1` and `password` in the `<wsse:UsernameToken>` element.
2. The `user1` and `password` values are authenticated by the Web services security run time and set in the current security context as the Java Authentication and Authorization Service (JAAS) Subject.
3. Web services 1 invokes EJB file 2 using the Remote Method Invocation over the Internet Inter-ORB Protocol (RMI/IIOP) protocol.

4. The `user1` identity is propagated to the downstream call.
5. The EJB container of EJB file 2 performs an authorization check against `user1`.
6. EJB file 2 calls Web services 3 and Web services 3 is configured to accept LTPA tokens.
7. The RunAs role of EJB file 2 is set to `user2`.
8. The LTPA CallbackHandler implementation extracts the LTPA token from the current JAAS Subject in the security context and Web services security run time inserts the token as `<wsse: BinarySecurityToken>` in the SOAP header.
9. The Web services security run time in Web services 3 calls the JAAS login configuration to validate the LTPA token and set it in the current security context as the JAAS Subject.
10. Web services 3 is configured to send LTPA security to Web services 4. In this case, assume that the RunAs role is not configured for Web services 3. The LTPA token of `user2` is propagated to Web services 4.
11. Client 2 uses the `<wsse:UsernameToken>` element to propagate the basic authentication data to Web services 4.

Web services security complements the WebSphere Application Server security run time and the J2EE role-based security. This scenario demonstrates how to propagate security tokens across multiple resources such as Web services and EJB files.

Configurations

The Web services security model used by WebSphere Application Server is the declarative model.

No Application Programming Interfaces (APIs) exist in WebSphere Application Server for programmatically interacting with Web services security. However, Service Provider Programming Interfaces (SPIs) are available for extending some security run-time behaviors. You can secure an application with Web services security by defining security constraints in the IBM extension deployment descriptors and in IBM extension bindings.

The development life cycle of a Web services security-enabled application is similar to the Java 2 Platform, Enterprise Edition (J2EE) model. See the following figure for more details.

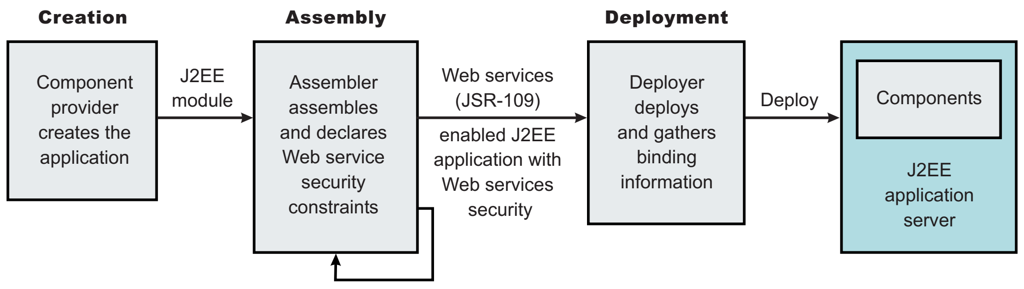

Creation
Component provider creates the application
J2EE module
Assembly
Assembler assembles and declares Web service security constraints
Web services (JSR-109)
enabled J2EE application with Web services security
Deployment
Deployer deploys and gathers binding information
Deploy
Components
J2EE application server

Figure 10. Application development life cycle

The Web services security constraints are defined by the assembler during the application assembly phase if the J2EE application is Web services-enabled. Create, define, and edit the Web services security constraints with the Assembly Toolkit, which can be downloaded from the following location:

http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP
&q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en

Web services security constraints

The security constraints for Web services security are specified in the IBM deployment descriptor extension for Web services. The assembler defines these constraints during the application assembly phase, if the J2EE application is Web services enabled. Define the Web services security constraints using the Assembly Toolkit.

The Web services security run time acts on the constraints to enforce Web services security for the SOAP message. The scope of the IBM deployment descriptor extension is at the EJB module or Web module level. There also are bindings associated with each of the following IBM deployment descriptor extensions:

Client (might be either a J2EE client (application client container) or Web services acting as a client)
- `ibm-webservicesclient-ext.xmi`
- `ibm-webservicesclient-bnd.xmi`

Server
- `ibm-webservices-ext.xmi`
- `ibm-webservices-bnd.xmi`

The IBM extension deployment descriptor and bindings are associated with each EJB module or Web module. See Figure 2 for more information. If Web services is acting as a client, then it contains the client IBM extension deployment descriptors and bindings in the EJB module or Web module.

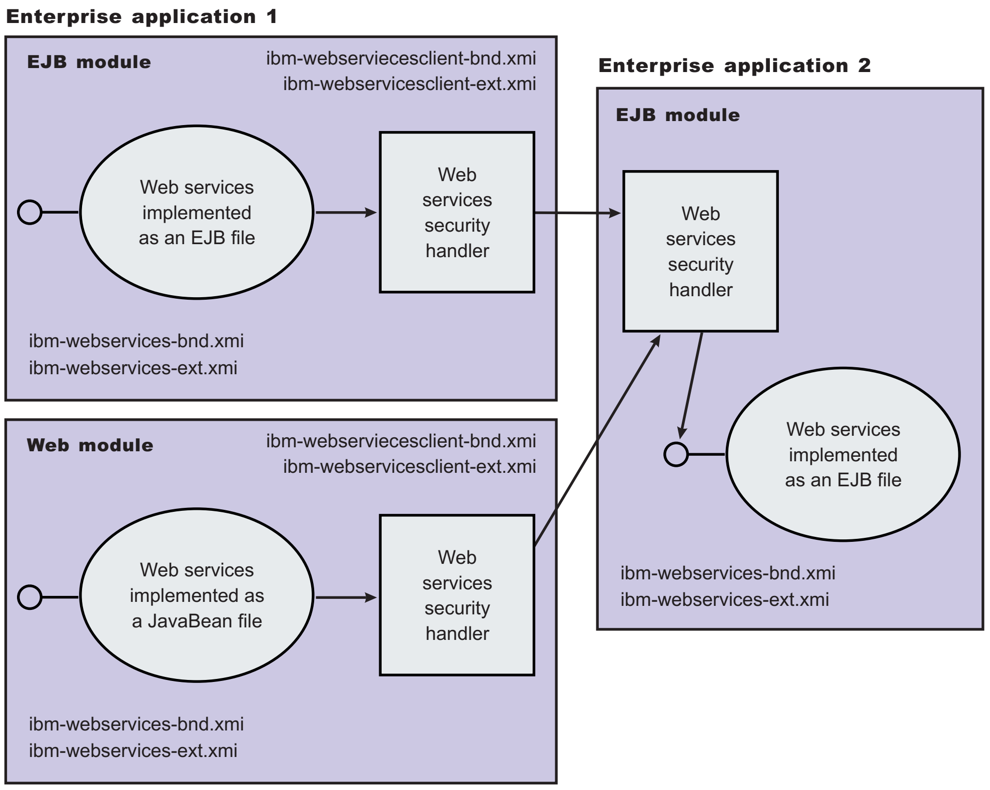

Enterprise application 1
EJB module
ibm-webserviecesclient-bnd.xmi
ibm-webservicesclient-ext.xmi
Web services implemented as an EJB file
Web services security handler
ibm-webservices-bnd.xmi
ibm-webservices-ext.xmi
Web module
ibm-webserviecesclient-bnd.xmi
ibm-webservicesclient-ext.xmi
Web services implemented as a JavaBean file
Web services security handler
ibm-webservices-bnd.xmi
ibm-webservices-ext.xmi
Enterprise application 2
EJB module
Web services security handler
Web services implemented as an EJB file
ibm-webservices-bnd.xmi
ibm-webservices-ext.xmi

Figure 11. IBM extension deployment descriptors and bindings

The Web services security handler acts on the security constraints defined in the IBM extension deployment descriptor and enforces the security constraints accordingly. There are outbound and inbound configurations in both the client and server security constraints.

In a SOAP request, the following message points exist:
- Sender outbound
- Receiver inbound
- Receiver outbound
- Sender inbound

These message points correspond to the following four security constraints:
- Request sender (sender outbound)
- Request receiver (receiver inbound)
- Response sender (receiver outbound)
- Response receiver (sender inbound)

The security constraints of request sender and request receiver must match. Also, the security constraints of the response sender and response receiver must match. For example, if you specify integrity as a constraint in the request receiver, then you must configure the request sender to have integrity applied to the SOAP message. Otherwise, the request is denied because the SOAP message does not include the integrity specified in the request constraint.

The four security constraints are shown in the following figure of Web services security constraints.

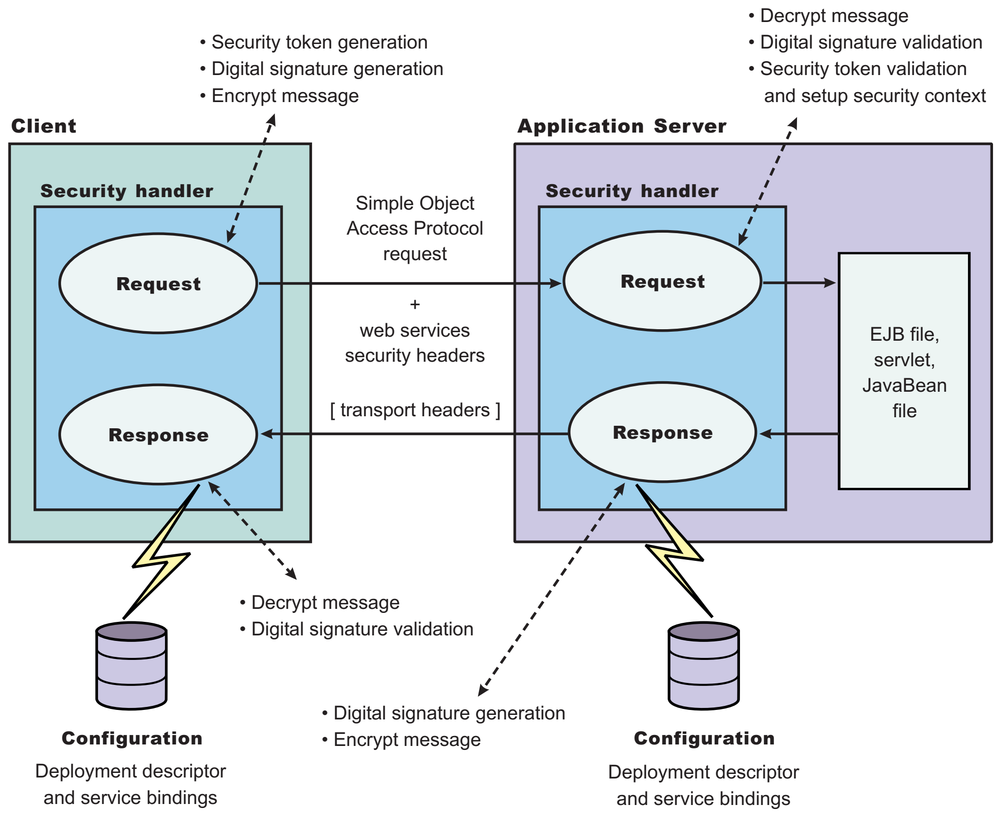

• Security token generation
• Digital signature generation
• Encrypt message
• Decrypt message
• Digital signature validation
• Security token validation and setup security context
Client
Security handler
Request
Response
Simple Object Access Protocol request
+
web services security headers
[transport headers]
Application Server
Security handler
Request
Response
EJB file, servlet, JavaBean file
• Decrypt message
• Digital signature validation
• Digital signature generation
• Encrypt message
Configuration
Deployment descriptor and service bindings
Configuration
Deployment descriptor and service bindings

Figure 12. Web services security constraints

Sample configuration

WebSphere Application Server provides the following sample key stores for sample configurations. These sample key stores are for testing and sample purposes only. Do not use them a in production environment.

- `{USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks`
  - The keystore password is `client`
  - Trusted certificate with alias name, `soapca`
  - Personal certificate with alias name, `soaprequester` and key password `client` issued by intermediary certificate authority `Int CA2`, which is, in turn, issued by `soapca`
- `{USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks`
  - The keystore password is `server`
  - Trusted certificate with alias name, `soapca`
  - Personal certificate with alias name, `soapprovider` and key password `server`, issued by intermediary certificate authority `Int CA2`, which is, in turn, issued by `soapca`
- `{USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks`
  - The keystore password is `storepass`
  - Secret key `CN=Group1`, alias name `Group1`, and key password `keypass`
  - Public key `CN=Bob, O=IBM, C=US`, alias name `bob`, and key password `keypass`
  - Private key `CN=Alice, O=IBM, C=US`, alias name `alice`, and key password `keypass`

- {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
  - The keystore password is storepass
  - Secret key CN=Group1, alias name Group1, and key password keypass
  - Private key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass
  - Public key CN=Alice, O=IBM, C=US, alias name alice, and key password keypass
- {USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer
  - The intermediary certificate authority is Int CA2.

Default binding (cell and server level)

WebSphere Application Server provides the following default binding information:

Trust anchors
Used to validate the trust of the signer certificate.

- SampleClientTrustAnchor is used by the response receiver to validate the signer certificate.
- SampleServerTrustAnchor is used by the request receiver to validate the signer certificate.

Collection Certificate Store
Used to validate the certificate path.

- SampleCollectionCertStore is used by the response receiver and the request receiver to validate the signer certificate path.

Key Locators
Used to locate the key for signature, encryption, and decryption.

- SampleClientSignerKey is used by the requesting sender to sign the SOAP message. The signing key name is clientsignerkey, which can be referenced in the signing information as the signing key name.
- SampleServerSignerKey is used by the responding sender to sign the SOAP message. The signing key name is serversignerkey, which can be referenced in the signing information as the signing key name.
- SampleSenderEncryptionKeyLocator is used by the sender to encrypt the SOAP message. It is configured to use the {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks keystore and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator keystore key locator.
- SampleReceiverEncryptionKeyLocator is used by the receiver to decrypt the encrypted SOAP message. The implementation is configured to use the {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks keystore and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator keystore key locator. The implementation is configured for symmetric encryption (DES or TRIPLEDES). However, to use it for asymmetric encryption (RSA), you must add the private key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass.
- SampleResponseSenderEncryptionKeyLocator is used by the response sender to encrypt the SOAP response message. It is configured to use the {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks keystore and the com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator key locator. This key locator maps an authenticated identity (of the current thread) to a public key for encryption. By default, WebSphere Application Server is configured to map to public key alice, and you must change WebSphere Application Server to the appropriate user. The SampleResponseSenderEncryptionKeyLocator key locator also can set a default key for encryption. By default, this key locator is configured to use public key alice.

Trusted ID Evaluator
Used to establish trust before asserting to the identity in identity assertion.

SampleTrustedIDEvaluator is configured to use the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl implementation. The default implementation of `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` contains a list of trusted identities. The list is defined as properties with `trustedId_*` as the key and the value as the trusted identity. Define this information for the server level in the administration console by completing the following steps:

1. Click **Servers > Application Servers >** *server1*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trusted ID Evaluators >** *SampleTrustedIDEvaluator*

For the cell level, click **Security > Web Services > Trusted ID Evaluators >** *SampleTrustedIDEvaluator*.

Login Mapping

Used to authenticate the incoming security token in the Web services security SOAP header of a SOAP message.

- The BasicAuth authentication method is used to authenticate user name security token (user name and password).
- The signature authentication method is used to map a distinguished name (DN) into a WebSphere Application Server Java Authentication and Authorization Server (JAAS) Subject.
- The IDAssertion authentication method is used to map a trusted identity into a WebSphere Application Server JAAS Subject for identity assertion.
- The Lightweight Third Party Authentication (LTPA) authentication method is used to validate a LTPA security token.

The previous default bindings for trust anchors, collection certificate stores, and key locators are for testing or sample purpose only. Do not use them for production.

A sample configuration

The following examples demonstrate what IBM deployment descriptor extensions and bindings can do. The unnecessary information was removed from the examples to improve clarity. Do not copy and paste these examples into your application deployment descriptors or bindings. These examples serve as reference only and are not representative of the recommended configuration.

Use the following tools to create or edit IBM deployment descriptor extensions and bindings:

- Use the Assembly Toolkit to create or edit the IBM deployment descriptor extensions.
- Use the Assembly Toolkit or the administrative console to create or edit the bindings file.

The following example illustrates a scenario that:

- Signs the SOAP body, time stamp, and security token.
- Encrypts the body content and user name token.
- Sends the user name token (basic authentication data).
- Generates the time stamp for the request.

For the response, the SOAP body and time stamp are signed, the body content is encrypted, and the SOAP message freshness is checked using the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints.

The request sender and the request receiver are a pair. Similarly, the response sender and the response receiver are a pair.

Tip: It is recommended that you use the WebSphere Application Server variables for specifying the path to the key stores. In the administrative console, click **Environment > Manage WebSphere**

Variables. These variables often help with platform differences such as file system naming conventions. In the following examples, `${USER_INSTALL_ROOT}` is used for specifying the path to the key stores.

Client-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

Request Sender
- Signs the SOAP body, time stamp and security token
- Encrypts the body content and user name token
- Sends the basic authentication token (user name and password)
- Generates the time stamp to expire in three minutes

Response Receiver
- Verifies that the SOAP body and time stamp are signed
- Verifies that the SOAP body content is encrypted
- Verifies that the time stamp is present (also check for message freshness)

Example 1: Sample client IBM deployment descriptor extension

The `xmi:id` statements are removed for readability. These statements must be added for this example to work.

Important: In the following code sample, lines 2 through 4 were split into three lines due to the width of the printed page.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscext=
  http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi">
  <serviceRefs serviceRefLink="service/myServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <clientServiceConfig actorURI="myActorURI">
        <securityRequestSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </integrity>
          <confidentiality>
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </confidentiality>
          <loginConfig authMethod="BasicAuth"/>
          <addCreatedTimeStamp flag="true" expires="PT3M"/>
        </securityRequestSenderServiceConfig>
        <securityResponseReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
          </requiredIntegrity>
          <requiredConfidentiality>
            <confidentialParts part="bodycontent"/>
          </requiredConfidentiality>
          <addReceivedTimeStamp flag="true"/>
        </securityResponseReceiverServiceConfig>
      </clientServiceConfig>
    </portQnameBindings>
  </serviceRefs>
</com.ibm.etools.webservice.wscext:WsClientExtension>
```

Client-side IBM extension bindings

Example 2 shows the client-side IBM extension binding for the security constraints described previously in the discussion on client-side IBM deployment descriptor extensions.

The signer key and encryption (decryption) key for the message can be obtained from the keystore key locator implementation (`com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`). The signer key is used for encrypting the response. The sample is configured to use the Java Certification Path API to validate the certificate path of the signer of the digital signature. The user name token (basic authentication) data is collected from the standard in (stdin) prompts using one of the default Java Authentication and Authorization Service (JAAS) implementations :`javax.security.auth.callback.CallbackHandler` implementation (`com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`).

Example 2: Sample client IBM extension binding

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wscbnd=
  "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">
  <serviceRefs serviceRefLink="service/MyServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <securityRequestSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="clientsignerkey" locatorRef="SampleClientSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <keyLocators name="SampleClientSignerKey" classname=
        "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr" path=
          "${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks" type="JKS"/>
          <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
        </keyLocators>
        <encryptionInfo name="EncInfo1">
          <encryptionKey name="CN=Bob, O=IBM, C=US" locatorRef=
          "SampleSenderEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SampleSenderEncryptionKeyLocator" classname=
        "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws" path=
          "${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks" type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
        </keyLocators>
        <loginBinding authMethod="BasicAuth" callbackHandler=
        "com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
      </securityRequestSenderBindingConfig>
      <securityResponseReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleClientTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <trustAnchors name="SampleClientTrustAnchor">
```

```
          <keyStore storepass="{xor}PDM2OjEr" path=
          "${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks" type="JKS"/>
        </trustAnchors>
        <certStoreList>
          <collectionCertStores provider="IBMCertPath" name="SampleCollectionCertStore">
            <x509Certificates path="${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer"/>
          </collectionCertStores>
        </certStoreList>
        <encryptionInfos name="EncInfo2">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator" classname=
        "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr" path=
          "${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks" type="JKS"/>
          <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
        </keyLocators>
      </securityResponseReceiverBindingConfig>
    </portQnameBindings>
  </serviceRefs>
</com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Server-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

Request Receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi`)

- Verifies that the SOAP body, time stamp, and security token are signed.
- Verifies that the SOAP body content and user name token are encrypted.
- Verifies that the basic authentication token (user name and password) is in the Web services security SOAP header.
- Verifies that the time stamp is present (also check for message freshness). The freshness of the message indicates whether the message complies with predefined time constraints.

Response Sender (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi`)

- Signs the SOAP body and time stamp
- Encrypts the SOAP body content
- Generates the time stamp to expire in 3 minutes

Example 3: Sample server IBM deployment descriptor extension

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wsext=
http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi">
  <wsDescExt wsDescNameLink="MyServ">
    <pcBinding pcNameLink="Port1">
      <serverServiceConfig actorURI="myActorURI">
        <securityRequestReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </requiredIntegrity>
          <requiredConfidentiality">
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </requiredConfidentiality>
          <loginConfig>
            <authMethods text="BasicAuth"/>
          </loginConfig>
          <addReceivedTimestamp flag="true"/>
        </securityRequestReceiverServiceConfig>
        <securityResponseSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
            <references part="timestamp"/>
          </integrity>
          <confidentiality>
            <confidentialParts part="bodycontent"/>
          </confidentiality>
          <addCreatedTimestamp flag="true" expires="PT3M"/>
        </securityResponseSenderServiceConfig>
      </serverServiceConfig>
    </pcBinding>
  </wsDescExt>
</com.ibm.etools.webservice.wsext:WsExtension>
```

Server-side IBM extension bindings

The following binding information reuses some of the default binding information defined either at the server level or the cell level, which depends upon the installation. For example, request receiver is referencing the `SampleCollectionCertStore` certification store and the `SampleServerTrustAnchor` trust store is defined in the default binding. However, the encryption information in the request receiver is referencing a `SampleReceiverEncryptionKeyLocator` key locator defined in the application-level binding (the same `ibm-webservices-bnd.xmi` file). The response sender is configured to use the signer key of the digital signature of the request to encrypt the response using one of the default key locator (`com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator`) implementations.

Example 4: Sample server IBM extension binding

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wsbnd=
  http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
  <wsdescBindings wsDescNameLink="MyServ">
    <pcBindings pcNameLink="Port1" scope="Session">
      <securityRequestReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleServerTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <encryptionInfos name="EncInfo1">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator" classname=
         "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws" path="${USER_INSTALL_ROOT}/
           etc/ws-security/samples/enc-receiver.jceks" type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
          <keys alias="bob" keypass="{xor}NDomLz4sLA==" name="CN=Bob, O=IBM, C=US"/>
        </keyLocators>
      </securityRequestReceiverBindingConfig>
      <securityResponseSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="serversignerkey" locatorRef="SampleServerSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <encryptionInfo  name="EncInfo2">
          <encryptionKey locatorRef="SignerKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SignerKeyLocator" classname=
         "com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator"/>
      </securityResponseSenderBindingConfig>
    </pcBindings>
  </wsdescBindings>
  <routerModules transport="http" name="StockQuote.war"/>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

View Web services client deployment descriptor

Use this page to view your client deployment descriptor.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **View Web Services Client Deployment Descriptor**.

Application-level, server-level, and cell-level are the three levels of bindings that WebSphere Application Server Network Deployment offers. The information in the following implementation descriptions indicates how to configure your application-level bindings. If the Web server is acting as a client, the default bindings are used. To configure the server-level bindings, which are the defaults, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security**.
3. To configure the cell-level bindings, click **Security > Web Services**.

If you are using any of the following configurations, verify that the deployment descriptor is configured properly:

- Request signing
- Request encryption
- BasicAuth authentication
- Identity (ID) Assertion authentication
- Identity (ID) Assertion authentication with the signature TrustMode
- Response digital signature verification
- Response decryption

Request signing

If the integrity constraints (digital signature) are specified, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* **Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit > Signing Information > New**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Request encryption

If the confidentiality constraints (encryption) are specified, verify that you configured the encryption information in the binding files.

To configure the encryption parameters, complete the following steps:
1. Click **Applications** > **Enterprise Applications** >*application_name*.
2. Under Related Items, click **Web Modules** >*URI_file_name* > **Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit** > **Encryption Information** > **New**.

To configure the key locators, complete the following steps:
1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Key Locators**.

BasicAuth authentication

If BasicAuth authentication is configured as the required security token, specify the CallbackHandler in the binding file to collect the basic authentication data. The following list contains the CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.GuiPromptCallbackHandler
This implementation prompts for BasicAuth information (user name and password) in an interface.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation reads the BasicAuth information from the binding file.

com.ibm.wsspi.wssecurity.auth.callback.StdPromptCallbackHandler
This implementation prompts for a user name and password using the standard in (stdin) prompt.

To configure the login binding information, 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 Request Sender Bindings, click **Edit** > **Login Binding**.

Identity (ID) Assertion authentication with BasicAuth TrustMode

Configure a login binding in the bindings file with a com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler implementation. Specify a BasicAuth user ID and password that a TrustedIDEvaluator on a downstream server trusts.

To configure the login binding information, 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 **Request Sender Bindings**, click **Edit** > **Login Binding**.

Identity (ID) Assertion authentication with the Signature TrustMode

Configure the signing information in the bindings file with a signing key pointing to a key locator. The key locator contains the X.509 certificate that is trusted by the downstream server.

To configure ID assertion, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Login Mappings** > **IDAssertion**.

To configure the login binding information, 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 Request Sender Bindings, click **Edit** > **Login Binding**.

Response digital signature verification

If the integrity constraints (signature required) are defined, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:

1. Click **Applications** > **Enterprise Applications** >*application_name*.
2. Under Related Items, click **Web Modules** >*URI_file_name* > **Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit** > **Signing Information** > **New**.

To configure the trust anchors, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Trust Anchors** > **New**.

To configure the collection certificate store, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Collection Certificate Store** > **New**.

Response decryption

If the confidentiality constraints (encryption) are specified, verify that you defined the encryption information.

To configure the encryption information, complete the following steps:

1. Click **Applications** > **Enterprise Applications** >*application_name*.
2. Under Related Items, click **Web Modules** >*<URI_file_name* > **Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit** > **Encryption Information** > **New**.

To configure the key locators, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Key Locators**.

View Web services server deployment descriptor

Use this page to view your server deployment descriptor settings.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Enterprise Applications** >*application_name*.

2. Under Related Items, click **Web Modules** > *URI_file_name* > **View Web Services Server Deployment Descriptor**.

WebSphere Application Server Network Deployment has three levels of bindings: application-level, server-level, and cell-level. The information in the following implementation descriptions indicate how to configure your application-level bindings. To configure the server-level bindings, which are the defaults, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security**.

To configure the cell-level bindings, click **Security > Web Services**.

- Request digital signature verification
- Request decryption
- BasicAuth authentication
- Identity (ID) Assertion authentication
- Identity (ID) Assertion authentication with the signature TrustMode
- Response signing
- Response encryption

Request digital signature verification

If the integrity constraints (signature required) are defined, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* > **Web Services: Server Security Bindings**.
3. In the Request Receiver Binding column, click **Edit > Signing Information**.

To configure the trust anchor, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

To configure the collection certificate store, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

To configure the key locators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Request decryption

If the confidentiality constraints (encryption) are specified, verify that the encryption information is defined.

To configure the encryption information parameters, complete the following steps:

1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Module.**

3. Under Additional Properties, click **Web Services: Server Security Bindings**. Under Request Receiver Binding, click **Edit > Encryption Information**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

BasicAuth authentication

If BasicAuth authentication is configured as the required security token, specify the CallbackHandler in the binding file to collect the basic authentication data. The following list contains CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.GuiPromptCallbackHandler
The implementation prompts for BasicAuth information (user name and password) in an interface panel.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation reads the BasicAuth information from the binding file.

com.ibm.wsspi.wssecurity.auth.callback.StdPromptCallbackHandler
This implementation prompts for a user name and password using the standard in (stdin) prompt.

To configure the login mapping information, complete the following steps:
1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

Identity (ID) Assertion authentication with the BasicAuth TrustMode

Configure a login binding in the bindings file with a com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler implementation. Specify a BasicAuth user ID and password that a TrustedIDEvaluator on a downstream server trusts.

To configure the login mapping information, complete the following steps:
1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

Identity (ID) Assertion authentication with the Signature TrustMode

Configure the signing information in the bindings file with a signing key pointing to a key locator. The key locator contains the X.509 certificate that is trusted by the downstream server.

To configure the login mapping information, complete the following steps:
1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

The Java Authentication and Authorization Service (JAAS) uses `WSLogin` as the name of the login configuration. To configure JAAS, click **Security > JAAS Configuration > Application Logins**.

The value of the `<TrustedIDEvaluatorRef>` tag in the binding must match the value of the `<TrustedIDEvaluator>` name.

To configure the trusted ID evaluators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Services, click **Web Services: Default bindings for Web Services Security > Trusted ID Evaluators**.

Response signing

If the integrity constraints (digital signature) are defined, verify that you have the signing information configured in the binding files.

To specify the signing information, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* **> Web Services: Server Security Bindings** .
3. In the Request Receiver Binding column, click **Edit > Signing Information**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Response encryption

If the confidentiality constraints (encryption) are specified, verify that the encryption information is defined.

To specify the encryption information, complete the following steps:
1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Module**.
3. Under Additional Properties, click **Web Services: Server Security Bindings**.
4. Under Request Receiver Binding, click **Edit > Encryption Information**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Authentication method overview

The Web services security implementation for WebSphere Application Server supports the following authentication methods: BasicAuth, Lightweight Third Party Authentication (LTPA), digital signature, and identity assertion.

When the WebSphere Application Server is configured to use the BasicAuth authentication method, the sender attaches the LTPA token as a BinarySecurityToken from the current security context or from basic authentication data configuration in the binding file in the SOAP message header. The Web services security message receiver authenticates the sender by validating the user name and password against the configured user registry. With the LTPA method, the sender attaches the LTPA BinarySecurityToken it previously received in the SOAP message header. The receiver authenticates the sender by validating the LTPA token and the token expiration time. With the Digital Signature authentication method, the sender attaches a BinarySecurityToken from a X509 certificate to the Web services security message header along with a digital signature of the message body, time stamp, security token, or any combination of the three. The receiver authenticates the sender by verifying the validity of the X.509 certificate and the digital signature using the public key from the verified certificate.

The identity assertion authentication method is different from the other three authentication methods. This method establishes the security credential of the sender based on the trust relationship. You can use the identity assertion authentication method, for example, when an intermediary server must invoke a service from a downstream server on behalf of the client, but does not have the client authentication information. The intermediary server might establish a trust relationship with the downstream server and then assert the client identity to the same downstream server.

Web Services Security supports the following trust modes:

- BasicAuth
- Digital signature
- Presumed trust

When you use the BasicAuth and digital signature trust modes, the intermediary server passes its own authentication information to the downstream server for authentication. The presumed trust mode establishes a trust relationship using some external mechanism. For example, the intermediary server might pass SOAP messages through a Secure Socket Layers (SSL) connection with the downstream server and transport layer client certificate authentication.

The Web services security implementation for WebSphere Application Server validates the trust relationship by following this procedure:

1. The downstream server validates the authentication information of the intermediary server.
2. The downstream server verifies whether the authenticated intermediary server is authorized for identity assertion. For example, the intermediary server must be in the trust list for the downstream server.

The client identity might be represented by a name string, a distinguished name (DN), or an X.509 certificate. The client identity is attached in the Web services security message in a UsernameToken with just a user name, DN, or in a BinarySecurityToken of a certificate. The following table summarizes the type of security token that is required for each authentication method.

Table 3. Authentication methods and their security tokens

Authentication method	**Security token**
BasicAuth	BasicAuth requires `<wsse:UsernameToken>` with `<wsse:Username>` and `<wsse:Password>`.
Signature	Signature requires `<ds:Signature>` and `<wsse:BinarySecurityToken>`.
IDAssertion	IDAssertion requires `<wsse:UsernameToken>` with `<wsse:Username>` or `<wsse:BinarySecurityToken>` with a X.509 certificate for client identity depending on `<idType>`. This method also requires other security tokens according to the `<trustMode>`: • If the `<trustMode>` is BasicAuth, IDAssertion requires `<wsse:UsernameToken>` with `<wsse:Username>` and `<wsse:Password>`. • If the `<trustMode>` is Signature, IDAssertion requires `<wsse:BinarySecurityToken>`.
LTPA	LTPA requires `<wsse:BinarySecurityToken>` with an LTPA token.

A Web service can support multiple authentication methods simultaneously. The receiver side of the Web services deployment descriptor can specify all the authentication methods that are supported in the `ibm-webservices-ext.xmi` XML file. The Web services receiver-side, as shown in the following example, is configured to accept all the authentication methods described previously:

```
<loginConfig xmi:id="LoginConfig_1052760331326">
      <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
      <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
      <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
      <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

You can define only one authentication method in the sender-side Web services deployment descriptor. A Web service client can use any of the authentication methods that are supported by the particular Web services application. The following example illustrates an identity assertion authentication method configuration in the `ibm-webservicesclient-ext.xmi` deployment descriptor extension of the Web service client:

```
<loginConfig xmi:id="LoginConfig_1051555852697">
      <authMethods xmi:id="AuthMethod_1051555852698" text="IDAssertion"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1051555852697" idType="Username" trustMode="Signature"/>
```

As shown in the previous example, the client identity type is `Username` and the trust mode is digital signature.

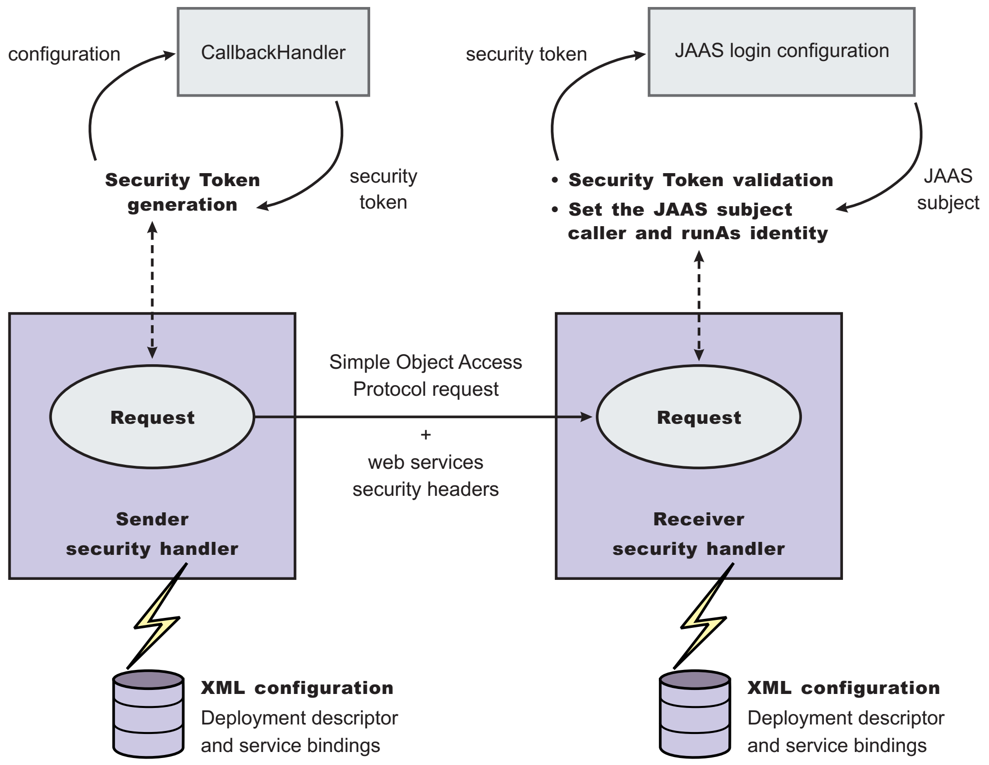

configuration
CallbackHandler
security token
Security Token generation
security token
JAAS login configuration
• Security Token validation
• Set the JAAS subject caller and runAs identity
JAAS subject
Request
Simple Object Access Protocol request
+
web services security headers
Request
Sender security handler
Receiver security handler
XML configuration
Deployment descriptor and service bindings
XML configuration
Deployment descriptor and service bindings

Figure 13. Security token generation and validation

The sender security handler invokes the `handle()` method of an implementation of the `javax.security.auth.callback.CallbackHandler` interface. The `javax.security.auth.callback.CallbackHandler` interface creates the security token and passes it back to the sender security handler. The sender security handler constructs the security token based on the authentication information in the callback array and inserts the security token into the Web services security message header.

The receiver security handler compares the token type in the message header with the expected token types configured in the deployment descriptor. If none of the expected token types are found in the Web services security header of the SOAP message, the request is rejected with a SOAP fault exception. Otherwise, the token type is used to map to a Java Authentication and Authorization Service (JAAS) login configuration for validating the token. If the authentication is successful, a JAAS Subject is created and associated with the running thread. Otherwise, the request is rejected with a SOAP fault exception.

XML digital signature

XML-Signature Syntax and Processing (XML signature) is a specification that defines XML syntax and processing rules to sign and verify digital signatures for digital content. The specification was developed jointly by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF).

XML signature does not introduce new cryptographic algorithms. WebSphere Application Server uses XML signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method.

XML canonicalization (c14n) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. For example, although their octet representations are different, the following examples are identical:

- `<person first="John" last="Smith"/>`
- `<person last="Smith" first="John"></person>`

C14n is a process used to canonicalize XML information. Select an appropriate c14n algorithm because the information that is canonicalized is dependent upon this algorithm. One of the major c14n algorithms, Exclusive XML Canonicalization, canonicalizes the character encoding scheme, attribute order, namespace declarations, and so on. The algorithm does not canonicalize white space outside tags, namespace prefixes, or data type representation.

XML signature in the Web Services Security-Core specification

The Web Services Security-Core (WSS-Core) specification defines a standard way for Simple Object Access Protocol (SOAP) messages to incorporate an XML signature. You can use almost all of the XML signature features in WSS-Core except enveloped signature and enveloping signature. However, WSS-Core has some recommendations such as exclusive canonicalization for the c14n algorithm and some additional features such as `SecurityTokenReference` and `KeyIdentifier`.

By including XML signature in SOAP messages, the following are realized:

Message integrity
A message receiver can confirm that attackers or accidents have not altered parts of the message after these parts are signed by a key.

Authentication
You can assume that a valid signature is *proof of possession*. A message with a digital certificate issued by a certificate authority and a signature in the message that is validated successfully by a public key in the certificate, is proof that the signer has the corresponding private key. The receiver can authenticate the signer by checking the trustworthiness of the certificate.

XML signature in the current implementation

XML signature is supported in Web services security, however, an application programming interface (API) is not available. The current implementation has many hardcoded behaviors and has some user-operable configuration items. To configure the client for digital signature, see Configuring the client for response digital signature verification: Verifying the message parts. To configure the server for digital signature, see Configuring the server for request digital signature verification: Verifying the message parts.

Security considerations

In a replay attack, an attacker taps the lines, receives a signed message, and then returns the message to the receiver. In this case, the receiver receives the same message twice and might process both of them if the signatures are valid. Processing both messages can cause damage to the receiver if the message is a claim for money. If you have the signed generation time stamp and the signed expiration time in a message replay, attacks might be reduced. However, this is not a complete solution. A message must have a nonce value to prevent these attacks and the receiver must reject a message that contains a processed nonce. The current implementation does not provide a standard way to generate and check nonces in messages. In WebSphere Application Server, Version 5.1, nonce is supported in username tokens only. The username token profile contains concrete nonce usage scenarios for username tokens. Applications handle nonces (such as serial numbers) and they need to be signed.

Signing information collection

Use this page to view a list of signing parameters. Signing information is used to sign and validate parts of a message including the body, time stamp, and user name token. You can also use these parameters for X.509 validation when the authentication method is IDAssertion and the ID type is X509Certificate in the server-level configuration. In such cases, you must fill in the certificate path fields only.
To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >** *URI_file_name* > **Web Services: Server Security Bindings** .
3. In the Request Receiver Binding column, click **Edit > Signing Information**.
4. Click **New** to create a signing parameter. Click **Delete** to delete a signing parameter.

Signature Method:

Specifies the unique name of the signature method.

Signing information configuration settings

Use this page to configure new signing parameters.
The specifications listed on this page for the signature method, digest method, and canonicalization method are located in the W3C document entitled, ″XMLSignature Syntax and Specification: W3C Recommendation 12 Feb 2002″.

To view this administrative console page:

1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Modules >** *URI_file_name* **> Web Services: Server Security Bindings**.
3. In the Request Receiver Binding column, click **Edit > Signing Information**.
4. Click **New** to create a signing parameter or click **Delete** to delete a signing parameter.

Signature Method:

Specifies the algorithm Uniform Resource Identifiers (URI) of the signature method. This method contains the actual value of the digital signature encoded using base64.

The following algorithms are supported:

- `http://www.w3.org/2000/09/xmldsig#rsa-sha1`
- `http://www.w3.org/2000/09/xmldsig#dsa-sha1`

Digest Method:

Specifies the algorithm URI of the digest method.

The http://www.w3.org/2000/09/xmldsig#sha1 algorithm is supported.

Canonicalization Method:

Specifies the algorithm URI of the canonicalization method.

The following algorithms are supported:
- `http://www.w3.org/2001/10/xml-exc-c14n#`
- `http://www.w3.org/2001/10/xml-exc-c14n#WithComments`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments`

Signing Key:

Specifies the key information that is used for signing. These fields are ignored in receiver-side configuration.

If you specify a **Key Name** and a **Key Locator Reference**, select **None** for the Certificate Path.

Certificate Path:

Specifies the settings for the certificate path validation. When you select **Trust Any**, this validation is skipped and all the incoming certificates are trusted. These fields are ignored in sender-side configuration.

If you click **Trust Any** or select a **Trust Anchor** and a **Certificate Store**, select **None** for the Signing Key in the previous field.

Trust Anchor

The selections available for Trust Anchor are specified by clicking **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

Certificate Store

The selections available for the Collection Store are specified by clicking **Servers > Application Servers >***server_name*. Under Related Items, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

Signing parameter configuration settings

Use this page to configure new signing parameters.
The specifications listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XMLSignature Syntax and Specification: W3C Recommendation 12 Feb 2002*.

To view this administrative console page, complete the following steps:
1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Modules >** *URI_file_name* > **Web Services: Client Security Bindings**.
3. In the Request Sender Binding column, click **Edit > Signing Information**.

If the signing information is not available, select **None**.

If the signing information is available, select **Dedicated Signing Information** and specify the configuration in the following fields:

Signature Method:

Specifies the algorithm Uniform Resource Identifiers (URI) of the signature method. This method contains the actual value of the digital signature encoded using base64.

The following algorithms are supported:
- `http://www.w3.org/2000/09/xmldsig#rsa-sha1`
- `http://www.w3.org/2000/09/xmldsig#dsa-sha1`

Digest Method:

Specifies the algorithm URI of the digest method.

The `http://www.w3.org/2000/09/xmldsig#sha1` algorithm is supported.

Canonicalization Method:

Specifies the algorithm URI of the canonicalization method.

The following algorithms are supported:
- `http://www.w3.org/2001/10/xml-exc-c14n#`
- `http://www.w3.org/2001/10/xml-exc-c14n#WithComments`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments`

Signing Key:

Specifies the key information that is used for signing. These fields are ignored in receiver-side configuration.

If the signing key is not available, select **None**.

Certificate Path:

Specifies the settings for the certificate path validation. When you select **Trust Any**, this validation is skipped and all the incoming certificates are trusted. These fields are ignored in sender-side configuration.

If there is not a certificate path, select **None**.

If there is a certificate path, select **Trust Any** or select a Trust Anchor and a Certificate Store.

Trust Anchor

Specify the selections for the Trust Anchor field by clicking **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

Certificate Store

Specify the selections for the Collection Store field by clicking **Servers > Application Servers***server_name*. Under Related Items, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

Securing Web services using XML digital signature

WebSphere Application Server provides several different methods to secure your Web services; Extensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

XML digital signature provides both message integrity and authentication capabilities when it is used with SOAP messages. A message receiver can verify that attackers or accidents have not altered parts of the message after the message was signed by a key. If a message has a digital certificate issued by a certificate authority (CA) and a signature in the message is validated successfully by a public key in the certificate, it is proof that the signer has the corresponding private key. To use XML digital signature to secure Web services, complete the following steps:

1. Define the security constraints or extensions. To configure the security constraints, you must use the Application Server Toolkit, which is available at the following Web site:

   http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP
   &q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en

   a. Configure the client to digitally sign a message request. To configure the client, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The client in these steps is the request sender.

      1) Specify the message parts by following the steps found in Configuring the client for request signing: digitally signing message parts.

      2) Select the method used to digitally sign the request message. You can select the digital signature method by following the steps in Configuring the client for request signing: choosing the digital signature method.

   b. Configure the server to verify the digital signature that is used in the message request. To configure the server, you must specify which parts of the SOAP message, sent by the request sender, contain digitally signed information and which method was used to digitally sign the message. The settings chosen for the request receiver, or the server in this step, must match the settings chosen for the request sender in the previous step.

      1) Define the message parts by following the steps found in Configuring the server for request digital signature verification: verifying message parts.

      2) Select the same method used by the request sender to digitally sign the message. You can select the digital signature method by following the steps in Configuring the server for request digital signature verification: choosing the verification method

   c. Configure the server to digitally sign a message response. To configure the server, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The sender in these steps is the response sender.

      1) Specify which message parts to digitally sign by following the steps found in Configuring the server for response signing: digitally signing message parts.

      2) Select the method used to digitally sign the response message. You can select the digital signature method by following the steps in Configuring the server for response signing: choosing the digital signature method

   d. Configure the client to verify the digital signature that is used in the message response. To configure the client, you must specify which parts of the SOAP message sent by the response sender contain digitally signed information and which method was used to digitally sign the message. The settings chosen for the response receiver, or client in this step, must match the settings chosen for the response sender in the previous step.

      1) Define the message parts by following the steps found in Configuring the client for response digital signature verification: verifying message parts

    2) Select the same method used by the response sender to digitally sign the message. You can select the digital signature method by following the steps in Configuring the client for response digital signature verification: choosing the verification method
2. Define the client security bindings. To configure the client security bindings, complete the steps in either of the following topics:
   - Configuring the client security bindings using the Application Server Toolkit
   - Configuring the client security bindings using the administrative console
3. Define the server security bindings. To configure the server security bindings, complete the steps in either of the following topics:
   - Configuring the server security bindings using the Application Server Toolkit
   - Configuring the server security bindings using the administrative console

After completing these steps, you have secured your Web services using XML digital signature.

Transport level security

Transport level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP.
SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections. To run with HTTPS, the service endpoint 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 for more information. Web services applications can also use Federal Information Processing Standard (FIPS) approved ciphers for more secure TLS connections.

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

Configuring HTTP outbound transport level security

The HTTP outbound transport-level security for a Web service is based on the Secured Sockets Layer (SSL) configuration of the WebSphere Application Server Web container. Review Configuring Secure Sockets Layer and HTTP transport collection for more information.

To configure HTTP outbound transport-level security, complete the following steps:
1. Configure the HTTP outbound transport-level security settings.
   - Use either the WebSphere Application Server administrative console or the Assembly Toolkit to configure the HTTP outbound transport-level security for a Web Service acting as a client to another Web service.
     – Before installing the Web services application, use the Assembly Toolkit to configure the HTTP SSL Configuration in the Web Services Client Port Binding page. The Web Services Client Port Binding page is available after double-clicking the `webservicesclient.xml` file.
     – After installing the Web services application, use the administrative console to configure the Web services client security binding collection. To access the collection, complete the following steps:
       a. Click **Applications > Enterprise Applications.**
       b. Under Related Items, click either **Web Modules** or **EJB Modules**.
       c. Click the name of the URI.
       d. Under Additional Properties, click **Web Services: Client Security Bindings**.

     Attention: If the HTTP outbound transport-level security settings are not configured, the default Secure Sockets Layer (SSL) settings for the Java Secure Socket Extension (JSSE) file are used.
   - Use the properties to configure the HTTP outbound transport-level security for a Web service client.

a. 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
```

b. Set the `com.ibm.webservices.sslConfigURL` Java system property to the absolute path of the created property file.

Attention: If the outbound transport-level security is not configured, the default SSL settings of the JSSE file are used.

2. Optional: Accept the redirection of 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.

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**.

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.

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint to the HTTP basic authentication. The basic authentication is located in the HTTP header that carries the SOAP request. 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 Secure Sockets Layer (SSL) protocol.

In some cases, a firewall is present using the 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

To configure HTTP basic authentication, use the WebSphere Application Server tools to modify the binding information or programmatically set properties in a Stub or Call object. The values set programmatically take precedence over the values defined in the binding. However, you only can programmatically configure the HTTP proxy authentication.

1. Use either the Assembly Toolkit or the administrative console to configure the HTTP basic authentication
   - Before installing the Web services application, use Assembly toolkit to 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 `webservicesclient.xml` file.
   - After installing the Web services application, you can use administrative console to configure the Web services client security bindings for a Web service only. To access the collection, complete the following steps:
     a. Click **Applications > Enterprise Applications.**
     b. Under Related Items, click either **Web Modules** or **EJB Modules**.
     c. Click the name of the URI.
     d. Under Additional Properties, click **Web Services: Client Security Bindings**.
2. Programmatically set the properties in the stub or call object for a Web service or a Web service client Programmatically 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
   ```
3. Programmatically set the properties in the stub or call object to configure the HTTP proxy authentication. Programmatically 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
   ```

   Programmatically 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
   ```

HTTP basic authentication collection:

Use this page to specify a user ID 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 ID for the HTTP basic authentication for this port.

Basic Authentication Password:

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

Trust anchors

A *trust anchor* specifies key stores that contain trusted root certificates that validate the signer certificate. These key stores are used by the request receiver (as defined in the ibm-webservices-bnd.xmi file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when Web services is acting as client) to validate the signer certificate of the digital signature. The key stores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these key stores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The trust anchor is defined as javax.security.cert.TrustAnchor in the Java CertPath application programming interface (API). The Java CertPath API uses the trust anchor and the certificate store to validate the incoming X.509 certificate that is embedded in the SOAP message.

The Web services security implementation in WebSphere Application Server supports this trust anchor. In WebSphere Application Server, the trust anchor is represented as a Java key store object. The type, path, and password of the key store are passed to the implementation through the administrative console or by scripting.

Configuring trust anchors using the Assembly Toolkit

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. You can configure an application-level trust anchor using the Assembly Toolkit or the administrative console. This document describes how to configure the application-level trust anchor using the Assembly Toolkit. For more information on creating and configuring trust anchors at the server or cell level, see either Configuring the server security bindings using the Assembly Toolkit or Configuring the server security bindings using the administrative console.

A trust anchor specifies key stores that contain trusted root certificates, which validate the signer certificate. These key stores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when Web services is acting as client) to validate the signer certificate of the digital signature. The key stores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these key stores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The steps in this document assume that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with Java Specification Requests (JSR) 109 enterprise application. If you have not created a Web services-enabled J2EE with JSR 109 enterprise application, see Developing Web services. Also, see either Configuring the server security bindings using the Assembly Toolkit or Configuring the server security bindings using the administrative console for an introduction on how to manage Web services security binding information on the server.

1. Configure the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` bindings extensions file.
   a. Launch the Assembly Toolkit and click **Windows > Open Perspective > J2EE**.
   b. Select the Web services-enabled Enterprise JavaBeans (EJB) or Web module.
   c. In the Package Explorer window, click the `META-INF` directory for an EJB module or the `WEB-INF` directory for a Web module.
   d. Right-click the **`webservicesclient.xml`** file, select **Open With > Web Services Client Editor**, and click the **Web Services Client Binding** tab. The Web Services Client Binding editor is displayed.
   e. Locate the Port Qualified Name Binding section and either select an existing entry or click **Add**, to add a new port binding. The Web Services Client Port Binding editor displays for the selected port.
   f. Locate the Trust Anchor section and click **Add**. The Trust Anchor dialog box is displayed.
      1) Enter a unique name within the port binding for the **Trust anchor name**.
         The name is used to reference the trust anchor that is defined.
      2) Enter the key store password, path, and key store type.
         The supported key store types are Java Cryptography Extension (JCE) and JCEKS.

      Click **Edit** to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
   g. Save the changes.
2. Configure the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` bindings extensions file.
   a. Launch the Assembly Toolkit and click **Windows > Open Perspective > J2EE**.
   b. Select the Web services enabled EJB or Web module.
   c. In the Package Explorer window, click the `META-INF` directory for an EJB module or the `WEB-INF` directory for a Web module.
   d. Right-click the **`webservices.xml`** file, select **Open With > Web Services Editor**, and click the **Bindings** tab. The Web Services Bindings editor is displayed.
   e. Locate the Web Service Description Bindings section and either select an existing entry or click **Add** to add a new Web services descriptor.
   f. Click **Binding Configurations**. The Web Services Binding Configurations editor is displayed for the selected Web services descriptor.
   g. Locate the Trust Anchor section and click **Add**. The Trust Anchor dialog box is displayed.
      1) Enter a unique name within the binding for the **Trust anchor name**.
         This unique name is used to reference the trust anchor defined.
      2) Enter the key store password, path, and key store type. The supported key store types are JCE and JCEKS.

Click **Edit** to edit the selected trust anchor.

Click **Remove** to remove the selected trust anchor.

When you start the application, the configuration is validated in the run time while the binding information is loading.

h. Save the changes.

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the Web services is acting as client) to verify the signer certificate.

The request receiver or the response receiver (if the Web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

To complete the signing information configuration process for request receiver, complete the following tasks:

1. Configure the server for request digital signature verification: Verifying the message parts
2. Configure the server for request digital signature verification: Choosing the verification method

To complete the process for the response receiver, if the Web services is acting as a client, complete the following tasks:

1. Configure the client for response digital signature verification: Verifying the message parts
2. Configure the client for response digital signature verification: Choosing the verification method

Configuring trust anchors using the administrative console

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. For more information on creating and configuring trust anchors at the server or cell level, see either Configuring the server security bindings using the Application Server Toolkit or Configuring the server security bindings using the administrative console.

You can configure an application-level trust anchor using the Application Server Toolkit or the administrative console. This document describes how to configure the application-level trust anchor using the administrative console.

A trust anchor specifies key stores that contain trusted root certificates, which validate the signer certificate. These key stores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when Web services is acting as client) to validate the signer certificate of the digital signature. The keystores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The steps in this document assume that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with Java Specification Requests (JSR) 109 enterprise application. If you have not created a Web services-enabled J2EE with JSR 109 enterprise application, see Developing Web services. Also, see either Configuring the server security bindings using the Application Server Toolkit or Configuring the server security bindings using the administrative console for an introduction on how to manage Web services security binding information for the server.

The following steps are for the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` file and the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` file.

1. Click **Applications** > **Enterprise Applications** > *enterprise_application*.
2. In the Related Links section, click either **EJB Modules** or **Web Modules** and then click the Web services-enabled module in the **Uri** field.
3. Under Additional Properties, click **Web Services: Client Security Bindings** to edit the response receiver binding information, if Web services is acting as a client.
   a. Under Response Receiver Binding, click **Edit**.
   b. Under Additional Properties, click **Trust Anchors**.
   c. Click **New** to create a new trust anchor.
   d. Enter a unique name within the request receiver binding for the **Trust anchor name** field. The name is used to reference the trust anchor that is defined.
   e. Enter the key store password, path, and key store type.
   f. Click the trust anchor name link to edit the selected trust anchor.
   g. Click **Remove** to remove the selected trust anchor or anchors.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
4. Return to the Web services-enabled module panel accessed in step 2.
5. Under Additional Properties, click **Web Services: Server Security Bindings** to edit the request receiver binding information.
   a. Under Request Receiver Binding, click **Edit**.
   b. Under Additional Properties, click **Trust Anchors**.
   c. Click **New** to create a new trust anchor

      Enter a unique name within the request receiver binding for the **Trust anchor name** field. The name is used to reference the trust anchor that is defined.

      Enter the key store password, path, and key store type.

      Click the trust anchor name link to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor or anchors.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
6. Save the changes.

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the Web services is acting as client) to verify the signer certificate.

The request receiver or the response receiver (if the Web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

To complete the signing information configuration process for request receiver, complete the following tasks:

1. Configure the server for request digital signature verification: Verifying the message parts
2. Configure the server for request digital signature verification: Choosing the verification method

To complete the process for the response receiver, if the Web services is acting as client, complete the following tasks:

1. Configure the client for response digital signature verification: Verifying the message parts
2. Configure the client for response digital signature verification: Choosing the verification method

Trust anchors collection:

Use this page to view a list of keystore objects that contain trusted root certificates. These objects are used for certificate path validation of incoming X.509-formatted security tokens. Keystore objects within trust anchors contain trusted root certificates used by the CertPath API to validate the trust of a certificate chain.

To create the keystore file, use the key tool located in the `install_dir\java\jre\bin\keytool` directory.

To view this administrative console page, click **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

Click **New** to create a new trust anchor.

Click **Delete** to delete a trust anchor.

If you click **Update runtime**, the Web services security run time is updated with the default binding information, which is contained in the `ws-security.xml` file that was previously saved. If you make changes on this panel, you must complete the following steps:
1. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you are returned to the administrative console home panel.
2. Return to the Trust Anchors collection panel and click **Update runtime**. When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.

Trust Anchor Name:

Specifies the unique name used to identify the trust anchor.

Key Store Path:

Specifies the location of the keystore file that contains the trust anchors.

Key Store Type:

Specifies the type of keystore file.

The value for this field is either **JKS** or **JCEKS**.

Trust anchor configuration settings:

Use this information to configure a trust anchor. Trust anchors point to key stores that contain trusted root or self-signed certificates. This information enables you to specify a name for the trust anchor and the information needed to access a key store. The application binding uses this name to reference a predefined trust anchor definition in the binding file (or the default).

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors > New**.

Trust Anchor Name:

Specifies the unique name used by the application binding to reference a predefined trust anchor definition in the default binding.

Key Store Password:

Specifies the password needed to access the key store file.

Key Store Path:

Specifies the location of the keystore file.

Use ${USER_INSTALL_ROOT} as this path expands to the WebSphere Application Server path on your machine.

Key Store Type:

Specifies the type of key store file.

The value in this field is either **JKS** or **JCEKS**.

JKS Specify this option if you are not using Java Cryptography Extensions (JCE).

JCEKS
: Specify this option if you are using Java Cryptography Extensions. Although the JCEKS key store format is more secure, it decreases performance.

Data type	String
Default	JKS
Range	JKS, JCEKS

Collection certificate store

A *collection certificate store* is a collection of nonroot, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message.
The collection certificate stores are used when processing a received SOAP message. This collection is configured in the securityRequestReceiverBindingConfig section of the binding file for servers and in the securityResponseReceiverBindingConfig section of the binding file for clients.

A collection certificate store is one kind of certificate store. A certificate store is defined as javax.security.cert.CertStore in the Java CertPath application programming interface (API). The Java CertPath API defines the following types of certificate stores:

Collection certificate store
: A collection certificate store accepts the certificates and CRLs as Java collection objects.

Lightweight Directory Access Protocol certificate store
: The Lightweight Directory Access Protocol (LDAP) certificate store accepts certificates and CRLs as LDAP entries.

The CertPath API uses the certificate store and the trust anchor to validate the incoming X.509 certificate that is embedded in the SOAP message.

The Web services security implementation in the WebSphere Application Server supports the collection certificate store. Each certificate and CRL is passed as an encoded file. This configuration is done using either the administrative console or by scripting.

Configuring the client-side collection certificate store using the Application Server Toolkit

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the WebSphere Application Server Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the WebSphere Application Server Toolkit.

1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Port Binding** tab in the Web Services Client Editor within the WebSphere Application Server Toolkit. The Web Services Client Port Binding window displays.
6. Select one of the Port Qualified Name Binding entries.
7. Expand the **Security Response Receiver Binding Configuration > Certificate Store List > Collection Certificate Store** section.
8. Click **Add** to create a new collection certificate store, **Edit** to edit an existing certificate store, or **Remove** to delete an existing certificate store.
9. Enter a name in the **Name** field. This is a name that is referenced in the **Certificate store reference** field in the Signing info dialog box.
10. Leave the **Provider** field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be:`${`*`USER_INSTALL_ROOT`*`}/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you are done adding paths.

Configuring the client-side collection certificate store using the administrative console

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the WebSphere Application Server Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the administrative console.

1. Connect to the WebSphere Application Server administrative console. You can connect to the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click either **Web Services: Client Security Bindings** to add the collection certificate store to the client security bindings. If you do not see any entries, return to the WebSphere Application Server Toolkit and configure the security extensions for either the client or the server.

   To configure the security extensions for the client, see the following topics:

   - Configuring the client for response digital signature verification: verifying the message parts
   - Configuring the client for response digital signature verification: choosing the verification method
6. Click **Edit** under Response Receiver Binding to edit the client security bindings.
7. Click **Collection Certificate Store**.

8. Click a Certificate Store Name to edit an existing certificate store or click **New** to add a new certificate store name.
9. Enter a name in the **Certificate Store Name** field. The name entered in this field is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
10. Leave the **Certificate Store Provider** field as `IBMCertPath`.
11. Click **Apply**.
12. Under Additional Properties, click **X.509 Certificates > New**.
13. Enter the path to your certificate store. For example, the path might be: `${`*`USER_INSTALL_ROOT`*`}/etc/ws-security/samples/intca2.cer`. If you have any additional certificate store paths to enter, click **New** and add the path names.
14. Click **OK**.

Configuring the server-side collection certificate store using the Assembly Toolkit

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the Assembly Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the server-side collection certificate store using the Assembly Toolkit.

1. Launch the Assembly Toolkit and click**Windows > Open Perspective > J2EE**.
2. Select the Web services-enabled Enterprise JavaBean (EJB) or Web module.
3. In the Package Navigator window, locate the `META-INF` directory for an EJB module or the `WEB-INF` directory for a Web module.
4. Right-click the **`webservices.xml`** file, select **Open With > Web Services Editor**.
5. Click the **Binding Configurations** tab in the Web services editor within the Assembly Toolkit. The Web Service Binding Configuration window is displayed.
6. Select one of the Web service description binding entries under the Port Component Binding section.
7. Expand the **Request Receiver Binding Configuration Details > Certificate Store List > Collection Certificate Store** section.
8. Click **Add** to create a new collection certificate store, click **Edit** to edit an existing certificate store, or click **Remove** to delete an existing certification store.
9. Enter a name in the **Name** field. This name is referenced in the **Certificate store reference** field in the Signing info dialog.
10. Leave the **Provider** field as IBMCertPath.
11. Click **Add** to enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT]/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you finish adding paths.

Collection certificate store collection:

Use this page to view a list of certificate stores containing untrusted, intermediary certificate files awaiting validation. Validation might consist of checking to see if the certificate is on a certificate revocation list (CRL), checking that the certificate has not expired, and checking that the certificate was issued by a trusted signer.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers** *server_name*.

2. Under **Related Items**, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
3. Click **New** to specify a store name and provider for a new collection certificate store.
4. Click **Delete** to delete a collection certificate store.

Using this panel, complete the following steps:
1. Specify a certificate store name and certificate store provider.
2. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you are returned to the administrative console home panel.
3. Return to the collection certificate store collection panel and click **Update runtime**to update the Web services security run time with the default binding information, which is found in the `ws_security.xml` file.

Important: When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.

Certificate Store Name:

Specifies the name of the certificate store.

Certificate Store Provider:

Specifies the provider of the certificate store.

Collection certificate store configuration settings:

Use this page to specify the name and provider of a certificate store.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >***server_name*.
2. Under **Additional Properties**, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store > New**.

Certificate Store Name:

Specifies the name for the certificate store. The application binding uses the certificate store name to reference a predefined binding.

Certificate Store Provider:

Specifies the provider for the certificate store implementation.

Data type	String
Default	IBM CertPath

X.509 certificates collection:

Use this page to view a list of X.509 certificates.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

3. On the Collection Certificate Store page under Additional Properties, click **X.509 Certificates**.

Click **New** to create a new path to an X.509 certificate.

Click **Delete** to delete a path to an X.509 certificate.

X509 Certificate Path:

Specifies the location of the X.509 certificate.

X.509 certificate configuration settings:

Use this page to specify the location of your X.509 certificates.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >***server_name*.
2. Under **Additional Properties**, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
3. On the *Collection Certificate Store* page, under **Additional Properties**, click **X.509 Certificates > New**.

X509 Certificate Path:

Specifies the location of the X.509 certificate.

Configuring the server-side collection certificate store using the administrative console

A *collection certificate store* is a collection of nonroot, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the WebSphere Application Server Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the server-side collection certificate store using the administrative console.
1. Connect to the WebSphere Application Server administrative console. You can connect to the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click **Web Services: Server Security Bindings** to add the collection certificate store to the server security bindings. If you do not see any entries, return to the WebSphere Application Server Toolkit and configure the security extensions for the server.

   To configure the security extensions for the server, see the following topics:
   - Configuring the server for request digital signature verification: verifying the message parts
   - Configuring the server for request digital signature verification: choosing the verification method
6. Click **Edit** under Request Receiver Binding to edit the server security bindings.
7. Click **Collection Certificate Store**.
8. Click a Certificate Store Name to edit an existing certificate store or click **New** to add a new certificate store name.

9. Enter a name in the **Certificate Store Name** field. The name entered in this field is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
10. Leave the **Certificate Store Provider** field as `IBMCertPath`.
11. Click **Apply**.
12. Under Additional Properties, click **X.509 Certificates > New**.
13. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have any additional certificate store paths to enter, click **New** and add the path names.
14. Click **OK**.

Configuring default collection certificate stores at the server level in the WebSphere Application Server administrative console

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. A certificate store typically refers to a certificate store located in the file system. The location of the certificate store can vary from machine to machine so you might configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates. You also can specify the default binding information at the cell level.

Complete the following steps to configure the default collection certificate store at the server level using the WebSphere Application Server administration console:
1. Connect to administrative console. You can access the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Servers > Application Servers >** *server1*.
3. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
4. Enter a name in the **Certificate Store Name** field. This is a name that is referenced in the Certificate Store field on the Signing information configuration page.
5. Leave the **Certificate Store Provider** field as `IBMCertPath`.
6. Click **Apply**.
7. Under Additional Properties, click **X.509 Certificates > New**.
8. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.

   If you have any additional certificate store paths to enter, click **New** and add the path names.
9. Click **OK**.

Configuring default collection certificate stores at the cell level in the WebSphere Application Server administrative console

A *collection certificate store* is a collection of nonroot, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message. A certificate store typically refers to a certificate store located in the file system. The location of the certificate store can vary from machine to machine so you might configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates. You also can specify the default binding information at the server level.

Complete the following steps to configure the default collection certificate store at the cell level using the WebSphere Application Server administration console:

1. Connect to administrative console. You can access the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Security > Web Services > Collection Certificate Store**.
3. Click a listed Certificate Store Name to edit an existing store, or click **New** to add a new store. This is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
4. Leave the **Certificate Store Provider** field as `IBMCertPath`.
5. Click **Apply**.
6. Under Additional Properties, click **X.509 Certificates > New**.
7. Enter the path to your certificate store. For example, the path might be: `${`*`USER_INSTALL_ROOT`*`}/etc/ws-security/samples/intca2.cer`

   If you have any additional certificate store paths to enter, click **New** and add the path names.
8. Click **OK**.

Key locator

A key locator (`com.ibm.wsspi.wssecurity.config.KeyLocator`) is a abstraction of the mechanism that retrieves the key for digital signature and encryption.
You can use any of the following infrastructure from which to retrieve the keys depending upon the implementation:

- Java keystore file
- Database
- LDAP server

Key locators search the key using some type of a clue. The following types of clues are allowed:

- A string label of the key, which is explicitly passed through the application programming interface (API). The relationships between each key and its name (string label) is maintained inside the key locator.
- The execution context of the key locator; explicit information is not passed to the key locator. A key locators, by itself, determines the appropriate key according to their execution context.

For example, key locators can obtain the identity of the caller from the context and can retrieve the public key of the caller for response encryption.

Restriction: Current versions of key locators do not support the retrieval of verification keys because current Web services security implementations do not support the secret key-based signature. Since the key locators support the public key-based signature only, the key for verification is embedded in the X.509 certificate as a `<BinarySecurityToken>` element in the incoming message.

Usage scenarios

This section describes the usage scenarios for key locators.

Signing

The name of the signing key is specified in the Web services security configuration. This value is passed to the key locator and the actual key is returned. The corresponding X.509 certificate also can be returned.

Verification

As described previously, key locators are not used in signature verification.

Encryption

The name of the encryption key is specified in the Web services security configuration. This value is passed to the key locator and the actual key is returned.

Decryption

The Web services security specification recommends the usage of the key identifier instead of the key name. However, while the algorithm for computing the identifier for the public keys is defined in Internet Engineering Task Force (IETF) Request for Comment (RFC) 3280, there is no agreed upon algorithm for the secret keys. Therefore, the current implementation of Web services security uses the identifier only when public key-based encryption is performed. Otherwise, the ordinal key name is used.

When you use public key-based encryption, the value of key identifier is embedded in the incoming encrypted message. Then, the Web services security implementation searches for all the keys managed by the key locator and decrypts the message using the key whose identifier value matches the one in the message.

When you use secret key-based encryption, the value of key name is embedded in the incoming encrypted message. The Web services security implementation asks the key locator for the key whose name matches the one in the message and decrypts the message using the key.

Key locator collection:

Use this page to view a list of available key locators. Key locators identify the keys needed for digital signature and encryption. A key locator must implement the com.ibm.wsspi.wssecurity.config.KeyLocator interface. The two default implementations are:
com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator and
com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator.

To view this administrative console page, complete the following steps:
1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Key Locators**.
3. Click **New** to create a key locator. Click **Delete** to delete a key locator.

Using this panel, complete the following steps:
1. Specify a key locator name and key locator class name on the panel
2. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you return to the administrative console home panel.
3. After saving your changes, return to the **Key Locator** collection panel to update the Web services security run time with the default binding information, which is found in the `ws_security.xml` file.
4. To update the Web services security run time, click **Update runtime**. When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.
5. Once you define key locators, click the key locator name to specify additional properties and keys under **Additional Properties**.

Key Locator Name:

Specifies the unique name of the key locator.

Key Locator Classname:

Specifies the class name of the key locator in the keystore file.

Key locator configuration settings:

Use this page to specify the settings for key locators.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators > New**.

Key Locator Name:

Specifies the name of the key locator.

Data type	String

Key Locator Classname:

Specifies the name for the key locator class implementation.

WebSphere Application Server has the following default key locator class implementations:

com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator
: Maps an authenticated identity to a key. This class is used by the response sender. If encryption is used, this class is used to locate a key to encrypt the response message. The com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, user1 is mapped to mappedName_1. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
: Maps a name to an alias. This class is used by the response receiver, request sender, and request receiver. The encryption process uses this class to obtain a key to encrypt a message, and the digital signature process uses this class to obtain a key to sign a message. The com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator class maps a logical name to a key alias in the keystore file. For example, key #105115176771 is mapped to `CN=Alice, O=IBM, c=US`.

Data type	String
Defaults	com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator

Key Store Password:

Specifies the password used to access the keystore file.

Key Store Path:

Specifies the location of the keystore file.

Use `${USER_INSTALL_ROOT}` as this path expands to the WebSphere Application Server path on your machine.

Key Store Type:

Specifies the type of keystore file.

The value for this field is either JKS or JCEKS:

JKS Use this option if you are not using Java Cryptography Extensions (JCE).

JCEKS
Use this option if you are using Java Cryptography Extensions.

Default	JKS
Range	JKS, JCEKS

Keys

Keys are used for XML signature and encryption.

Largely, there two kinds of keys are used in the current Web services security implementation:

- Public key - such as RSA and DSA
- Secret key - such as DES

In public-key-based signature, a message is signed using sender private key and is verified using the sender public key. In public key-based encryption, a message is encrypted using receiver public key and is decrypted using the receiver private key. In secret key-based signature and encryption, the same key is used by both parties.

While the current implementation of Web services security can deal with both kinds of keys, there are a few things to be noted:

- Secret key-based signature is not supported.
- The format of the message differ slightly between public key-based encryption and secret key-based encryption.

Key collection:

Use this page to view a list of logical names that are mapped to a key alias in the keystore file.

To view this administrative console page, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security** > **Key Locators** > *key_locator_name*.
3. Under Additional Properties, click **Keys**.
4. Click **New** to create a new key object in the keystore file.
5. Click **Delete** to a delete a mapping of a key object within the keystore file.

Key Name:

Specifies the name of the key object found in the keystore file.

Key Alias:

Specifies an alias for the key object.

The alias is used when the key locator searches for the key objects in the keystore.

Key configuration settings:

Use this page to define a mapping of a logical name to a key alias in a keystore file.

To view this administrative console page, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.

2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators >** *key_locator_name*.
3. Under Additional Properties, click **Keys > New**.

Key Name:

Specifies the name of the key object. This name is used by the key locator to find the key within the keystore file.

Key Alias:

Specifies the alias for the key object contained in the keystore file.

Key Password:

Specifies the password needed to access the key object within the keystore file.

Web services security service provider programming interfaces

Several Service Provider Programming Interfaces (SPIs) are provided to extend the capability of the Web services security run time. The following is a list of SPIs that are available for WebSphere Application Server:

- `com.ibm.wsspi.wssecurity.config.KeyLocator` is an abstract for obtaining the keys for digital signature and encryption. The following are the default implementations:
  - `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`

    Implements the Java keystore.
  - `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator`

    Provides a mapping of authenticated identity to a key for encryption or use the default key specified. This is typically used in the response sender configuration.
  - `com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator`

    Provides the capability of using the signer key for encryption in the response message. This is typically used in the response sender configuration.
- `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` is an interface that used to evaluate the trust for identity assertion. The following is the default implementation:
  - `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`

    Enables you to define a list of trusted identities.
- The JAAS CallbackHandler Application Programming Interfaces (APIs) are used for token generation by the request sender. This can be extended to generate custom token to be inserted in the Web services security header. The following are the default implementations are provided by WebSphere Application Server:
  - `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`

    Presents a login prompt to gather the basic authentication data. Use this implementation in the client environment only.
  - `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`

    Collects the basic authentication data in the standard in (stdin) prompt. Use this implementation in the client environment only.
  - `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

    Reads the basic authentication data from the application binding file. This might be used on the server side to generate a user name token.
  - `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`

Generates Lightweight Third Party Authentication (LTPA) token in the Web services security header as binary security token. If there is basic authentication data defined in the application binding file, it is used to perform a login, extract the LTPA token from the WebSphere credentials, and insert the token in the Web services security header. Otherwise, it will extract the LTPA security token from the invocation credentials (run as identity) and insert the token in the Web services security header.

The JAAS LoginModule API is used for token validation on the request receiver side of the message. You can implement a custom LoginModule to perform validation of the custom token on the request receiver of the message. Once the token is verified and validated, the token is set as the caller, run as identity in the WebSphere run time, and the identity is used for authorization checks by the containers before a Java 2 Platform, Enterprise Edition (J2EE) resource is invoked. The following are the default ″AuthMethod″ configurations provided by WebSphere Application Server:

BasicAuth
Validates a user name token

Signature
Maps a distinguished name (DN) of a verified certificate to a Java Authentication and Authorization Service (JAAS) subject.

IDAssertion
Maps a trusted identity to a JAAS subject.

LTPA Validates an LTPA token received in the message and creates a JAAS subject.

Configuring key locators using the Assembly Toolkit

This task provides instructions on how to configure key locators using the Assembly Toolkit. You can configure key locators in various locations within the Assembly Toolkit. This task provides instructions how to configure key locators at any of these locations because the concept is the same.

1. Launch the Assembly Toolkit and click **Windows > Open Prospective > J2EE**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file and click **Open With > Web Services Client Editor** or right-click the `webservices.xml` file and click **Open With > Web Services Editor**.
5. Click the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit or the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit.
6. Expand one of the Binding Configuration sections.
7. Expand the Key Locators section.
8. Click **Add**to create a new key locator, **Edit** to edit an existing key locator, or **Remove** to delete an existing key locator.
9. Enter a key locator name. The name entered for the **Key locator name** is used to refer to the key locator from the Encryption information and Signing Information sections.
10. Enter a key locator class. The key locator class is the implementation of the `KeyLocator` interface. When using default implementations, select a class from the menu
11. Determine whether to click **Use key store**. Select this option when you use the default implementations as they use key stores. If you click **Use key store**, complete the following steps:
    a. Enter a value in the **key store storepass** field. The key store storepass is the password used to access the key store.
    b. Enter a path name in the **key store path** field. The key store path is the location on the file system where the key store resides. Make sure that the location can be found wherever you deploy the application.

c. Enter a type value in the **key store type** field. The valid types to enter are JKS and JCEKS. JKS is used when you are not using Java Cryptography Extensions (JCE). JCEKS is used when you are using JCE. Although the JCEKS type is more secure, it might decrease performance.

d. Click **Add** to create an entry for a key in the key store.

1) Enter a value in the **Alias** field.

The key alias is a reference to this particular key from the Signing Information section.

2) Enter a value in the **Key pass** field.

The key pass is the password associated with the certificate when created using Java Development Kit, `keytool.exe`.

3) Enter a value in the **Key name** field.

The key name refers to the alias of the certificate as found in the key store.

12. Click **Add** to create a custom property. The property can be used by custom implementations of `KeyLocator`. For example, you can use properties with the `WSIdKeyStoreMapKeyLocator` default implementation. The `KeyLocator` has the following property names:

- *id_*, which maps to a credential user ID.
- *mappedName_* , which maps to the key alias to use for this user name.
- *default*, which maps to a key alias to use when a credential does not have an associated *id_* entry

A typical set of properties for this key locator might be: `id_1=user1`, `mappedName_1=key1`, `id_2=user2`, `mappedName_2=key2`, `default=key3`. If `user1` or `user2` authenticates, then the associated `key1` or `key2` is used, respectively. However, if none of the user properties authenticate or the user is not `user1` or `user2`, then `key3` is used.

a. Enter a name in the **Name** field. The name entered is the property name.

b. Enter a value in the **Value** field. This value entered is the property value.

Configuring key locators using the administrative console

This task provides instructions on how to configure key locators using the WebSphere Application Server administrative console. You can configure binding information in the administrative console, but for extensions, you must use the Application Server Toolkit. The following steps are used to configure a key locator in the administrative console for a specific application:

1. Connect to administrative console by typing `http://:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules**, depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click either **Web Services: Client Security Bindings** or **Web Services: Server Security Bindings** depending on whether you are adding the key locator to the client security bindings or the server security bindings. If you do not see any entries, return to the WebSphere Application Server Toolkit and configure the Security Extensions.
6. Edit the Request Sender Binding, Response Receiver Binding, Request Receiver Binding, or Response Sender Binding
   - If you are editing your client security bindings, click **Edit** for either the Request Sender Binding or Response Receiver Binding.
   - If you are editing your server security bindings, click **Edit** for either the Request Receiver Binding or Response Sender Binding.
7. Click **Key Locators**.

8. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
   a. Specify a name for the key locator in the **Key Locator Name** field.
   b. Specify a name for the key locator class implementation in the **Key Locator Classname** field. WebSphere Application Server has the following default key locator class implementations:

      `com.ibm.wsspi.wssecurity.config.WSldKeyStoreMapKeyLocator`
      This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSldKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

      `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`
      This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice`, `O=IBM`, `c=US`.
   c. Specify the password used to access the keystore password in the **Key Store Password** field. This field is optional is the key locator does not use a keystore.
   d. Specify the path name used to access the keystore in the **Key Store Path** field. This field is optional is the key locator does not use a keystore. Use `${`*`USER_INSTALL_ROOT`*`}` as this path expands to the WebSphere Application Server path on your machine.
   e. Select a keystore type from the **Key Store Type** field. This field is optional is the key locator does not use a keystore. Use the **JKS** option if you are not using Java Cryptography Extensions (JCE) and use **JCEKS** if you are using JCE.

Configuring server and cell level key locators using the administrative console

A key locator typically locates a key store in the file system. The location of key stores can vary from machine to machine so it is often helpful to configure a default key locator for a specific machine and reference it from within the encryption or signing information. This information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single key locator for all applications that need to use the same keys. In a Network Deployment environment, you also can specify the default binding information at the cell level.

This task provides instructions on how to configure server and cell-level key locators for a specific application using the WebSphere Application Server administrative console. You can configure binding information in the administrative console, but for extensions, you must use the Application Server Toolkit.

- Configure default key locators at the server level
  1. Connect to administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
  2. Click **Servers > Application Servers >***server1*.
  3. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.
  4. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:

a. Specify a name for the key locator in the **Key Locator Name** field.

b. Specify a name for the key locator class implementation in the **Key Locator Classname** field.

   WebSphere Application Server has the following default key locator class implementations:

   `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator`
   This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

   `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`
   This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice, O=IBM, c=US`.

c. Specify the password used to access the keystore password in the **Key Store Password** field.

   This field is optional is the key locator does not use a keystore.

d. Specify the path name used to access the keystore in the **Key Store Path** field.

   This field is optional is the key locator does not use a keystore. Use `${USER_INSTALL_ROOT}` as this path expands to the WebSphere Application Server path on your machine.

e. Select a keystore type from the **Key Store Type** field.

   This field is optional is the key locator does not use a keystore. Use the **JKS** option if you are not using Java Cryptography Extensions (JCE) and use **JCEKS** if you are using JCE.

- Configure default key locators at the cell level.
  1. Connect to administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
  2. Click **Security > Web Services > Key Locators**.
  3. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
     a. Specify a name for the key locator in the **Key Locator Name** field.
     b. Specify a name for the key locator class implementation in the **Key Locator Classname** field.

        WebSphere Application Server has the following default key locator class implementations:

        `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator`
        This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

        `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`
        This class, used by the response receiver, request sender, and request receiver, maps a

name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice, O=IBM, c=US`.

c. Specify the password used to access the keystore password in the **Key Store Password** field.

   This field is optional is the key locator does not use a keystore.

d. Specify the path name used to access the keystore in the **Key Store Path** field.

   This field is optional is the key locator does not use a keystore. Use `${USER_INSTALL_ROOT}` as this path expands to the WebSphere Application Server path on your machine.

e. Select a keystore type from the **Key Store Type** field.

   This field is optional is the key locator does not use a keystore. Use the **JKS** option if you are not using Java Cryptography Extensions (JCE) and use **JCEKS** if you are using JCE.

Trusted ID evaluator

Trusted ID evaluator (`com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator`) is a abstraction of the mechanism that evaluates whether the given ID name is trusted.
Depending upon the implementation, you can use various types of infrastructure to store a list of the trusted IDs, such as:

- Plain text file
- Database
- Lightweight Directory Access Protocol (LDAP) server

The trusted ID evaluator is typically used by the ultimate receiver in a multi-hop environment. The Web services security implementation invokes the trusted ID evaluator and passes the identity name of the intermediary as a parameter. If the identity is evaluated and deemed trustworthy, the procedure continues. Otherwise, an exception is thrown and the procedure is aborted.

Trusted ID evaluator collection:

Use this page to view a list of trusted identity (ID) evaluators. The trusted ID evaluator determines whether to trust the identity-asserting authority. Once the ID is trusted, the WebSphere Application Server issues the proper credentials, which are used in a downstream call for invoking resources. The trusted ID evaluator implements the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Services, click **Web Services: Default bindings for Web Services Security > Trusted ID Evaluators**.

Click **New** to create a trusted ID evaluator.

Click **Delete** to a delete a trusted ID evaluator.

Using this panel, complete the following steps:

1. Specify a trusted ID evaluator name and trusted ID evaluator class name.
2. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you return to the administrative console home panel.
3. Return to the Trusted ID Evaluator collection panel to update the Web services security run time with the default binding information, which is found in the `ws_security.xml` file.
4. Click **Update runtime**. The configuration changes made to the other Web services also are updated in the Web services security run time.

Trusted ID Evaluator Name:

Specifies the unique name of the trusted ID evaluator.

Trusted ID Evaluator Classname:

Specifies the class name of the trusted ID evaluator.

Trusted ID evaluator configuration settings:

Use this information to configure trust identity (ID) evaluators.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust ID Evaluators > New**.

You must specify the name and value properties for the default trusted ID evaluator to create the trusted ID list for evaluation.

Trusted ID Evaluator Name:

Specifies the unique name used by the application binding to refer to a trusted identity (ID) evaluator defined in the default binding.

You must specify the trusted ID evaluator name in the form, `trustedId_n`, where_`n` is an integer from 0 to n.

Trusted ID Evaluator Class Name:

Specifies the class name of the trusted ID evaluator.

Default	com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl

Login mappings

Login mappings, found in the `ibm-webservices-bnd.xmi` eXtended Markup Language (XML) file, contains a mapping configuration. This mapping configuration defines how the Web services security handler maps the token `<ValueType>` element, contained within the security token extracted from the message header, to the corresponding authentication method. The token `<ValueType>` element is contained within the security token extracted from a SOAP message header.
The sender-side Web services security handler generates and attaches security tokens based on `<AuthMethods>` element specified in the deployment descriptor. For example, if the authentication method is BasicAuth, the sender-side security handler generates and attaches `UsernameToken` (with both user name and password) to the SOAP message header. Web services security run time uses the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface as a security provider to generate security tokens on the client side (or when Web services is acting as client).

The sender security handler invokes the `handle()` method of a `javax.security.auth.callback.CallbackHandler` interface implementation. This implementation creates the security token and passes the token back to the sender security handler. The senders security handler constructs the security token based on the authentication information in the callback array. The security handler then inserts the security token into the Web Services Security message header.

The `CallbackHandlerinterface` implementation you use to generate the required security token is defined in the `<loginBinding>` element in the `ibm-webservicesclient-bnd.xmi` web services security binding file. For example,

```
<loginBinding xmi:id="LoginBinding_1052760331526" authMethod="BasicAuth"
      callbackHandler="com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
```

The `<loginBinding>` element associates the `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler` interface with the BasicAuth authentication method. WebSphere Application Server provides the following set of CallbackHandler interface implementations you can use to create various security token types:

`com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`
If there is no basic authentication data defined in the login binding information (this is not the same as the HTTP basic authentication information), the previous token type prompts for user name and password through a GUI login panel. It uses the basic authentication data defined in the login binding if it is defined. Use this CallbackHandler with the BasicAuth authentication method.

Attention: Do not use this CallbackHandler on the server because it prompts you for login binding information.

`com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`
If basic authentication data is not defined in the login binding (this is not the same as the HTTP basic authentication information), it prompts for the user name and password using standard in (stdin). It uses the basic authentication data defined in the login binding if it is defined. Use this CallbackHandler with BasicAuth authentication method.

Attention: Do not use this CallbackHandler on the server because it prompts you for login binding information.

`com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`
This CallbackHandler does not prompt. Rather, it uses the basic authentication data defined in the login binding (this is not the same as the HTTP basic authentication information). This CallbackHandler is meant to be used with BasicAuth authentication method. This can be used when Web services is running as a client and needs to send basic authentication (`<wsse:UsernameToken>`) to the downstream call.

Attention: You must define the basic authentication data in the login binding information for this CallbackHandler.

`com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`
The CallbackHandler generates Lightweight Third Party Authentication (LTPA) tokens from the run as `JAASSubject` (invocation Subject) of the current WebSphere Application Server security context. However, if there is basic authentication data defined in the login binding information (this is not the same as the HTTP basic authentication information), it uses the basic authentication data and LTPA token generated. The Web services security run time inserts the LTPA token as a binary security token (`<wsse:BinarySecurityToken>`) into the message SOAP header. The value type is mandatory. (See LTPA for more information). Use this CallbackHandler with the LTPA authentication method.

Attention: The **Token Type URI** and **Token Type Local Name** must be defined in the login binding information for this CallbackHandler. The token value type is used to validate the token to the request sender and request receiver binding configuration.

Figure.1 shows the sender security handler in the request sender message process.

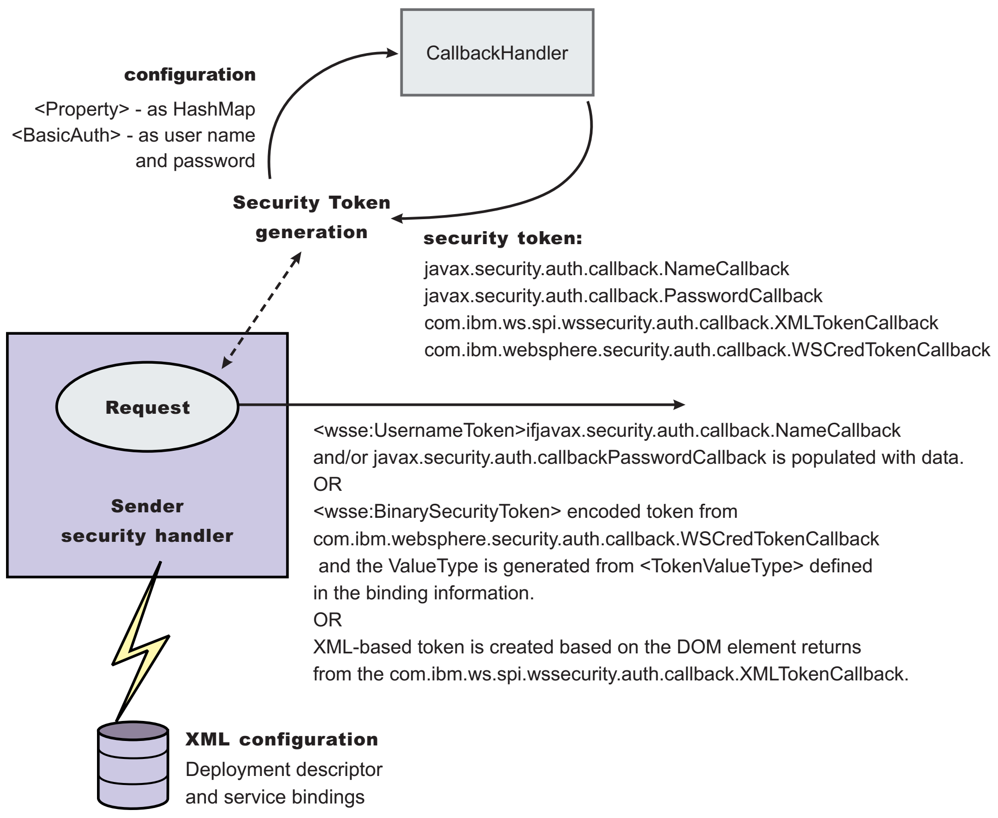

CallbackHandler
configuration
<Property> - as HashMap
<BasicAuth> - as user name and password
Security Token generation
security token:
javax.security.auth.callback.NameCallback
javax.security.auth.callback.PasswordCallback
com.ibm.ws.spi.wssecurity.auth.callback.XMLTokenCallback
com.ibm.websphere.security.auth.callback.WSCredTokenCallback
Request
Sender security handler
<wsse:UsernameToken>ifjavax.security.auth.callback.NameCallback and/or javax.security.auth.callbackPasswordCallback is populated with data.
OR
<wsse:BinarySecurityToken> encoded token from com.ibm.websphere.security.auth.callback.WSCredTokenCallback and the ValueType is generated from <TokenValueType> defined in the binding information.
OR
XML-based token is created based on the DOM element returns from the com.ibm.ws.spi.wssecurity.auth.callback.XMLTokenCallback.
XML configuration
Deployment descriptor
and service bindings

Figure 14. Request sender SOAP message process

The receiver-side security server can be configured to support multiple authentication methods and multiple types of security tokens. Upon receiving a message, the receiver Web services security handler compares the token type (in the message header) with the expected token types configured in the deployment descriptor. The Web services security handler extracts the security token form the message header and maps the token `<ValueType>` element to the corresponding authentication method. The mapping configuration is defined in the `<loginMappings>` element in the `ibm-webservices-bnd.xmi` XML file. For example:

```
<loginMappings xmi:id="LoginMapping_1051977980074" authMethod="LTPA"
      configName="WSLogin">
     <callbackHandlerFactory xmi:id="CallbackHandlerFactory_1051977980081"
     classname="com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl"/>
      <tokenValueType xmi:id="TokenValueType_1051977980081"
      uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2" localName="LTPA"/>
</loginMappings>
```

The `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface is a factory for `javax.security.auth.callback.CallbackHandler`. The Web services security run time initiates the factory implementation class and passes the authentication information from Web services security header to the factory class through the set methods. The Web services security run time then invokes the `newCallbackHandler()` method to obtain the `javax.security.auth.CallbackHandler` object, which generates the required security token). When the security handler receives an LTPA `BinarySecurityToken`, it uses the `WSLogin` JAAS login configuration and the `newCallbackHandler()` method to validate the security token. If none of the expected token types are found in the SOAP message Web services security header, the request is rejected with an SOAP fault. Otherwise, the token type is used to map to a JAAS login configuration for token validation. If authentication is successful, a JAAS Subject is created and associated with the thread of execution. Otherwise, the request is rejected with an SOAP fault.

Figure.2 shows the receiver security handler in the request receiver message process.

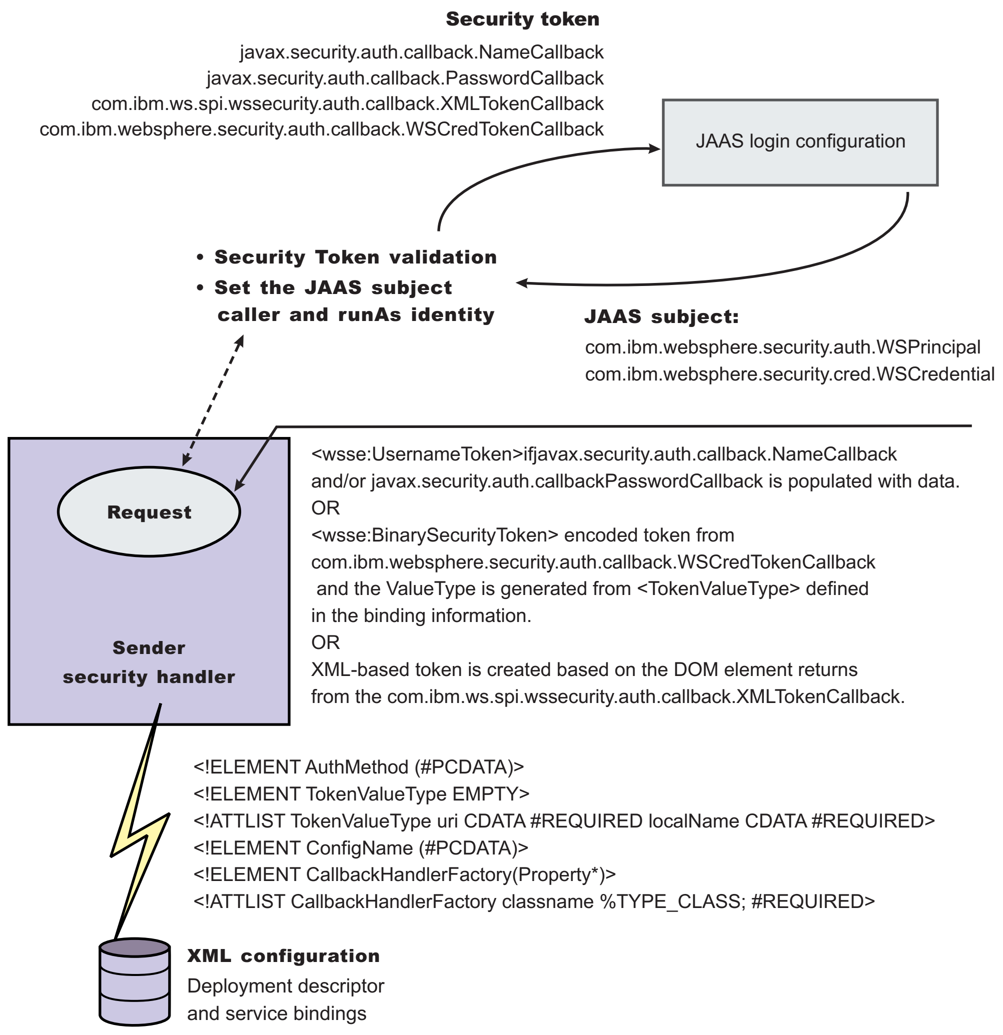

Security token
javax.security.auth.callback.NameCallback
javax.security.auth.callback.PasswordCallback
com.ibm.ws.spi.wssecurity.auth.callback.XMLTokenCallback
com.ibm.websphere.security.auth.callback.WSCredTokenCallback
JAAS login configuration
• Security Token validation
• Set the JAAS subject caller and runAs identity
JAAS subject:
com.ibm.websphere.security.auth.WSPrincipal
com.ibm.websphere.security.cred.WSCredential
Request
Sender security handler
<wsse:UsernameToken>ifjavax.security.auth.callback.NameCallback and/or javax.security.auth.callbackPasswordCallback is populated with data.
OR
<wsse:BinarySecurityToken> encoded token from com.ibm.websphere.security.auth.callback.WSCredTokenCallback and the ValueType is generated from <TokenValueType> defined in the binding information.
OR
XML-based token is created based on the DOM element returns from the com.ibm.ws.spi.wssecurity.auth.callback.XMLTokenCallback.
<!ELEMENT AuthMethod (#PCDATA)>
<!ELEMENT TokenValueType EMPTY>
<!ATTLIST TokenValueType uri CDATA #REQUIRED localName CDATA #REQUIRED>
<!ELEMENT ConfigName (#PCDATA)>
<!ELEMENT CallbackHandlerFactory(Property*)>
<!ATTLIST CallbackHandlerFactory classname %TYPE_CLASS; #REQUIRED>
XML configuration
Deployment descriptor and service bindings

Figure 15. Request receiver SOAP message process

The following table shows the authentication methods and JAAS login configurations.

Authentication method	JAAS login configuration
BasicAuth	`WSLogin`
Signature	`system.wssecurity.Signature`
LTPA	`WSLogin`
IDAssertion	`system.wssecurity.IDAssertion`

The default `<LoginMapping>` is defined in the cell-level `ws-security.xml` and server-level `ws-security.xml` files. If nothing is defined in the binding file information, the `ws-security.xml` default is used. However, an administrator can override the default by defining a new `<LoginMapping>` element in the binding file.

The client reads the default binding information in the `${WAS_HOME}/properties/Web Services Security.xml` file. The server run-time component loads the cell-level *Web Services Security.xml* and server-level *Web Services Security.xml* files if they exist.

The two files are merged in the run time to form one effective set of default binding information. On a base application server the server run time component only loads the server-level *Web Services Security.xml* file. The server-side *Web Services Security.xml* file and application Web services security binding information are managed by the administrative console and also by WSADMIN. You can specify the binding information during application deployment either through the administrative console or WSADMIN. The Web services security policy is defined in the deployment descriptor extension (`ibm-webservicesclient-ext.xmi`) and the bindings are stored in the IBM binding extension (`ibm-webservicesclient-bnd.xmi`). However, `${WAS_HOME}/properties/Web Services Security.xml` defines the default binding value for the client. If the binding information is not specified in the binding file, the run time reads the binding information from the default `${WAS_HOME}/properties/Web Services Security.xml` file.

Login mappings collection:

Use this page to view a list of configurations for validating security tokens within incoming messages. Login mappings map an authentication method to a Java Authentication and Authorization Service (JAAS) login configuration to validate the security token. Four authentication methods are predefined in the WebSphere Application Server: BasicAuth, Signature, IDAssertion, and LTPA.

To view this administrative console page, complete the following steps:

1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

Click **New** to create a login mapping.

Click **Delete** to delete a login mapping.

If you click **Update runtime**, the Web services security run time is updated with the default binding information, which is contained in the `ws-security.xml` file that was previously saved. After you specify the authentication method, the Java Authentication and Authorization Service (JAAS) configuration name, and the Callback Handler Factory class name on this panel, you must complete the following steps:

Click **Save** at the top of the administrative console. When you click **Save**, you return to the administrative console home panel.

Return to the Login Mappings collection panel and click **Update runtime**.

Important: When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.

Authentication Method:

Specifies the authentication method used for validating the security tokens.

The following authentication methods are available:

BasicAuth
Basic authentication includes both a user name and password in the security token. The information in the token is authenticated by the receiving server and used to create a credential.

Signature
When the authentication method is signature, an X.509 certificate is sent as a security token. For Lightweight Directory Access Protocol (LDAP) registries, the distinguished name (DN) is mapped

to a credential, which is based on the LDAP certificate filter settings. For local OS registries, the first attribute of the certificate, usually the common name (CN) is mapped directly to a user ID in the registry.

IDAssertion
Identity assertion maps a trusted identity (ID) to a WebSphere credential. This authentication method only includes a user name in the security token. An additional token is included in the message for trust purposes. Once the additional token is trusted, the IDAssertion token user name is mapped to a credential.

LTPA Lightweight Third Party Authentication (LTPA) validates an LTPA token.

JAAS Configuration Name:

Specifies the name of the Java Authentication and Authorization Service (JAAS) configuration.

Callback Handler Factory Class Name:

Specifies the name of the factory for the CallbackHandler class.

Login mapping configuration settings:

Use this page to specify the Java Authentication and Authorization Service (JAAS) login configuration settings used to validate security tokens within incoming messages.

To view this administrative console page, complete the following steps:

1. Click **Servers** > **Application Servers** > *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings > New**.

Authentication Method:

Specifies the method of authentication.

You can use any string, but the string must match the element in the service-level configuration. The following words are reserved and have special meanings:

BasicAuth
Uses both a user name and a password.

IDAssertion
Uses only a user name, but requires that additional trust is established on the receiving server using a TrustedIDEvaluator mechanism.

Signature
Uses the distinguished name (DN) of the signer.

LTPA Validates a token.

JAAS Configuration Name:

Specifies the name of the Java Authentication and Authorization Service (JAAS) configuration.

Specify your JAAS configurations using the administrative console by clicking **Security > JAAS Configuration > Application**.

Callback Handler Factory Class Name:

Specifies the name of the factory for the CallbackHandler class.

You must implement the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory class in this field.

Default:	com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory

Token Type URI:

Specifies the namespace Uniform Resource Identifiers (URI), which denotes the type of security token accepted.

If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType element identifies the type of security token and its namespace. If Extensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.

If the reserved words are specified previously in the Authentication Method field, this field is ignored.

Data type:	Unicode characters except for non-ASCII characters, but including the number sign (#), the percent sign (%), and the square brackets ([ ]).

Token Type Local Name:

Specifies the local name of the security token type, for example, X509v3.

If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType attribute identifies the type of security token and its namespace. If Extensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.

If the reserved words are specified previously in the Authentication Method field, this field is ignored.

Nonce Maximum Age:

Specifies the time, in seconds, before the nonce time stamp expires. Nonce is a randomly generated value.

You must specify a minimum of 300 seconds for the Nonce Maximum Age field. However, the maximum value cannot exceed the number of seconds specified in the Nonce Cache Timeout field for either the server level or the cell level. You can specify the **Nonce Maximum Age** value for the server level:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**.
3. Specify the **Nonce Maximum Age** value for the cell level by clicking **Security > Web Services > Properties**.

Important: The Nonce Maximum Age field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value: `Nonce is not supported for authentication methods other than BasicAuth`.

If you specify BasicAuth, but do not specify values for the Nonce Maximum Age field, the Web services security run time searches for a Nonce Maximum Age value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 300 seconds.

Default	300 seconds

Range	300 to Nonce Cache Timeout seconds

Nonce Clock Skew:

Specifies the clock skew value, in seconds, to consider when WebSphere Application Server checks the freshness of the message. Nonce is a randomly generated value.

You must specify a minimum of 0 seconds for the Nonce Clock Skew field. However, the maximum value cannot exceed the number of seconds specified in the Nonce Maximum Age on this Login Mappings panel.

Important: The Nonce Clock Skew field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value: `Nonce is not supported for authentication methods other than BasicAuth`.

Note: If you specify BasicAuth, but do not specify values for the Nonce Clock Skew field, the Web services security run time searches for a Nonce Clock Skew value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 0 seconds.

Default	0 seconds
Range	0 to Nonce Maximum Age seconds

Configuring the client for request signing: digitally signing message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the WebSphere Application Server Toolkit.

- Configuring the client security bindings using the WebSphere Application Server Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the WebSphere Application Server Toolkit.
6. Expand **Request Sender Configuration > Integrity**. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see XML digital signature.

7. Select the parts of the message in which to sign by clicking **Add** and selecting one of the following three parts: body, timestamp, or SecurityToken The following is a list and description of the message parts

   Body This is the user data portion of the message.

   Timestamp
   The timestamp determines if the message is valid based on the time the message was sent and then received. If **timestamp** is selected, proceed to the next step to **Add Created Time Stamp** to the message.

   Securitytoken
   The security token authenticates the client. If **securitytoken** is selected, the message is signed.

   You can choose to digitally sign the message using a time stamp if **Add Created Time Stamp** is selected and configured. You can digitally sign the message using a security token if a login configuration authentication method is selected.
8. Optional: Expand the **Add Created Time Stamp** section and select this option if you want a time stamp added to the message You can specify an expiration time for the time stamp, which helps defend against replay attacks.

   The lexical representation for duration is the [ISO 8601] extended format *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years
   - *nM* represents the number of months
   - *nD* represents the number of days
   - *T* is the date and time separator
   - *nH* represents the number of hours
   - *nM* represents the number of minutes
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, the format is: `P1Y2M3DT10H30M`. Typically, configure a message time stamp for about 10 to 30 minutes, which is `P0Y0M0DT0H10M0S`.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:
- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor**field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:
- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts to digitally sign when the client sends a message to a server.

Once you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message. See Configuring the client for request signing: choosing the digital signature method for more information.

Configuring the client for request signing: Choosing the digital signature method

Prior to completing these steps, read either of the following topics to become familiar with the Security Extensions tab and the Port Binding tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the WebSphere Application Server Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and the Web services security bindings, respectively. You must specify which parts of the message sent by the client must be digitally signed. See Configuring the client for request signing: Digitally signing message parts for more information.

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

1. Launch the Assembly Toolkit and click **Windows > Open Perspective > J2EE**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the `META-INF` directory for an EJB module or the `WEB-INF` directory for a Web module.
4. Right-click the **webservicesclient.xml** file and click **Open With > Web Services Client Editor**.
5. Click the **Port Binding** tab.
6. Expand **Security Request Sender Binding Configuration > Signing Information**.
7. Select **Edit** to view the signing information and select a digital signature method from the **Signature method algorithm** field. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following Web site http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the `<SignedInfo>` element before the information is digested as part of the signature operation.
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. Signing the `<DigestValue>` element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signature method algorithm	Converts the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

Name	Purpose
Signing key name	Represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is found in the key store and is used to sign the request.
Signing key locator	Represents a reference to a key locator implementation class that locates the correct hey store where the alias and the certificate exist.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method is used to digitally sign a message when the client sends a message to a server.

After you configure the client to digitally sign the message, you must configure the server to verify the digital signature. See Configuring the server for request digital signature verification: Verifying the message parts for more information.

Configuring the server for request digital signature verification: verifying the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which parts of the message sent by the client must be digitally signed. See Configuring the client for request signing: digitally signing message parts to determine which message parts are digitally signed. The message parts specified for the client request sender must match the message parts specified for the server request receiver.

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which parts of the request to verify.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Security Extensions** tab in the Web Services Editor.
6. Expand the **Request Receiver Service Configuration Details > Required Integrity** section. Required integrity refers to the parts of the message that require digital signature verification. The purpose of digital signature verification is to make sure that the message parts have not been modified while it was transmitted across the Internet.
7. Select the parts of the message to verify by clicking **Add** and selecting one of the following three parts: body, timestamp, or SecurityToken. You can determine which parts of the message to verify by looking at the Web Service Request Sender Configuration in the client application. To view the Web Service Request Sender Configuration information in the Web Services Client Editor, click the **Security Extensions** tab and expand **Request Sender Configuration > Integrity**. The following information is a list and description of the message parts:

   Body This is the user data portion of the message.

   Timestamp
   The timestamp determines if the message is valid based on the time the message is sent and then received. If **timestamp** is selected, proceed to the next step to **Add Created Time Stamp** to the message.

   Securitytoken
   The security token authenticates the client. If **securitytoken** is selected, the message is signed.
8. Optional: Expand the **Add Received Time Stamp** section. The Add Received Time Stamp indicates to validate the Add Created Time Stamp configured by the client. You must select option this if you selected Add Created Time Stamp on the client. The time stamp ensures message integrity by indicating the freshness of the request. This option helps to defend against replay attacks.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the server when the client sends a message to a server.

After you have specified what message parts contain a digital signature that must be verified by the server, you must configure the server to recognize the digital signature method used to digitally sign the message. See Configuring the server for request digital signature verification: choosing the verification method for more information.

Configuring the server for request digital signature verification: choosing the verification method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which message parts contain digital signature information that must be verified by the server. See Configuring the server for request digital signature verification: verifying the message parts to specify which message parts are digitally signed by the client and must be verified by the server. The message parts specified for the client request sender must match the message parts specified for the server request receiver. Likewise, the digital signature method chosen for the client must match the digital signature method used by the server.

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the server will use during verification.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Binding Configurations** tab.
6. Expand the **Security Request Receiver Binding Configuration Details > Signing Information** section.
7. Click **Edit** to edit the signing information. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**

- **Use certificate path reference**
- **Trust anchor reference**
- **Certificate store reference**
- **Trust any certificate**

For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which can be found at: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the `<DigestValue>` binds resource content to the signer key. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is sent with the message. This public key or certificate might not be validated at the receiving end. By selecting **User certificate path reference**, you must configure a trust anchor reference and a certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a key store that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application. See

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.

- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:
- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method the server uses to verify the digital signature in the message parts.

After you configure the client for request signing and the server for request digital signature verification, you must configure the server and the client to handle the response. Next, specify the response signing for the server. See Configuring the server for response signing: digitally signing message parts for more information.

Configuring the server for response signing: digitally signing message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:
- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which message parts to digitally sign when configuring the server for response signing:
1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the WebSphere Application Server Toolkit.
6. Expand **Response Sender Service Configuration Details > Integrity**. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see XML digital signature.

7. Select the parts of the message in which to sign by clicking **Add** and selecting **Body**, **Timestamp**, or **Securitytoken**.

   Body The body is the user data portion of the message.

   Timestamp
   The timestamp determines if the message is valid based on the time that the message is sent and then received. If **timestamp** if selected, proceed to the next step to **Add Created Time Stamp** to the message.

   Securitytoken
   If security token is selected, the authentication information is added to the message.
8. Optional: Expand the **Add Created Time Stamp** section. Select this option if you want to add a time stamp to the message. Also, you can specify an expiration time for the time stamp, which helps defend against replay attacks. The lexical representation for duration is the ISO 8601 extended format, *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years.
   - *nM* represents the number of months.
   - *nD* represents the number of days.
   - *T* is the date and time separator.
   - *nH* represents the number of hours.
   - *nM* represents the number of minutes.
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, if you want to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, write: `P1Y2M3DT10H30M`. Typically, a message time stamp is configured for about 10 to 30 minutes. 10 minutes is represented as: `P0Y0M0DT0H10M0S`.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:
- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:
- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts to digitally sign when the server sends a response to the client.

Once you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message. See Configuring the server for response signing: choosing the digital signature method for more information.

Configuring the server for response signing: choosing the digital signature method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which digital signature method to use when configuring the server for response signing:

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Binding Configurations** tab.
6. Expand **Response Sender Binding Configuration Details > Signing Information**.
7. Click **Edit** to choose a signing method. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Signing key name**
   - **Signing key locator**

   The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation. The same algorithm used here should also be used on the client response receiver. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

Name	Purpose
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signing key name	The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.
Signing key locator	The signing key locator represents a reference to a key locator implementation. For more information on configuring key locators, see any of the following files: • Configuring key locators using the WebSphere Application Server Toolkit • Configuring key locators using the administrative console • Configuring server and cell level key locators using the administrative console

The **Signing key name** refers to a key entry associated with the signing key locator. The key entry has an alias, which is found in the keystore or wherever the certificates are stored based upon the key locator implementation. The **Signing key locator** references the implementation class that locates the correct key store where the alias and certificate exists.

You have specified which method is used to digitally sign a message when the server sends a message to a client.

Once you have configured the server to digitally sign the response message, you must configure the client to verify the digital signature contained in the response message. See Configuring the client for response digital signature verification: verifying the message parts for more information.

Configuring the client for response digital signature verification: verifying the message parts

Prior to completing these steps, read either of the following topics to becomes familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Servers Client Editor within the WebSphere Application Server Toolkit:

- Configuring the client security bindings using the WebSphere Application Server Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which parts of the response to verify.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Security Extensions** tab.
6. Expand the **Response Receiver Configuration > Required Integrity** section. Required Integrity refers to parts that require digital signature verification. Digital signature verification decreases the risk that the message parts have been modified while the message is transmitted across the Internet.
7. Select the parts of the message that must be verified. You can determine which parts of the message to select by looking at the Web service response sender configuration. To add parts of the message, click **Add** and select one of the following three parts:

   Body This is the user data portion of the message.

   Timestamp
   The time stamp determines if the message is valid based on the time the message was sent and then received. If timestamp is selected, proceed to the next step to **Add Received Time Stamp** to the message.

   Securitytoken
   The security token authenticates the client. If **Securitytoken** is selected, the message is signed.
8. Optional: Expand the **Add Received Time Stamp** section. Select **Add Received Time Stamp** to add the received time stamp to the message.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the client when the server sends a response message to the client.

After you have specified which message parts contain a digital signature that must be verified by the client, you must configure the client to recognize the digital signature method used to digitally sign the message. See Configuring the client for response digital signature verification: choosing the verification method for more information.

Configuring the client for response digital signature verification: choosing the verification method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Editor within the Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which message parts contain digital signature information that must be verified by the client. See Configuring the client for response digital signature verification: verifying the message parts to specify which message parts are digitally signed by the server and must be verified by the client. The message parts specified for the server response sender must match the message parts specified for the client response receiver. Likewise, the digital signature method chosen for the server must match the digital signature method used by the client.

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the client will use during verification.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Port Binding** tab.
6. Expand the **Security Response Receiver Binding Configuration > Signing Information** section.
7. Click **Edit** to select a digital signature method. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Signing key name**
   - **Signing key locator**

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which can be found at: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation.

Name	Purpose
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the `<DigestValue>` binds resource content to the signer key. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is transmitted with the message. To validate this public key at the receiving end, configure a certificate path reference. By selecting **User certificate path reference,** you must configure a trust anchor reference and certificate store reference to validate the certificate sent with the message. By selecting **trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a keystore that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method the client uses to verify the digital signature in the message parts.

After you configure the server for response signing and the client for request digital signature verification, verify that you have configured the client and the server to handle the message request.

Configuring the client security bindings using the Assembly Toolkit

When configuring a client for Web services security, the bindings describe how to execute the security specifications found in the extensions. Use the Web Services Client Editor within the Assembly Toolkit to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a Web service or from a Web service accessing a downstream Web service. This document focuses on the pure client situation. However, the concepts, and in most cases the steps, also apply when a Web service is configured to communicate downstream to another Web service that has client bindings. Complete the following steps to edit the security bindings on a pure client (or server acting as a client) using the Assembly Toolkit:

1. Import the Web services client EAR file into the Assembly Toolkit. When you edit the client bindings on a server acting as a client, the same basic steps apply. Complete the following steps to import your client EAR file into the Assembly Toolkit. Refer to the Assembly Toolkit documentation for additional information.
   a. Download and install the Assembly Toolkit. You can download the Assembly Toolkit from the following Web site:

      http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP &q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en
   b. Start the Assembly Toolkit and open the Java Perspective by selecting **Window > Open Perspective > J2EE**.
   c. Import the client EAR file by selecting **File > Import > EAR file**.
   d. Click **Next**.
   e. Enter the path name to the EAR file in the **EAR File** field or click **Browse** to locate the file.
   f. Enter the project name in the **Project name** field.
   g. Click **Finish**.
2. Open the Web Services Client Editor within the Assembly Toolkit to begin editing the client bindings. To access the client bindings using the Assembly Toolkit, complete the following steps:
   a. Open the Navigator by clicking **Window > Show View > Navigator**.
   b. Expand your application Java archive (JAR) from the Navigator.
   c. Expand the J2EE client application (appClientModule, ejbModule, or WebContent), which should be part of the client JAR package that you selected.
   d. Expand the **META-INF** directory and locate the `webservicesclient.xml` file.
   e. Right-click the `webservicesclient.xml` file and click **Open With > Web Services Client Editor**. In the Web Services Client Editor (for `webservicesclient.xml` and outbound requests and inbound responses Web services configuration), there are several tabs at the bottom of the editor including **Service References**, **Handlers**, **Security Extensions**, **Web Services Client Binding**, and **Port Binding**. The security extensions are edited using the **Security Extensions** tab. The security bindings are edited using the **Port Binding** tab.

3. On the **Security Extensions** tab, select the port qualified name bindings that you want to configure. The Web services security extensions are configured for outbound requests and inbound responses. You need to configure the following information for Web services security extensions. These topics are discussed in more detail in other sections of the documentation.

   Request sender configuration details

   Details
   Configuring the client for request signing: digitally signing message parts

   Integrity
   Configuring the client for request signing: digitally signing message parts

   Confidentiality
   Configuring the client for request encryption: encrypting the message parts

   Login Config

   BasicAuth
   Configuring the client for basicauth authentication: specifying the method

   IDAssertion
   Configuring the client for identity assertion authentication: specifying the method

   Signature
   Configuring the client for signature authentication: specifying the method

   LTPA Configuring the client for LTPA token authentication: specifying LTPA token authentication

   ID Assertion
   Configuring the client for identity assertion authentication: specifying the method

   Add Created Time Stamp
   Configuring the client for request signing: digitally signing message parts

   Response receiver configuration details

   Required Integrity
   Configuring the client for response digital signature verification: verifying the message parts

   Required Confidentiality
   Configuring the client for response decryption: decrypting message parts

   Add Received Time Stamp
   Configuring the client for response digital signature verification: verifying the message parts

4. From the **Port Binding** tab, select the port qualified name binding that you want to configure. The Web services security bindings are configured for outbound requests and inbound responses. You need to configure the following information for Web services security bindings. These topics are discussed in more details in other sections of the documentation.

   Security request sender binding configuration

   Signing information
   Configuring the client for request signing: choosing the digital signature method

   Encryption information
   Configuring the client for request encryption: choosing the encryption method

   Key locators
   Configuring key locators using the Assembly Toolkit

   Login binding

Basic auth
: Configuring the client for basicauth authentication: collecting the authentication information

ID assertion
: Configuring the client for identity assertion: Collecting the authentication method

Signature
: Configuring the client for signature authentication: collecting the authentication information

LTPA
: Configuring the client for LTPA token authentication: Collecting the authentication method information

Security response receiver binding configuration

Signing information
: Configuring the client for response digital signature verification: choosing the verification method

Encryption information
: Configuring the client for response decryption: choosing a decryption method

Trust anchor
: Configuring trust anchors using the Assembly Toolkit

Certificate store list
: Configuring the client-side collection certificate store using the Application Server Toolkit

Key locators
: Configuring key locators using the Assembly Toolkit

Important: When configuring the Security Request Sender Binding Configuration, you must synchronize the information used to perform the specified security with the Security Request Receiver Binding Configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects as there is no negotiation during run time to figure out the requirements of the server. For example, when configuring the encryption information in the Security Request Sender Binding Configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This is an example of how the client and server configuration are tightly coupled. Additionally, when configuring the Security Response Receiver Binding Configuration, the server must send the response using security information known by this client Security Response Receiver Binding Configuration.

The following table shows the related configurations between the client and server. The client request sender and server request receiver are relative configurations that must be synchronized with each other. The server response sender and client response receiver are related configurations that must be synchronized with each other. Note that related configurations are end points for any request or response. One end point must communicate its actions with the other end point because they do not negotiate any run time requirements.

Table 4. Related configurations

Client configuration	Server configuration
Request sender	Request receiver
Response receiver	Response sender

Configuring the security bindings on a server acting as a client using the administrative console

When configuring a client for Web services security, the bindings describe how to execute the security specifications found in the extensions. Use the Web Services Client Editor within the WebSphere Application Server Toolkit to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a Web service or from a Web service accessing a downstream Web service. Complete the following steps to find the location in which to edit the client bindings from a Web service that is running on the server. When a Web service communicates with another Web service, you must configure client bindings to access the downstream Web service.

1. Deploy the Web service using the WebSphere Application Server administrative console by clicking **Applications > Install New Application**. You can access the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number. For more information on installing an application, see Installing a new application
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending upon which type of service is the client to the downstream service.
   - For Web Modules, click the Web Archive (WAR) file that you configured as the client.
   - For EJB Modules, click the Java Archive (JAR) file that you configured as the client.
4. Click the name of the WAR or JAR file.
5. Under Additional Properties, click **Web Services: Client Security Bindings**. A table displays with the following columns:
   - **Component Name**
   - **Port**
   - **Web Service**
   - **Request Sender Binding**
   - **Request Receiver Binding**
   - **HTTP Basic Authentication**
   - **HTTP SSL Configuration**

   For Web services security, you must edit the Request Sender Binding and Response Receiver Binding configurations. You can use the defaults for some of the information at the server level (and at the cell level for some information). Default bindings are convenient because you can configure commonly reused elements such as key locators once and then reference their aliases in the application bindings.
6. View the default bindings for the server using the Administrative Console by clicking **Servers > Application Server >** *server1*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**. You can configure the following sections. These topics are discussed in more detail in other sections of the documentation.
   - Request sender binding
     – Signing information
     – Encryption information
     – Key locators
     – Login bindings
   - Response receiver binding
     – Signing information
     – Encryption information
     – Trust anchors
     – Collection certificate store

– Key locators

Important: When configuring the Security Request Sender Binding Configuration, you must synchronize the information used to perform the specified security with the Security Request Receiver Binding Configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects as there is no negotiation during run time to figure out the requirements of the server. For example, when configuring the encryption information in the Security Request Sender Binding Configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This is an example of how the client and server configuration are tightly coupled. Additionally, when configuring the Security Response Receiver Binding Configuration, the server must send the response using security information known by this client Security Response Receiver Binding Configuration.

The following table shows the related configurations between the client and server. The client request sender and server request receiver are relative configurations that must be synchronized with each other. The server response sender and client response receiver are related configurations that must be synchronized with each other. Note that related configurations are end points for any request or response. One end point must communicate its actions with the other end point because they do not negotiate any run time requirements.

Table 5. Related configurations

Client configuration	Server configuration
Request sender	Request receiver
Response receiver	Response sender

Configuring the server security bindings using the Assembly Toolkit

Create an enterprise JavaBean (EJB) file Java archive (JAR) file or Web archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the WSDL2Java command. You can edit these files using the Web Services Editor in the Assembly Toolkit.

When configuring server-side security for Web services security, the security extensions configuration specifies what security is to be performed while the security bindings configuration indicates how to perform what is specified in the security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference them from the WAR and JAR binding configurations.

Prior to importing the Web services enterprise archive (EAR) file into the Assembly Toolkit, make sure that you have already run the `wsdl2java` command on your Web service to enable your J2EE application. You must import the Web services enterprise archive (EAR) file into the Assembly Toolkit. Complete the following steps to import your EAR file into the Assembly Toolkit:

1. Download, install, and launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Import the application EAR file by clicking **File > Import > EAR file**.
4. Click **Next** and indicate both the EAR file name in the **EAR File** field and the project name in the **Project name** field.
5. Click **Finish**.

Refer to Assembly Toolkit documentation for more information.

Open the Web Services Editor in the Assembly Toolkit to begin editing the server security extensions and bindings. The following steps will help you locate the server security extensions and bindings. Other tasks specify how to configure each section of the extensions and bindings in more detail.

1. Expand your application module from the Navigator. If the Navigator is not shown, you can open it by clicking **Window > Show View > Navigator**.
2. If your application is a Web application (WAR) file, the following steps apply:
   a. Expand the **WebContent > WEB-INF** section.
   b. Locate the **webservices.xml** file. The **webservices.xml** file represents the server-side (inbound) Web services configuration. The **webservicesclient.xml** file represents the client-side (outbound) Web services configuration.
      1) To configure the server for inbound requests and outbound responses security configuration, right-click the **webservices.xml** file and click **Open With > Web Services Editor**.
      2) To configure the client for outbound requests and inbound responses security configuration, right-click the **webservicesclient.xml** file and click **Open With > Web Services Client Editor**. For more information, see Configuring the client security bindings using the Assembly Toolkit.
3. If your application is an EJB Application (JAR), the following steps apply:
   a. Expand the **ejbModule > META-INF** section.
   b. Locate the **webservices.xml** file. The **webservices.xml** file represents the server-side (inbound) Web services configuration. The **webservicesclient.xml** file represents the client-side (outbound) Web services configuration.
      1) To configure the server for inbound requests and outbound responses security configuration, right-click the **webservices.xml** file and click **Open With > Web Services Editor**.
      2) To configure the client for outbound requests and inbound responses security configuration, right-click the **webservicesclient.xml** file and click **Open With > Web Services Client Editor**. For more information, see Configuring the client security bindings using the Assembly Toolkit.
4. In the Web Services Editor (for **webservices.xml** and inbound requests and outbound responses Web services configuration), there are several tabs at the bottom of the editor including **Web Services**, **Port Components**, **Handlers**, **Security Extensions**, **Bindings**, and **Binding Configurations**. The security extensions are edited using the **Security Extensions** tab. The security bindings are edited using the **Security Bindings** tab.
   a. On the **Security Extensions** tab, select the port component binding that you want to edit. The Web services security extensions are configured for inbound requests and outbound responses. You need to configure the following information for Web services security extensions. These topics are discussed in more detail in other sections of the documentation.

      Request receiver service configuration details

      Required integrity
         Configuring the server for request digital signature verification: verifying the message parts

      Required confidentiality
         Configuring the server for request decryption: decrypting message parts

      Login config

         Basic auth
            Configuring the server to handle basicauth authentication

         ID assertion
            Configuring the server to handle identity assertion authentication

         Signature
            Configuring the server to handle signature authentication

         LTPA Configuring the server to handle LTPA token authentication

Add received time stamp
Configuring the server for request digital signature verification: verifying the message parts

Response sender service configuration details

Details
Configuring the server for response signing: digitally signing message parts

Integrity
Configuring the server for response signing: digitally signing message parts

Confidentiality
Configuring the server for response encryption: encrypting message parts

Add created time stamp-
Configuring the server for response signing: digitally signing message parts

b. On the **Binding Configurations** tab, select the port component binding that you want to edit. The Web services security bindings are configured for inbound requests and outbound responses. You need to configure the following information for Web services security bindings. These topics are discussed in more details in other sections of the documentation.

Response receiver binding configuration details

Signing Information
Configuring the server for request digital signature verification: choosing the verification method

Encryption Information
Configuring the server for request decryption: choosing the decryption method

Trust Anchor
Configuring trust anchors using the Assembly Toolkit

Certificate Store List
Configuring the server-side collection certificate store using the Assembly Toolkit

Key Locators
Configuring key locators using the Assembly Toolkit

Login Mapping

Basic auth
Configuring the server to validate basicauth authentication information

ID assertion
Configuring the server to validate identity assertion authentication information

Signature
Configuring the server to validate signature authentication information

LTPA Configuring the server to validate LTPA token authentication information

Trusted ID Evaluator

Trusted ID Evaluator Reference

Response sender binding configuration details

Signing information
Configuring the server for response signing: choosing the digital signature method

Encryption information
Configuring the server for response encryption: choosing the encryption method

Key Locators
Configuring key locators using the Assembly Toolkit

Configuring the server security bindings using the Administrative Console

Create an enterprise JavaBean (EJB) file Java archive (JAR) file or Web archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the WSDL2Java command. You can edit these files using the Web Services Editor in the Assembly Toolkit.

When configuring server-side security for Web services security, the security extensions configuration specifies what security is to be performed while the security bindings configuration indicates how to perform what is specified in the security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference them from the WAR and JAR binding configurations.

The following steps describe how to edit bindings for a Web service after they are deployed on a server. When one Web service communicates with another Web service, you also must configure the client bindings to access the downstream Web service.

1. Deploy the Web service using the WebSphere Application Server Administrative Console. The Administrative Console is accessible by typing `http://localhost:9090/admin` in a Web browser. Once you have logged into the Administration Console, click **Applications > Install New Application** to deploy the Web service. For more information, see Installing a new application.
2. Once you have deployed the Web service, click **Applications > Enterprise Applications >** *application _name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending on which service you want to configure.
   a. If you select **Web Modules**, click the WAR file that you want to edit.
   b. If you select **EJB Modules**, click the JAR file that you want to edit.
4. Once you select a WAR or JAR file, under **Additional Properties**, click **Web Services: Client Security Bindings** for outbound requests and inbound responses or click **Web Services: Server Security Bindings** for inbound requests and outbound responses.
5. If you click **Web Services: Server Security Bindings**, the following sections can be configured. These topics are discussed in more details in other sections of the documentation.
   - Request receiver binding
     – Signing Information
     – Encryption Information
     – Trust anchors
     – Collection certificate store
     – Key locators
     – Trusted ID evaluators
     – Login mappings
   - Response sender binding
     – Signing Information
     – Encryption Information
     – Key locators

XML encryption

XML Encryption is a specification developed by W3C in 2002 that contains the steps to encrypt data; the steps to decrypt encrypted data; the XML syntax to represent encrypted data; the information used to decrypt the data; and a list of encryption algorithms such as triple DES, AES, and RSA.

You can apply XML encryption to an XML element, XML element content, and arbitrary data, including an XML document. For example, suppose that you need to encrypt the `CreditCard` element shown in the example 1.

Example 1: Sample XML document

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Example 2: XML document with a common secret key

Example 2 shows the XML document after encryption. The `EncryptedData` element represents the encrypted `CreditCard` element. The `EncryptionMethod` element describes the applied encryption algorithm, which is triple DES in this example. The `KeyInfo` element contains the information to retrieve a decryption key, which is a `KeyName` element in this example. The `CipherValue` element contains the ciphertext obtained by serializing and encrypting the `CreditCard` element.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
       Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
       <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
         <KeyName>John Smith</KeyName>
       </KeyInfo>
       <CipherData>
         <CipherValue>ydUNqHkMrD...</CipherValue>
       </CipherData>
  </EncryptedData>
</PaymentInfo>
```

Example 3: XML document encrypted with the public key of the recipient

In example 2, it is assumed that both the sender and recipient have a common secret key. If the recipient has a public and private key pair, which is a most likely the case, the `CreditCard` element can be encrypted as shown in example 3. The `EncryptedData` element is the same as the `EncryptedData` element found in example 2. However, the `KeyInfo` element contains an `EncryptedKey`.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
```

```
      <CipherValue>ydUNqHkMrD...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

XML Encryption in WSS-Core

WSS-Core is a specification under development by OASIS. The specification describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. The message confidentiality is realized by encryption based on XML Encryption.

The WSS-Core specification allows encryption of any combination of body blocks, header blocks, their sub-structures, and attachments of a SOAP message. The specification also requires that when you encrypt parts of a SOAP message, you must prepend a reference from the security header block to the encrypted parts of the message. The reference can be a clue for a recipient to identify which encrypted parts of the message to decrypt.

The XML syntax of the reference varies according to what information is encrypted and how it is encrypted. For example, suppose that the `CreditCard` element in example 4 is encrypted with either a common secret key or the public key of the recipient.

Example 4: Sample SOAP message

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <CreditCard Limit='5,000' Currency='USD'>
        <Number>4019 2445 0277 5567</Number>
        <Issuer>Example Bank</Issuer>
        <Expiration>04/02</Expiration>
      </CreditCard>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The resulting SOAP messages are shown in examples 5 and 6. In these example, the `ReferenceList` and `EncryptedKey` elements are used as references, respectively.

Example 5: SOAP message encrypted with a common secret key

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <ReferenceList xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
```

```
          <KeyName>John Smith</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6: SOAP message encrypted with public key of the recipient

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI='#ed1'/>
        </ReferenceList>
      </EncryptedKey>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Relationship to Digital Signature

The WSS-Core specification also provides message integrity, which is realized by digital signature based on XML-Signature.

CAUTION:
A combination of encryption and digital signature over common data introduces cryptographic vulnerabilities.

Securing Web services using XML encryption

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) encryption is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

XML encryption enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as a XML document. Like XML digital signature, a message is sent by the client as the request sender to the server as the request receiver. The response is sent by the server as the response sender to the client as the request receiver. Unlike XML digital signature, which verifies the authenticity of the sender, XML encryption scrambles the message content using a key, which can be unscrambled by a receiver that possess the same key. You can use XML encryption in conjunction with XML digital signature; scrambling the content while verifying the authenticity of the message sender. To use XML encryption to secure Web services, complete the following tasks. You must use the WebSphere Application Server Toolkit, which is available at the following Web site:http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q =ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en

1. Specify the encryption settings for the request sender. The message parts and encryption method settings chosen for the request sender on the client must match the message parts and method settings chosen for the request receiver on the server. To specify the encryption settings for the request sender, complete the following steps:
   a. Configure the client for request encryption: encrypting the message parts.
   b. Configure the client for request encryption: choosing the encryption method.
2. Specify the encryption settings for the request receiver.

   Remember: The decryption settings chosen for the request receiver must match the encryption settings chosen for the request sender.

   To specify the decryption settings for the request receiver, complete the following steps:
   a. Configure the server for request decryption: decrypting message parts.
   b. Configure the server for request decryption: choosing the decryption method.
3. Specify the encryption settings for the response sender. The message parts and encryption method settings chosen for the response sender on the server must match the message parts and method settings chosen for the response receiver on the client. To specify the encryption settings for the response sender, complete the following steps:
   a. Configure the server for response encryption: encrypting message parts.
   b. Configure the server for response encryption: choosing the encryption method.
4. Specify the encryption settings for the response receiver.

   Remember: The decryption settings chosen for the response receiver must match the encryption settings chosen for the response sender.

   To specify the decryption settings for the response receiver, complete the following steps:
   a. Configure the client for response decryption: decrypting message parts.
   b. Configure the client for response decryption: choosing the decryption method.

After completing these steps, you have secured your Web services using XML encryption.

Encryption information collection

Use this page to specify the configuration for the encrypting and decrypting parameters. This configuration is used to encrypt and decrypt parts of the message, including the body and user name token.
To view this administrative console page, complete the following steps:

1. Click **Enterprise Applications>** *application_name*.

2. Under Related Items, Click **Web Module**.
3. Under Additional Properties, click **Web Services: Server Security Bindings**.
4. Under Request Receiver Binding, click **Edit > Encryption Information**.
5. Click **New** to create an encryption method.
6. Click **Delete** to delete an encryption method.

Encryption Information:

Specifies the name of the encryption information.

Encryption information configuration settings

Use this page to configure the encryption and decryption parameters. You can use these parameters to encrypt and decrypt various parts of the message including the body and user name token.
To view this administrative console page, click **Applications > Enterprise Applications** >*application_name*. Under Related Items, click **Web Module** > *URI_file_name* > **Web Services: Server Security Bindings**. Under Request Receiver Binding, click **Edit > Encryption Information > New**.

Encryption Information Name:

Specifies the name for the encryption information.

Key Locator Reference:

Specifies the name used to reference the key locator.

To specify key locator references, click **Servers > Application Servers** >*server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Encryption Key Name:

Specifies the name of the encryption key, which is resolved to the actual key by the specified key locator.

Key Encryption Algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the key encryption method.

The following algorithms are supported:
- `http://www.w3.org/2001/04/xmlenc#rsa-1_5`
- `http://www.w3.org/2001/04/xmlenc#kw-tripledes`

5.1 + The following additional algorithms are supported:
- `http://www.w3.org/2001/04/xmlenc#kw-aes128`
- `http://www.w3.org/2001/04/xmlenc#kw-aes256`
- `http://www.w3.org/2001/04/xmlenc#kw-aes192`

Data Encryption Algorithm:

Specifies the algorithm URI of the data encryption method.

The following algorithm is supported:
- `http://www.w3.org/2001/04/xmlenc#tripledes-cbc`

5.1 + The following additional algorithms are supported:
- `http://www.w3.org/2001/04/xmlenc#aes128-cbc`

- `http://www.w3.org/2001/04/xmlenc#aes256-cbc`
- `http://www.w3.org/2001/04/xmlenc#aes192-cbc`

Encryption information configuration settings

Use this page to configure the encryption and decryption parameters.

The specifications listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XML Encryption Syntax and Processing: W3C Recommendation 10 Dec 2002*.

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: Server Security Bindings**.
3. Under Response Sender Binding, click **Edit > Encryption Information**.
4. If the encryption information is not available, select **None**.
5. If the encryption information is available, select **Dedicated Encryption Information**.

Then, specify the configuration in the following fields:

Encryption Information Name:

Specifies the name for the encryption information.

Key Locator Reference:

Specifies the name used to reference the key locator.

To specify key locator references, click **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Encryption Key Name:

Specifies the name of the encryption key, which is resolved to the actual key by the specified key locator.

Key Encryption Algorithm:

Specifies the algorithm URI of the key encryption method.

The following algorithms are supported:
- `http://www.w3.org/2001/04/xmlenc#rsa-1_5`
- `http://www.w3.org/2001/04/xmlenc#kw-tripledes`

5.1 + The following additional algorithms are supported:
- `http://www.w3.org/2001/04/xmlenc#kw-aes128`
- `http://www.w3.org/2001/04/xmlenc#kw-aes256`
- `http://www.w3.org/2001/04/xmlenc#kw-aes192`

By default the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256- bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files. Before downloading these policy files, back up the existing policy files (`local_policy.jar` and `US_export_policy.jar` in the jre/lib/security/ directory) prior to overwriting them in case you want to restore the original files later. To download the policy files, complete either of the following sets of steps:

- For WebSphere Application Server platforms using IBM Developer Kit, Java Technology Edition Version 1.4.1, including the AIX, Linux, and Windows platforms, you can obtain unlimited jurisdiction policy files by completing the following steps:
  - Go to the following Web site: `http://www.ibm.com/developerworks/java/jdk/security/index.html`
  - Click **Policy Files**.

    The `unrestrict.jar` file is downloaded onto your machine.
- For WebSphere Application Server platforms using the Sun-based Java Development Kit (JDK) Version 1.4.1, including the Solaris environments and the HP-UX platform, you can obtain unlimited jurisdiction policy files by completing the following steps:
  - Go to the following Web site: `http://java.sun.com/j2se/1.4.1/download.html`
  - Go to the bottom of the Web page and click Download, which is next to Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 1.4.1. The `jce_policy-1_4_1.zip` file is downloaded onto your machine.

After following either of these sets of steps, two Java Archive (JAR) files are placed in the JVM `jre/lib/security/` directory.

Data Encryption Algorithm:

Specifies the algorithm Uniform Resource Identifiers (URI) of the data encryption method.

The following algorithm is supported:

- `http://www.w3.org/2001/04/xmlenc#tripledes-cbc`

5.1 + The following additional algorithms are supported:

- `http://www.w3.org/2001/04/xmlenc#aes128-cbc`
- `http://www.w3.org/2001/04/xmlenc#aes256-cbc`
- `http://www.w3.org/2001/04/xmlenc#aes192-cbc`

By default the JCE is shipped with restricted or limited strength ciphers. To use 192-bit and 256- bit AES encryption algorithms, you must apply unlimited jurisdiction policy files.

Login bindings configuration settings

Use this page to configure the encryption and decryption parameters.

The pluggable token uses the Java Authentication and Authorization Service (JAAS) CallBackHandler (javax.security.auth.callback.CallBackHandler) interface to generate the token that is inserted into the message. The following list describes the CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback

This implementation is used for generating binary tokens inserted as `<wsse:BinarySecurityToken/@ValueType>` in the message.

javax.security.auth.callback.NameCallback and javax.security.auth.callback.NameCallback

This implementation is used for generating user name tokens inserted as `<wsse:UsernameToken>` in the message.

com.ibm.wsspi.wssecurity.auth.callback.XMLTokenSenderCallback

This implementation is used to generate eXtensible Markup Language (XML) tokens and is inserted as the `<SAML: Assertion>` element in the message.

com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

This implementation is used to obtain properties specified in the binding file.

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 Request Sender Bindings, click **Edit > Login Binding**.

If the encryption information is not available, select **None**.

If the encryption information is available, select **Dedicated Login Binding** and specify the configuration in the following fields:

Authentication Method:

Specifies the unique name for the authentication method.

Callback Handler:

Specifies the name of the callback handler.The callback handler must implement the javax.security.auth.callback.CallbackHandler interface.

Basic Auth User ID:

Specifies the user name for basic authentication. The Basic Auth authentication method provides the capability to define a user ID and password in the binding file.

Basic Auth Password:

Specifies the password for basic authentication.

Token Type URI:

Specifies the Uniform Resource Identifiers (URI) for the token type. This information is inserted as `<wsse:BinarySecurityToken>/ValueType` for the XML token `<SAML: Assertion>`.

Token Type Local Name:

Specifies the local name for the token type. This information is inserted as `<wsse:BinarySecurityToken>/ValueType` for the XML token `<SAML: Assertion>`.

Request sender

The security handler on the request sender side of the SOAP message enforces the security constraints, located in the `ibm-webservicesclient-ext.xmi` file, and bindings, located in the `ibm-webservicesclient-bnd.xmi file`. These constraints and bindings apply both to J2EE application clients or when Web services is acting as a client. The security handler acts on the security constraints before sending the SOAP message. For example, the security handler might digitally sign the message, encrypt the message, create a time stamp, or insert a security token.
The security handler on the request sender side of the SOAP message enforces the security constraints, located in the `ibm-webservicesclient-ext.xmi` file, and bindings, located in the `ibm-webservicesclient-bnd.xmi file`. These constraints and bindings apply both to J2EE application clients or when Web services is acting as a client. The security handler acts on the security constraints before sending the SOAP message. For example, the security handler might digitally sign the message, encrypt the message, create a time stamp, or insert a security token. You can specify the following security requirements for the request sender and apply them to the SOAP message:

Integrity (digital signature)

You can select multiple parts of a message to be digitally signed. The following is a list of integrity options:

- Body
- Time stamp
- Security token

Confidentiality (encryption)

You can select multiple parts of a message to be encrypted. The following is a list of confidentiality options:

- Body content
- Username token

Security token

You can insert only one token into the message. The following is a list of security token options:

- Basic authentication, which requires both a user name and password
- Identity assertion, which requires a user name only
- X.509 binary security token
- LTPA binary security token
- Custom token , which is pluggable and allows custom-defined tokens to be inserted into the SOAP message

Timestamp

You can have a time stamp for indicating the freshness of the message.

- Timestamp

Note: Request sender security constraints apply to the SOAP message and must match the security constraint requirements defined in request receiver.

Request sender binding collection:

Use this page to specify the binding configuration to send request messages for Web services security.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Client Security Bindings**.
3. Under Request Sender Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of the message including the body and time stamp.

You can also use these parameters for X.509 validation when *Authentication Method* is `IDAssertion` and *ID Type* is `X509Certificate` in the server-level configuration. In such cases, you must fill in the *Certificate Path* fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters. Encryption information is used for encrypting and decrypting various parts of a message including the body and user name token.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Login Mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, use the administrative console and click **Security > JAAS Configuration**.

Configuring the client for request encryption: Encrypting the message parts

Prior to completing these steps, read either of the following topics to familiarize yourself with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which message parts to encrypt when configuring the client for request encryption:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand **Request Sender Configuration > Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone being able to understand the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting , see XML encryption.
8. Select the parts of the message that you want to encrypt by clicking **Add**. You can select one of the following parts:

   `Bodycontent`
   User data portion of the message

   `Usernametoken`
   Basic authentication information, if selected

Once you have specified which message parts to encrypt, you must specify which method to use to encrypt the request message. See Configuring the client for request encryption: Choosing the encryption method for more information.

Configuring the client for request encryption: Choosing the encryption method

Prior to completing these steps, read either of the following topics to familiarize yourself with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which encryption method to use when configuring the client for request encryption:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand **Security Request Sender Binding Configuration > Encryption Information**.
8. Select an encryption option and click **Edit** to view the encryption information or click **Add** to add another option.   The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   The encryption name refers to the name of the encryption information entry.

   Data encryption method algorithm
   The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks.

   Key encryption method algorithm
   The key encryption method algorithms are public key encryption algorithms that are specified for encrypting and decrypting keys.

   Encryption key name
   The encryption key name represents a *Subject* (*Owner* field of the certificate) from a certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

   Note: The key chosen must be a public key of the target. Encryption must be done using the public key and decryption must be done by the target using the private key (the personal certificate of the target).

   Encryption key locator
   The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

The encryption key name chosen must refer to a public key of the target. For the encryption key name, use the Subject (Owner field of the certificate) of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator , the encryption key name might be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

You must specify which parts of the request message to encrypt. See Configuring the client for request encryption: Encrypting the message parts if you have not previously specified this information.

Request receiver

The security handler on the request receiver side of the SOAP message enforces the security specifications defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`) and bindings (`ibm-webservices-bnd.xmi`). The request receiver defines the security requirement of the SOAP message. If the incoming SOAP message does not meet all the security requirements defined, then the request is

rejected with the appropriate fault code returned to the sender. For security tokens, the token is validated using Java Authentication and Authorization Service (JAAS) login configuration and authenticated identity is set as the identity for the downstream invocation.
The security handler on the request receiver side of the SOAP message enforces the security specifications defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`) and bindings (`ibm-webservices-bnd.xmi`). The request receiver defines the security requirement of the SOAP message. If the incoming SOAP message does not meet all the security requirements defined, then the request is rejected with the appropriate fault code returned to the sender. For security tokens, the token is validated using Java Authentication and Authorization Service (JAAS) login configuration and authenticated identity is set as the identity for the downstream invocation.

For example, if there is a security requirement to have the SOAP body digitally signed by Joe Smith and if the SOAP body of the incoming SOAP message is not signed by Joe Smith, then the request is rejected.

You can define the following security requirements for request receiver:

Required integrity (digital signature)
You can select multiple parts of a message to be digitally signed. The following is a list of integrity options:
- Body
- Time stamp
- Security token

Required confidentiality (encryption)
You can select multiple parts of a message to be encrypted. The following is a list of confidentiality options:
- Body content
- Token

You can have multiple security tokens. The following is a list of security token options:
- Basic authentication, which requires both a user name and password
- Identity assertion, which requires a user name only
- X.509 binary security token
- LAP binary security token
- Custom token, which is pluggable and allows custom-defined tokens to be validated by the JAAS login configuration

Received time stamp
You can have a time stamp for checking the freshness of the message.
- Time stamp

Note: The security constraint for request sender must match the security requirement of the request receiver for the request to be accepted by the server.

Request receiver binding collection:

Use this page to specify the binding configuration to receive request messages for Web services security.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Request Receiver Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of a message including the body, time stamp, and user name token.

You also can use these parameters for X.509 certificate validation when Authentication Method is IDAssertion and ID Type is X509Certificate in the server-level configuration. In such cases, you must fill in the **Certificate Path** fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters. This configuration is used to encrypt and decrypt parts of the message that include the body and user name token.

Trust Anchors:

Specifies a list of keystore objects that contain the trusted root certificates that are issued by a certificate authority (CA).

The certificate authority authenticates a user and issues a certificate. The CertPath API uses the certificate to validate the certificate chain of incoming, X.509-formatted security tokens or trusted, self-signed certificates.

Collection Certificate Store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates.The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Trusted ID Evaluators:

Specifies a list of trusted ID evaluators that determine whether to trust the identity-asserting authority or message sender.

The trusted ID evaluators are used to authenticate additional identities from one server to another server. For example, a client sends the identity of user A to server 1 for authentication. Server 1 calls downstream to server 2, asserts the identity of user A, and includes the user ID and password of server 1. Server 2 attempts to establish trust with server 1 by authenticating its user ID and password and checking the trust based on the TrustedIDEvaluator implementation. If the authentication process and the trust check are successful, server 2 trusts that server 1 authenticated user A and a credential is created for user A on server 2 to invoke the request.

Login Mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, use the administrative console and click **Security > JAAS Configuration**.

Configuring the server for request decryption: decrypting the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete this task to specify which parts of the request message must be decrypted by the server. You must know which parts of the request message the client encrypts because the server must decrypt the same message parts.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Required Confidentiality** section.
8. Select the parts of the message to decrypt The message parts selected for the request decryption on the server must match the message parts selected for the message encryption on the client. Click **Add** and select either of the following message parts:

   `bodycontent`
   The user data section of the message.

   usernametoken
   This token is the basic authentication information.

Once you have specified which parts of the request message to decrypt, you must specify the method used to decrypt the message. See Configuring the server for request decryption: choosing the decryption method for more information.

Configuring the server for request decryption: choosing the decryption method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete this task to specify which decryption method is used by the server to decrypt the request message. You must know which decryption method the client uses because the server must use the same method.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.

4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Encryption Information** section.
8. Click **Edit** to view the encryption information. The following table describes the purpose for each of these selections. Some definitions take from the XML-Encryption specification , which can be found at: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Encryption name is the name of this encryption information entry. This is an alias for the entry.

   Data encryption method algorithm
   Data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Key encryption method algorithm
   Key encryption method algorithms are public key encryption algorithms specified for encrypting and decrypting keys. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Encryption key name
   Encryption key name represents a Subject (from a certificate) found by the encryption key locator. the Subject is used by the key encryption method algorithm to decrypt the secret key, and the secret key is used to decrypt the data.

   Attention: The key chosen here should be a private key in the keystore configured by the key locator. The key should have the same Subject used by the client to encrypt the data. Encryption must be done using the public key and decryption by the private key (personal certificate). To ensure that the client encrypts the data with the correct public or private key, you must extract the public key from the server keystore and add it to the keystore specified in the encryption configuration information for the client request sender.

   For example, the personal certificate of a server is `CN=Bob, O=IBM, C=US`. Therefore the server contains the public and private key pair. The client sending the request should encrypt the data using the public key for `CN=Bob, O=IBM, C=US`. The server decrypts the data using the private key for `CN=Bob, O=IBM, C=US`.

   Encryption key locator
   This represents a reference to a key locator implementation. For more information on configuring key locators, go to the following sections: Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

It is important to note that for decryption, the encryption key name chosen must refer to a personal certificate that can be located by the key locator of the server referenced in the encryption information. Enter the Subject of the personal certificate here, which is typically a Distinguished Name (DN). The Subject uses the default key locator to find the key. If a custom key locator is written, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that finds the correct key store where this alias and certificate exist. Refer to Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console for more information.

You must specify which parts of the request message to decrypt. See Configuring the server for request decryption: decrypting the message parts if you have not previously specified this information.

Response sender

The response sender defines the security requirements of the SOAP response message. The security handler acts on the security constraints defined for the response in the IBM extension deployment descriptors, located in the `ibm-webservices-ext.xmi` file and bindings, located in the `ibm-webservices-bnd.xmi` file. The security handler signs, encrypts, or generates the time stamp for the SOAP response message before the response is send to the caller.

The response sender defines the security requirements of the SOAP response message. The security handler acts on the security constraints defined for the response in the IBM extension deployment descriptors, located in the `ibm-webservices-ext.xmi` file and bindings, located in the `ibm-webservices-bnd.xmi` file. The security handler signs, encrypts, or generates the time stamp for the SOAP response message before the response is send to the caller.

Integrity constraints (digital signature)
You can select multiple parts of the message to be digitally signed.
- Body
- Time stamp

Confidentiality (encryption)
You can encrypt the body content of the message

Time stamp
You can have a time stamp for checking the freshness of the message.

Note: The security constraints that apply to the SOAP response message must match the security requirements defined in the response receiver. Otherwise, the response is rejected by the response receiver (caller).

Response sender binding collection:

Use this page to specify the binding configuration for sender response messages for Web services security.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Response Sender Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters.

You also can use these parameters for X.509 certificate validation when Authentication Method is IDAssertion and ID Type is X509Certificate in the server-level configuration. In such cases, you must fill-in the **Certificate Path** fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Configuring the server for response encryption: encrypting the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which parts of the response message to encrypt when configuring the server for response encryption:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand **Response Sender Service Configuration Details > Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone being able to understand the message flowing across the Internet. With confidentiality specifications, the response is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting, see XML encryption.
8. Select the parts of the response that you want to encrypt by clicking **Add** and selecting **Bodytoken** or **Usernametoken**. The following information describes the message parts:

   `Bodycontent`
   User data portion of the message.

   `Usernametoken`
   Basic authentication information, if selected.

   A user name token does not appear in the response. You do not need to select this option for the response. If you select this option, make sure that you also select it for the client response receiver. If you do not select it, make sure that you do not select it for the client response receiver either.

Once you have specified which message parts to encrypt, you must specify which method is used to encrypt the message. See Configuring the server for response encryption: choosing the encryption method for more information.

Configuring the server for response encryption: Choosing the encryption method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which method the server uses to encrypt the response message:

1. Launch the Assembly Toolkit.
2. Click **Windows > Open Perspective > J2EE** to access the Assembly Toolkit perspective.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand **Response Sender Binding Configuration Details > Encryption Information**.
8. Click **Edit** to view the encryption information. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   The encryption name refers to the name of the encryption information entry.

   Data encryption method algorithm
   The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

   Key encryption method algorithm
   The key encryption method algorithms are public key encryption algorithms that are specified for encrypting and decrypting keys. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

   Encryption key name
   The encryption key name represents a Subject from a certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

   Important: The key name chosen in the server response sender encryption information must be the public key of the key configured in the client response receiver encryption information. Encryption by the response sender must be done using the public key and decryption must be done by the response receiver using the associated private key (the personal certificate of the response receiver).

   Encryption key locator
   The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

The encryption key name chosen must refer to a public key of the response receiver. For the encryption key name, use the Subject of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator , the encryption key name might be anything used by the key locator to find the correct encryption key (a public key). The encryption key locator references the implementation class that finds the correct key store where the alias and certificate exist. For more information, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

You must specify which parts of the response message to encrypt. See Configuring the server for response encryption: encrypting the message parts if you have not previously specified this information.

Response receiver

The response receiver defines the security requirements of the response received from a request to a Web service. The security handler enforces the security constraints based on the security requirements

defined in the IBM extension deployment descriptor, located in the `ibm-webservicesclient-ext.xmi` file and in the bindings, located in the `ibm-webservicessclient-bnd.xmi` file.
The response receiver defines the security requirements of the response received from a request to a Web service. The security handler enforces the security constraints based on the security requirements defined in the IBM extension deployment descriptor, located in the `ibm-webservicesclient-ext.xmi` file and in the bindings, located in the `ibm-webservicessclient-bnd.xmi` file.

For example, the security requirement might be to have the response SOAP body encrypted. If the SOAP body of the SOAP message is not encrypted, the response is rejected and the appropriate fault code is communicated back to the caller of the Web services.

You can specify the following security requirements for response receiver:

Required integrity (digital signature)
You can select multiple parts of a message to be digitally signed. The following is a list of integrity options:
- Body
- Time stamp

Required confidentiality (encryption)
You can encrypt the body content of the message.

Received time stamp
You can have a time stamp for checking the freshness of the message.

Note: The security constraints for response sender must match the security requirements of the response receiver. If the constraints do not match, the response is not accepted by the caller or sender.

Response receiver binding collection:

Use this page to specify the binding configuration for receiver response messages for Web services security.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Response Sender Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of the message including the body and time stamp.

You can also use these parameters for X.509 validation when *Authentication Method* is `IDAssertion` and *ID Type* is `X509Certificate` in the server-level configuration. In such cases, you must fill in the *Certificate Path* fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters. Encryption information is used for encrypting and decrypting various parts of a message including the body and user name token.

Trust Anchors:

Specifies a list of keystore objects that contain the trusted root certificates, that are self-signed or issued by a certificate authority.

The certificate authority authenticates a user and issues a certificate. After the certificate is issued, the key store objects, which contain these certificates, use the certificate for certificate path or certificate chain validation of incoming X.509-formatted security tokens.

Collection Certificate Store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates.The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a key store file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Configuring the client for response decryption: decrypting the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:
- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which response message parts to decrypt when configuring the client for response decryption. The server response encryption and client response decryption configurations must match.
1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Response Receiver Configuration > Required Confidentiality** section.
8. Select the parts of the message that you must decrypt by clicking **Add** and selecting either **Bodycontent** or **Usernametoken**. The following information describes these message parts:

   Bodycontent
   The user data portion of the message.

   Usernametoken
   The basic authentication information, if selected.

   The information selected in this step is encrypted by the server in the response sender.

   Important: A username token is typically not sent in the response. Thus, you usually do not need to select username token.

Once you have specified which message parts to decrypt, you must specify which method to use when decrypting the response message. See Configuring the client for response decryption: choosing a decryption method for more information.

Configuring the client for response decryption: choosing a decryption method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which decryption method to use when the client decrypts the response message. The server response encryption and client response decryption configurations must match.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Response Receiver Binding Configuration > Encryption Information** section. For more information on encrypting and decrypting SOAP messages, see XML encryption.
8. Click **Edit** to view the encryption information. The following table describes the purpose for this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following address: http://www.w3.org/TR/xmlenc-core

   Encryption name

   The encryption name refers to the alias used for the encryption information entry.

   Data encryption method algorithm

   The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks.

   Key encryption method algorithm

   The key encryption method algorithms are public key encryption algorithms specified for encrypting and decrypting keys.

   Encryption key name

   The encryption key name represents a Subject from a certificate found by the encryption key locator. The Subject is used by the key encryption method algorithm to decrypt the secret key. The secret key is used to decrypt the data.

   Important: The key chosen must be a private key of the client. Encryption must be done using the public key and decryption must be done by the private key (personal certificate). For example, the personal certificate of the client is: `CN=Alice, O=IBM, C=US`. Therefore, the client contains the public and private key pair. The target server that sends the response encrypts the secret key using the public key for `CN=Alice, O=IBM, C=US`. The client decrypts the secret key using the private key for CN=Alice, O=IBM, C=US.

Encryption key locator
The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

For decryption, the encryption key name chosen must refer to a personal certificate that can be located by the client key locator. The Subject (owner field of the certificate) of the personal certificate should be entered in the Encryption key name, this is typically a Distinguished Name (DN). The default key locator uses the Encryption key name to find the key within the keystore. If you write a custom key locator, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

You must specify which parts of the request message to decrypt. See the topicConfiguring the client for response decryption: decrypting the message parts if you have not previously specified this information.

Securing Web services using basicauth authentication

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:
- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

With the basicauth authentication method, the request sender generates a basicauth security token using a callback handler. The request receiver retrieves the basicauth security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. Trust is established using user name and password validation. To use basicauth authentication to secure Web services, complete the following tasks:
1. Secure the client for basicauth authentication.
   a. Configure the client for basicauth authentication: specifying the method
   b. Configure the client for basicauth authentication: collecting the authentication information
2. Secure the server for basicauth authentication.
   a. Configure the server to handle basicauth authentication
   b. Configure the server to validate basicauth authentication information

After completing these steps, you have secured your Web services using basicauth authentication.

Configuring the client for basic authentication: Specifying the method

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a GUI prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction. For more information on BasicAuth authentication, see BasicAuth authentication method.

Attention: 5.1 + WebSphere Application Server supports nonce (randomly generated token) with BasicAuth authentication. For more information, see Nonce.

Complete the following steps to specify BasicAuth as the authentication method:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB module or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section. The only valid login configuration choices for a pure client are BasicAuth and Signature.
8. Select **BasicAuth** to authenticate the client using a user ID and password. This user ID and password must be specified in the target user registry. The other choice, Signature, attempts to authenticate the client using the certificate used to digitally sign the message.

For more information on getting started with the Web Services Client Editor within the Assembly Toolkit, see either of the following topics:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

Once you have specified BasicAuth as the authentication method, you must specify how to collect the authentication information. See Configuring the client for basic authentication: collecting the authentication information.

BasicAuth authentication method:

When you use the BasicAuth authentication method, the security token that is generated is a `<wsse:UsernameToken>` element with `<wsse:Username>` and `<wsse:Password>` elements.

WebSphere Application Server supports text passwords but not password digest because passwords are not stored and cannot be retrieved from the server. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

BasicAuth token generation

The request sender generates a BasicAuth security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler that is used is specified in the `<LoginBinding>` element of the bindings file, `ibm-webservicesclient-bnd.xmi` . The following callback handler implementations are provided with WebSphere Application Server and can be used with the BasicAuth authentication method:

- `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

You can add your own callback handlers that implement `javax.security.auth.callback.CallbackHandler`.

BasicAuth token validation

The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a JAAS login module. The `<wsse:Username>` and `<wsse:Password>` elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject then is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the `<LoginMapping>` element of the bindings file. There are default bindings specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file for a match. WebSphere Application Server provides the `WSLogin` default configuration entry, which is suitable for the BasicAuth authentication method.

Configuring the client for basic authentication: collecting the authentication information

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a GUI prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction. For more information on BasicAuth authentication, see BasicAuth authentication method.

Complete this task to specify the authentication information needed for BasicAuth authentication:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
8. Click **Edit** or **Enable** to view the Login Binding information. The login binding information displays and enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that will occur. Select BasicAuth to use basic authentication.

   Token value type URI and Token value type local name
   When you select BasicAuth, you cannot edit the token value type URI and local name values. These values are specifically for custom authentication types. For BasicAuth authentication, you do not need to enter any information.

   Callback handler
   The callback handler specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting the BasicAuth information. You can use the following default implementations for the callback handler:

   `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`
   This implementation is used for non-GUI console prompts.

   `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`
   This implementation is used for GUI panel prompts.

   `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`
   This implementation is used when you plan to always enter the user ID and password in the BasicAuth user ID and password section that follows.

Basic Authentication user ID and Basic Authentication password
When values for BasicAuth user ID and password are entered, regardless of the default callback handler indicated previously, these user ID and password values are used to authenticate to the server for the Web services security authentication. If you leave these values blank, use either the GUIPromptCallbackHandler or the StdinPromptCallbackHandler implementation, but only on a pure client. Always fill-in these values for any Web service that acts as a client to another Web service and you want to specify BasicAuth for authentication downstream. If you want the client identity of the originator to flow downstream, configure the Web service client to use either ID assertion or Lightweight Third Party Authentication (LTPA). To configure ID assertion, see Configuring the client for identity assertion authentication: Specifying method,Configuring the client for identity assertion authentication: Collecting the authentication method, Configuring the client for LTPA token authentication: Specifying the LTPA token authentication information, and Configuring the client for LTPA token authentication: Collecting the authentication information.

Property
This field enables you to enter properties and name and value pairs for use by custom callback handlers. For BasicAuth authentication, you do not need to enter any information. To enter a new property, click **Add** and enter the new property and value.

Attention: There is a basic authentication entry in the **Port Qualified Name Binding Details** section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the **Web services security basic authentication** section overrides the basic authentication information specified in the **Port Qualified Name Binding Details** section for authorizing the Web service.

For a server that acts as a client, do not specify a GUI or non-GUI prompt callback handler. To configure BasicAuth authentication from one Web service to a downstream Web service, select the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHander` implementation and explicitly specify the BasicAuth user ID and password. If you want the client identity of the originator to flow downstream, configure the Web service client to use ID assertion. To configure ID assertion, see Configuring the client for identity assertion authentication: Specifying method and Configuring the client for identity assertion authentication: Collecting the authentication method.

To use the BasicAuth authentication method, you must specify the method in the **Login Config** section of the Assembly Toolkit. See Configuring the client for basicauth authentication: specifying the method if you have not previously specified this information.

Identity assertion authentication method:

When using the Identity Assertion (IDAssertion) authentication method, the security token generated is a `<wsse:UsernameToken>` element that contains a `<wsse:Username>` element.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. These two operations, token generation and token validation, are described in the following sections.

Identity assertion token validation:

The request receiver retrieves the IDAssertion security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. With identity assertion, special processing is required to establish trust before asserting the identity as the established identity of the thread of execution. This special processing is defined by the `<IDAssertion>` element in the deployment descriptor file, `ibm-webservices-ext.xmi`. If all the validation checks are successful, the asserted identity is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the`<LoginMapping>` element of the bindings file. There are default bindings specified in the `ws-security.xml` file. However, you can override these bindings using the application specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found it searches the `wsjaas.conf` file. WebSphere Application Server provides the `system.wssecurity.IDAssertion` default configuration entry, which is suitable for the identity assertion authentication method.

The `<IDAssertion>` element in the `ibm-webservices-ext.xmi` deployment descriptor file specifies the special processing required when using the identity assertion authentication method. The `<IDAssertion>` element is composed of two sub-elements: `<IDType>` and `<TrustMode>`.

The `<IDType>` element specifies the method for asserting the identity. The supported values for asserting the identity are:

- Username
- Distinguished name (DN)
- X.509 certificate

When `<IDType>` is *username*, a username token (for example, Bob) is provided. This user name is mapped to a user in the user registry and is the asserted identity after successful trust validation. When the `<IDType>` is *DN*, a user name token containing a distinguished name is provided (for example, `cn=Bob Smith, o=ibm, c=us`). This DN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation. When the `<IDType>` is *X509Certificate*, a binary security token containing an X509 certificate is provided and the SubjectDN from the certificate (for example, `cn=Bob Smith, o=ibm, c=us`) is extracted. This SubjectDN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation.

The `<TrustMode>` element specifies how the trust authority, or asserting authority, provides trust information. The supported values are:

- Signature
- BasicAuth
- No value specified

When `<TrustMode>` is `Signature`, the signature is validated. Then, the signer (for example, `cn=IBM Authority, o=ibm, c=us`) is mapped to an identity in the user registry (for example, IBMAuthority). To ensure that the asserting authority is trusted, the mapped identity (for example, IBMAuthority) is validated against a list of trusted identities. When the `<TrustMode>` is `BasicAuth`, there is a user name token with a user name and password, which is the user name and password of the asserting authority. The user name and password are validated. If they are successfully validated, that user name (for example, IBMAuthority) is validated against a list of trusted identities. If a value is not specified for `<TrustMode>`, trust is presumed and additional trust validation is not performed. This type of identity assertion is called *presumed trust mode*. Use the presumed trust mode only in an environment where the trust is established using some other mechanism.

If all the validations described previously succeed, the asserted identity (for example, Bob) is set as the identity of the thread of execution. If any of the validations fail, the request is rejected with a SOAP fault exception.

Configuring the server to handle BasicAuth authentication information

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Once a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and password supplied is invalid, an exception is thrown and the request ends without invoking the resource. For more information on BasicAuth authentication, see BasicAuth.

Complete the following steps to configure the server to handle BasicAuth authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Config** section. The options you can select are:
   - BasicAuth
   - Signature
   - ID assertion
   - LTPA

   .
8. Select **BasicAuth** to authenticate the client using a user ID and password. The client must specify a valid user ID and password in the server user registry.

   Important: You can select multiple login configurations, which means that different types of security information might be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. This can cause problems if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token. For ID assertion to work properly, list ID assertion ahead of BasicAuth in the processing list or the BasicAuth processing overrides the IDAssertion processing.

Once you have specified how the server will handle BasicAuth authentication information, you must specify how the server validates the authentication information. See Configuring the server to validate basicauth authentication information for more information.

Configuring the server to validate BasicAuth authentication information

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Once a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and password supplied is invalid, an exception is thrown and the request ends without invoking the resource. For more information on BasicAuth authentication, see BasicAuth.

Complete the following steps to specify how the server validates the BasicAuth authentication information:

1. Launch the Assembly Toolkit.

2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog displays and either select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that occurs. Select **BasicAuth** to use basic authentication.

   Configuration name
   This specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the BasicAuth authentication method, enter **WSLogin** for the JAAS login Configuration name.

   Use token valid type
   The **Use token value type** option determines if you want to specify a custom token type. For the default Authentication method selections, you do not need to specify this option.

   Token value type URI and Token value type URI local name
   When you select BasicAuth, you cannot edit the token value type URI and local name values. These values are specified for custom authentication types. For BasicAuth authentication, you do not need to enter any information for these fields.

   Callback handler factory class name
   This class name creates a JAAS CallbackHandler implementation that understands the following callbacks:
   - `javax.security.auth.callback.NameCallback`
   - `javax.security.auth.callback.PasswordCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

   Callback handler factory property name and Callback handler factory property value
   This property is used to specify callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For BasicAuth, you do not need to enter any property values.

   Login mapping property name and Login mapping property value
   This property is used to specify properties for a custom login mapping to use. For the default implementations including BasicAuth, you do not need to enter any property values.

You must specify how the server will handle the BasicAuth authentication method. See Configuring the server to handle BasicAuth authentication information if you have not previously specified this information.

Identity assertion

Identity assertion is a method for expressing the identity of the sender (for example, user name) in a SOAP message. When identity assertion is used as a authentication method, the authentication decision is performed based only on the name of the identity, but not on other information such as passwords and certificates.

Identity assertion is a method for expressing the identity of the sender (for example, user name) in a SOAP message. When identity assertion is used as a authentication method, the authentication decision is performed based only on the name of the identity, but not on other information such as passwords and certificates.

ID type

The Web Services Security implementation in WebSphere Application Server can handle the following three types of identity.

User name
Denotes the user name, such as the one in the local operating system (for example, ″alice″). This name is embedded in the `<Username>` element within the `<UsernameToken>` element.

DN
Denotes the distinguished name (DN) for the user, such as ″CN=alice, O=IBM, C=US″. This name is embedded in the `<Username>` element within the `<UsernameToken>` element.

X.509 certificate
Represents the identity of the user as a X.509 certificate instead of a string name. This certificate is embedded in the `<BinarySecurityToken>` element.

Managing trust

The intermediary host in the SOAP message itinerary can assert the initial sender's claimed identity. Two methods (called *trust mode*) are supported for this assertion:

Basic authentication
The intermediary adds its user name and password pair to the message.

Signature
The intermediary digitally signs the `<UsernameToken>` element of the initial sender.

Note: This trust mode does not support the X.509 certificate ID type.

Typical scenario

ID assertion is typically used in the multi-hop environment where the SOAP message passes through one or more intermediary hosts. The intermediary host authenticates the initial sender. The following scenario describes the process:

1. The initial sender sends a SOAP message to the intermediary host with some embedded authentication information. This authentication information might be a user name and password pair and an LTPA token.
2. The intermediary host authenticates the initial sender according to the embedded authentication information.
3. The intermediary host removes the authentication information from the SOAP message and replaces it with the `<UsernameToken>` element, which contains a user name.
4. The intermediary host asserts the trust according to the trust mode.
5. The intermediary host sends the updated SOAP message to the ultimate receiver.
6. The ultimate receiver checks the trust against the intermediary host information according to the configured trust mode. Also, the trusted ID evaluator is invoked.
7. If trust is established by the ultimate receiver, it invokes the Web service under the authorization of the user name (that is, the initial sender) in the SOAP message.

Securing Web services using identity assertion authentication

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

With the identity assertion authentication method, the security token generates a `<wsee:Username Token>` element that contains a `<wsse:Username>` element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. Unlike basicauth authentication, trust is established through the use of a security token rather than through user name and password validation. To use identity assertion authentication to secure Web services, complete the following tasks:

1. Secure the client for identity assertion authentication.
   a. Configure the client for identity assertion authentication: specifying the method.
   b. Configure the client for identity assertion authentication: collecting the authentication information.
2. Secure the server for identity assertion authentication.
   a. Configure the server to handle identity assertion authentication.
   b. Configure the server to validate identity assertion authentication information.

After completing these steps, you have secured your Web services using identity assertion authentication.

Configuring the client for identity assertion: specifying the method

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a Web service acting as a client to a downstream Web service.

In order for the downstream Web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators.

Complete the following steps to specify identity assertion as the authentication method:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section.
8. Select **IDAssertion** as the authentication method. For more conceptual information on identity assertion authentication, see ID assertion.
9. Expand the **IDAssertion** section.
10. For the ID Type, select **Username**. This works with all registry types and originating authentication methods.
11. For the Trust Mode, select either **BasicAuth** or **Signature**.

- By selecting **BasicAuth**, you must include basic authentication information (user ID and password), which the downstream Web service has specified in the trusted ID evaluator as a trusted user ID. See Configuring the client for signature authentication: Collecting the authentication information to specify the user ID and password information.
- By selecting **Signature** , the certificate configured in the **Signature information** section used to sign the data also is used as the trusted subject. The Signature is used to create a credential and the user ID, which the certificate mapped to the downstream registry, is used in the trusted ID evaluator as a trusted user ID.

See Configuring the client security bindings using the Assembly Toolkit for more information on the Web Services Client Editor within the Assembly Toolkit.

Once you have specified identity assertion as the authentication method used by the client, you must specify how to collect the authentication information. See Configuring the client for identity assertion authentication: collecting the authentication information for more information.

Configuring the client for identity assertion: Collecting the authentication method

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a Web service acting as a client to a downstream Web service.

In order for the downstream Web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators.

Complete the following steps to specify how the client collects the authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
8. Click **Edit** to view the login binding information and select **IDAssertion**. The login binding dialog displays and either select or enter the following information:

   Authentication method

   The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use identity assertion.

   Token value type URI and Token value type Local name

   When you select IDAssertion, you cannot edit the token value type URI and the local name. These values are specifically for custom authentication types. For IDAssertion authentication, you do not need to enter any information.

   Callback handler

   The callback handler specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the BasicAuth information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler` implementation for IDAssertion.

Basic authentication User ID and Basic authentication Password
If the trust mode entered in the extensions is BasicAuth, you must specify the trusted user ID and password in these fields. The user ID specified must be an ID that is trusted by the downstream Web service. The Web service trusts the user ID if it is entered as a trusted ID in a trusted ID evaluator in the downstream Web service bindings. If the trust mode entered in the extensions is Signature, you do not need to specify any information in this field.

Property Name and Property Value
This field enables you to enter properties and name and value pairs, for use by custom callback handlers. For IDAssertion, you do not need to specify any information in this field.

To use the identity assertion authentication method, you must specify the method in the **Security Extentions** section of the Assembly Toolkit. See Configuring the client for identity assertion authentication: specifying the method if you have not previously specified this information.

Configuring the server to handle identity assertion authentication

Use this task to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client.

For the downstream Web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns `true` or `false` that this ID is trusted. Once it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to configure the server to handle identity assertion authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Config** section. The options you can select are:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **LTPA**
8. Select **IDAssertion** to authenticate the client using the identity assertion data provided. This user ID of the client must be in the target user registry configured in WebSphere Application Server global security. You can select global security in the Administrative Console by clicking **Security > Global security**.

   You can select multiple login configurations, which means that different types of security information can be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. This can cause problems if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token, which is the token that is being trusted. For ID assertion to work properly, you must list ID assertion ahead of BasicAuth in the list or BasicAuth processing overrides ID assertion processing.

9. Expand the **IDAssertion** section. You need to select both the **ID Type** and **Trust Mode**.
   a. For **ID Type**, the options are:
      - **Username**
      - **Distinguished name (DN)**
      - **X509certificate**

      These choices are just preferences and are not guaranteed. Most of the time **Username** is used. You must choose the same **ID Type** as the client.
   b. For **Trust Mode**, the options are:
      - **BasicAuth**
      - **Signature**

      The **Trust Mode** refers to the information sent by the client as the trusted ID.
      1) If you select **Signature**, the client signing certificate is sent. This certificate must be mappable to the configured user registry. For **Local OS**, the common name (CN) of the distinguished name (DN) is mapped to a user ID in the registry. For **LDAP**, the DN is mapped to the registry for the ExactDN mode. If it is in the CertificateFilter mode, attributes are mapped accordingly. In addition, the user name from the credential generated must be in the Trusted ID Evaluator trust list.
      2) If you select **BasicAuth**, the client sends basic authentication data (user ID and password). This basicauth data is authenticated to the configured user registry. Once the authentication occurs successfully, the user ID must be part of the trusted ID evaluator trust list.

For more information on getting started with the Web Services Editor within the Assembly Toolkit , see Configuring the server security bindings using the Assembly Toolkit.

Once you have specified how the server will handle identity assertion authentication information, you must specify how the server validates the authentication information. See Configuring the server to validate identity assertion authentication information for more information.

Configuring the server to validate identity assertion authentication information

Use this task to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client.

For the downstream Web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns `true` or `false` that this ID is trusted. Once it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to validate the identity assertion authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.

4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the login mapping information. Click **Add** to add new login mapping information. The login mapping dialog displays and either select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use basic authentication.

   Configuration name
   This specifies the JAAS login configuration name. For the IDAssertion authentication method, enter **system.wssecurity.IDAssertion** for the Java Authentication and Authorization Service (JAAS) login configuration name.

   Use token value type
   The **Use token value type** option determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.

   Token value type URI and Token value type local name
   When you select ID assertion, you cannot edit the **token value type URI** and **local name** values. These values are specifically for custom authentication types. For the ID assertion authentication method, you do not need to enter any information in these fields.

   Callback Handler Factory Class name
   This class name creates a JAAS CallbackHandler implementation that understands the following callbacks:
   - `javax.security.auth.callback.NameCallback`
   - `javax.security.auth.callback.PasswordCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

   For any of the default Authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including IDAssertion:

   ```
   com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
   ```

   This implementation creates the correct callback handler for the default implementations.

   Callback handler factory property name and Callback handler factory property value
   This property is used to specify callback handler properties for Custom callback handler factory implementations. The default callback handler factory implemetation does not need any properties to be specified. For ID assertion, you do not need to enter any values for this property.

   Login mapping property name and Login mapping property value
   This option is used to specify properties for a custom login mapping. For the default implementations including IDAssertion, you do not need to enter any properties for this option.
9. Expand the **Trusted ID Evaluator** section.
10. Click **Edit** to see a dialog displaying all the trusted ID evaluator information. The following table describes the purpose of this information.

Class name
The class name refers to the implementation of the trusted ID evaluator that you want to use. Enter the default implementation as `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`. If you want to implement your own trusted ID evaluator, you must implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

Property name
The name is the name of this configuration. Enter `BasicIDEvaluator` for lack of a better reference.

Property value
The property defines name and value pairs that can be used by the trusted ID evaluator implementation. For the default implementation, the trusted list is defined here. When a request comes in and the trusted ID is verified, the user ID, as it appears in the user registry, must be listed in this property. Specify the property as a name and value pair where the name is *trunstedId_n*. *n* is an integer starting from 0 and the value is the user ID associated with that name. An example list with the trusted names include two properties. For example: `trustedId_0 = user1`, `trustedId_1 = user2`. The previous example means that both user1 and user2 are trusted. user1 and user2 must be listed in the configured user registry.

11. Expand the **Trusted ID Evaluator Reference** section.
12. Click **Enable** to add a new entry. The text you enter for the **Trusted ID Evaluator Reference** must be the same as the name entered previously in the **Trusted ID Evaluator**. Make sure that the name matches exactly as the information is case sensitive. If an entry is already specified, you can change it by clicking **Edit**.

You must specify how the server will handle the identity assertion authentication method. See Configuring the server to handle identity assertion authentication if you have not previously specified this information.

Securing Web services using signature authentication

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

With the signature authentication method, the request sender generates a signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. To use signature authentication to secure Web services, complete the following tasks:

1. Secure the client for signature authentication.
   a. Configure the client for signature authentication: specifying the method.
   b. Configure the client for signature authentication: collecting the authentication information.
2. Secure the server for signature authentication.
   a. Configure the server to handle signature authentication.
   b. Configure the server to validate signature authentication information.

After completing these steps, you have secured your Web services using signature authentication.

Configuring the client for signature authentication: specifying the method

This task is used to configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server. For more information on signature authentication, see Signature authentication method.

Complete the following steps to specify signature as the authentication method:

1. Launch the Assembly Toolkit.
2. Click **Windows > Open Perspective > J2EE** to access the Assembly Toolkit perspective.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section. The following valid login configuration options for a managed client and Web services acting as a client are:

   BasicAuth

   You can use this option for a managed client.

   Signature

   You can use this option for a managed client.

   IDAssertion

   You can use this option for Web services acting as a client.
8. Select **Signature** to authenticate the client using the certificate used to digitally sign the request.

For more information on getting started with the Web Services Client Editor within the Assembly Toolkit , see Configuring the client security bindings using the Assembly Toolkit.

Once you have specified signature as the authentication method, you must specify how to collect the authentication information. See Configuring the client for signature authentication: collecting the authentication information for more information.

Signature authentication method:

When using the signature authentication method, the security token is generated with a `<ds:Signature>` and a `<wsse:BinarySecurityToken>` element.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

Signature token generation

The request sender generates a Signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler is specified in the `<LoginBinding>` element of the bindings file, `ibm-webservicesclient-bnd.xmi`. WebSphere Application Server provides the following callback handler implementation that can be used with the Signature authentication method:
`com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

You can add your own callback handlers that implement `javax.security.auth.callback.CallbackHandler`.

Security token validation

The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. The `<ds:Signature>` and `<wsse:BinarySecurityToken>` elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject then is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the `<LoginMapping>` element of the bindings file. There are default bindings specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImp` CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches in the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file. WebSphere Application Server provides the `system.wssecurity.Signature` default configuration entry, which is suitable for the signature authentication method.

Configuring the client for signature authentication: collecting the authentication information

This task is used to configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server. For more information on signature authentication, see Signature authentication method.

Complete the following steps to specify how the client collects the authentication information for signature authentication:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Signing Information** and click **Edit** to modify the signing key name and signing key locator. To create new signing information, click **Enable**. The certificate that is sent to login at the server is the one configured in the Signing Information section. Review the section on Key locators to understand how the signing key name maps to a key within the key locator entry.

   The following list describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core

   Canonicalization method algorithm

   The canonicalization method algorithm is used to canonicalize the SignedInfo element before it is digested as part of the signature operation.

   Digest mehod algorithm

   The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

Signature method algorithm
: The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` into the `<SignatureValue>`. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

Signing key name
: The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.

Signing key locator
: The signing key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Key locators.

8. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
9. Click **Edit** to view the Login Binding information. The login binding information displays and either select or enter the following information:

Authentication method
: The authentication method specifies the type of authentication that occurs. Select **Signature** to use signature authentication.

Token value type URI and Token value type URI local name
: When you select **Signature**, you cannot edit the **Token value type URI** and **Local name** values. These values are specifically for custom authentication types. For signature authentication, you do not need to enter any information.

Callback handler
: The callback handler specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting signature information. Enter the following callback handler for signature authentication:
`com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

: This callback handler is used because signature does not require user interaction.

Basic authentication User ID and Basic authentication Password
: Do not enter anything in the BasicAuth fields when Signature authentication is desired.

Property name and property value
: This field enables you to enter properties and name and value pairs for use by custom callback handlers. For signature authentication, you do not need to enter any information.

Important: There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the Web services security signature authentication section overrides the basic authentication information specified in the Port Qualified Name Binding Details section for authorizing the Web service.

To use the signature authentication method, you must specify the authentication method in the **Login Config** section of the Assembly Toolkit. See Configuring the client for signature authentication: specifying the method if you have not previously specified this information.

Configuring the server to handle signature authentication

This task is used to configure signature authentication at the server. Signature refers to the an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. Once a request is received by the server that contains certificate, the server needs to log in to form a credential. The credential is used for authorization. If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource. For more information on signature authentication, see Signature authentication method.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Config** section. You can select from the following options:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **LTPA**
8. Select **Signature** to authenticate the client using an X509 certificate. The certificate that is sent from the client is the certificate used for signing the message. You must be able to map this certificate to the configured user registry. For Local OS, the common name (cn) of the distinguished name (DN) is mapped to a user ID in the registry. For LDAP, you can configure multiple mapping modes:
   - EXACT_DN is the default mode that directly maps the DN of the certificate to an entry in the LDAP server.
   - CERTIFICATE_FILTER is the mode that allows the the LDAP advanced configuration to have a place to specify a filter that maps specific attributes of the certificate to specific attributes of the LDAP server.

For more information on getting started with the Web Services Editor within the Assembly Toolkit, see Configuring the server security bindings using the Assembly Toolkit.

Once you have specified how the server will handle signature authentication information, you must specify how the server validates the authentication information. See Configuring the server to validate signature authentication information for more information.

Configuring the server to validate signature authentication information

This task is used to configure signature authentication at the server. Signature refers to the an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. Once a request is received by the server that contains certificate, the server needs to log in to form a credential. The credential is used for authorization. If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource. For more information on signature authentication, see Signature authentication method.

Complete the following steps to configure the server to validate signature authentication:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.

6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog displays and either select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that will occur. Select **Signature** to use signature authentication.

   Configuration name
   This specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the signature authentication method, enter `system.wssecurity.Signature` for the JAAS login configuration name. This specification logs in with the `com.ibm.wsspi.wssecurity.auth.module.SignatureLoginModule` JAAS login module.

   Use token value type
   This determines if you want to specify a custom token type. For the default Authentication method selections, you don't need to specify this.

   URI and local name
   When you select Signature, you cannot edit the token value type URI and local name values. These values are specifically for custom authentication types. For signature authentication, you do not need to enter any information here.

   Callback handler factory class name
   This class name creates a JAAS CallbackHandler implementation that understands the following callback handlers:
   - `javax.security.auth.callback.NameCallback`
   - `javax.security.auth.callback.PasswordCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

   For any of the default Authentication methods (BasicAuth, IDAssertion, Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including signature:

   ```
   com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
   ```

   This implementation creates the correct callback handler for the default implementations.

   Callback handler factory property name and callback handler factory property value
   This field is used to specify callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For signature, you do not need to enter any properties for this field.

   Login mapping property name and login mapping property value
   This field is used to specify properties for a custom login mapping to use. For the default implementations including signature, you do not need to enter any properties for this field.

You must specify how the server will handle the signature authentication method. See Configuring the server to handle signature authentication if you have not previously specified this information.

Token type overview

A username token consists of a user name and, optionally, password information. You can include a username token directly in the `<Security>` header within the message. Binary tokens, such as X.509 certificates, Kerberos tickets,, Lightweight Third-party Authentication (LTPA) tokens, or other non-XML formats, require a special encoding for inclusion. The Web services security specification describes how to encode binary security tokens such as X.509 certificates and Kerberos tickets; and how to include opaque encrypted keys. The specification also includes extensibility mechanisms that you can use to further describe the characteristics of the credentials that are included with a message.

The proposed Web services security draft defined two types of security tokens:

- Username token
- Binary security token

A username token consists of a user name and, optionally, password information. You can include a username token directly in the `<Security>` header within the message. Binary tokens, such as X.509 certificates, Kerberos tickets,, Lightweight Third-party Authentication (LTPA) tokens, or other non-XML formats, require a special encoding for inclusion. The Web services security specification describes how to encode binary security tokens such as X.509 certificates and Kerberos tickets; and how to include opaque encrypted keys. The specification also includes extensibility mechanisms that you can use to further describe the characteristics of the credentials that are included with a message.

WebSphere Application Server, Version 5.0.2 supports user name tokens, which include both user name and password for basic authentication and user name, which are used for identity assertion. The WebSphere Application Server, Version 5.0.2 binary security token implementation supports both X.509 certificates and LTPA binary security. You can extended the implementation to generate other type of tokens. However, Kerberos tickets are not supported in WebSphere Application Server, Version 5.0.2. Each type of token is processed by a corresponding token generation and validation module. The binary token generation and validation modules are pluggable, which is based on the Java Authentication and Authorization Service (JAAS) framework. For example, arbitrary XML-based tokenformat is supported using the JAAS pluggable framework. WebSphere Application Server, Version 5.0.2 does not support an XML-based token that is used in SecurityTokenReference.

You can define the types of tokens that the message can accept in the deployment descriptor extension file, `ibm.webservices-ext.xmi`. A message receiver might support one or more types of security tokens. The following example shows that the receiver supports four types of security tokens:

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wsext=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi"
xmi:id="WsExtension_1052760331306" routerModuleName="StockQuote.war">
  <wsDescExt xmi:id="WsDescExt_1052760331306" wsDescNameLink="StockQuoteFetcher">
    <pcBinding xmi:id="PcBinding_1052760331326" pcNameLink="urn:xmltoday-delayed-quotes"
      scope="Session">
       <serverServiceConfig
         xmi:id="ServerServiceConfig_1052760331326"actorURI="myActorURI">
          <securityRequestReceiverServiceConfig
           xmi:id="SecurityRequestReceiverServiceConfig_1052760331326">
            <loginConfig xmi:id="LoginConfig_1052760331326">
              <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
              <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
              <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
              <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
           </loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

The message sender might choose one of the token types that are supported by the receiver when sending a message. You can define the type of token to be used by the sending side in the client descriptor extension file, `ibm-webservicesclient-ext.xmi`. The following example shows that the sender chooses to send a UsernameToken to the receiver:

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
mlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wscext=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi"
xmi:id="WsClientExtension_1052760331496">
<ServiceRefs xmi:id="ServiceRef_1052760331506" serviceRefLink="service/StockQuoteService">
    <portQnameBindings xmi:id="PortQnameBinding_1052760331506"
portQnameLocalNameLink="StockQuote">
      <clientServiceConfig xmi:id="ClientServiceConfig_1052760331506"
actorURI="myActorURI">
          <securityRequestSenderServiceConfig
xmi:id="SecurityRequestSenderServiceConfig_1052760331506" actor="myActorURI">
          <loginConfig xmi:id="LoginConfig_1052760331506" authMethod="BasicAuth"/>
```

Username token

You can use the UsernameToken to propagate a user name and, optionally, password information. Also, you can use this token type to carry basic authentication information. Both a user name and password are used to authenticate the message. A `UsernameToken` containing the user name is used in identity assertion, which establishes the identity of the user based on the trust relationship.
The following example shows the the syntax of the UsernameToken element:

```
<UsernameToken Id="...">
    <Username>...</Username>
    <Password Type="...">...</Password>
</UsernameToken>
```

The Web services security specification defines the following password types:

wsse:PasswordText (default)
This type is the actual password for the user name.

wsse:PasswordDigest
The type is the digest of the password for the user name. The value is a base64-encoded SHA1 hash value of the UTF8-encoded password.

WebSphere Application Server supports the default PasswordText type. However, it does not support password digest because most user registry security policies do not expose the password to the application software.

The following illustrates the use of the `<UsernameToken>` element:

```
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
    <S:Header>
            ...
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>Joe</wsse:Username>
                <wsse:Password>ILoveJava</wsse:Password>
            </wsse:UsernameToken>
        </wsse:Security>
    </S:Header>
</S:Envelope>
```

Nonce:

Nonce is a randomly generated, cryptographic token used to thwart the highjacking of username tokens used with SOAP messages. Nonce is used in conjunction with the basicauth authentication method.

Without nonce, when a username token is passed from one machine to another machine using a non-secure transport, such as HTTP, the token might be intercepted and used in a replay attack. The same key might be reused when the username token is transmitted between the client and the server, which leaves it vulnerable to attack. The username token can be high-jacked even if you use XML digital signature and XML encryption.

To help eliminate these replay attacks, the `<wsse:Nonce>` and `<wsu:Created>` elements are generated within the `<wsee: usernameToken>` element and used to validate the message. The request receiver or response receiver checks the freshness of the message to verify that difference between when the message is created and the current time falls within a specified time period. Also, WebSphere Application Server verifies that the token has not been processed already by the receiver within the specified time period. These two features are used to lessen the chance that a username token is used for a replay attack.

Configuring nonce for the application level:

Nonce is a randomly generated, cryptographic token used to thwart the highjacking of username tokens used with SOAP messages. Nonce is used in conjunction with the BasicAuth authentication method.

This task provides instructions on how to configure nonce for the application level using the WebSphere Application Server administrative console. You can configure nonce at the application level, server level, and cell level. However, you must consider the order of precedence.

The following list shows the order of precedence:
1. Application level
2. Server level
3. Cell level

If you configure nonce on the application level and the server level, the values specified for the application level take precedence over the values specified for the server level.Likewise, the values specified for the application level take precedence over the values specified for the server level and cell level.
1. Connect to administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Servers > Application Servers >***server1*.
3. Under Additional Properties, click **Web Services: Default Bindings for Web Services Security > Login Mappings > New**.
4. Specify a value, in seconds, for the **Nonce Maximum Age** field. The value specified for the **Nonce Maximum Age** field indicates how long the nonce is valid. You must specify a minimum of 300 seconds, but the value cannot exceed the number of seconds specified for the **Nonce Cache Timeout** field for either the server level or the cell level.

   You can specify the **Nonce Cache Timeout** value for the server level by completing the following steps:
   a. Click **Servers > Application Servers >***server_name*.
   b. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**.

   You can specify the **Nonce Cache Timeout** value for the cell level by clicking **Security > Web Services > Properties**.

   Important: The **Nonce Maximum Age** field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value:

   ```
   Nonce is not supported for authentication methods other than
   BasicAuth.
   ```

If you specify BasicAuth, but do not specify values for the **Nonce Maximum Age** field, the Web services security run time searches for a Nonce Maximum Age value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 300 seconds.

5. Specify a value, in seconds, for the **Nonce Clock Skew** field. The value specified for the **Nonce Clock Skew** field specifies the amount of time, in seconds, to consider when the message receiver checks the freshness of the value. Consider the following information when you set this value:
   - Difference in time between the message sender and message receiver if the clocks are unsynchronized.
   - Time needed to encrypt and transmit the message.
   - Time needed to get through network congestion.

   Important: The **Nonce Clock Skew** field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value:

   ```
   Nonce is not supported for authentication methods other than
   BasicAuth.
   ```

   If you specify BasicAuth, but do not specify values for the **Nonce Clock Skew** field, the Web services security run time searches for a Nonce Clock Skew value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 0 seconds.
6. Restart the server.

Configuring nonce for the server level:

Nonce is a randomly generated, cryptographic token used to thwart the highjacking of username tokens used with SOAP messages. Nonce is used in conjunction with the BasicAuth authentication method.

This task provides instructions on how to configure nonce for the server level using the WebSphere Application Server administrative console. You can configure nonce at the application level, server level, and cell level. However, you must consider the order of precedence. The following list shows the order of precedence:

1. Application level
2. Server level
3. Cell level

If you configure nonce on the application level and the server level, the values specified for the application level take precedence over the values specified for the server level. Likewise, the values specified for the application level take precedence over the values specified for the server level and cell level. In a WebSphere Application Server environment, you must specify values for the **Nonce Cache Timeout**, **Nonce Maximum Age**, and **Nonce Clock Skew** fields on the server level to use nonce effectively. However, in a WebSphere Application Server Network Deployment environment, these fields are optional on the server level, but required on the cell level. Complete the following steps to configure nonce on the server level:

1. Connect to administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Servers > Application Servers >***server1*.
3. Under Additional Properties, click **Web Services: Default Bindings for Web Services Security** .
4. Specify a value, in seconds, for the **Nonce Cache Timeout** field. The value specified for the **Nonce Cache Timeout** field indicates how long the nonce remains cached before it is expunged. You must

specify a minimum of 300 seconds. However, if you do not specify a value, the default is 600 seconds. This field is optional on the server level, but required on the cell level.

5. Specify a value, in seconds, for the **Nonce Maximum Age** field. The value specified for the **Nonce Maximum Age** field indicates how long the nonce is valid. You must specify a minimum of 300 seconds, but the value cannot exceed the number of seconds specified for the **Nonce Cache Timeout** field on the server level. Also, the value specified for the **Nonce Maximum Age** field must not exceed the **Nonce Maximum Age** value set on the cell level.

   You can specify the **Nonce Cache Timeout** value for the cell level by clicking **Security > Web Services > Properties**.

   This field is optional on the server level, but required on the cell level.
6. Specify a value, in seconds, for the **Nonce Clock Skew** field. The value specified for the **Nonce Clock Skew** field specifies the amount of time, in seconds, to consider when the message receiver checks the freshness of the value. Consider the following information when you set this value:
   - Difference in time between the message sender and message receiver if the clocks are unsynchronized.
   - Time needed to encrypt and transmit the message.
   - Time needed to get through network congestion.

   You must specify at least 0 seconds for the **Nonce Clock Skew** field. However, the maximum value cannot exceed the number of seconds specified in the **Nonce Maximum Age** field on the server level. If you do not specify a value, the default is 0 seconds.
7. Restart the server. If you change the **Nonce Cache Timeout** value and do not restart the server, the change is not recognized by the server.

Binary security token

The `ValueType` attribute identifies the type of the security token, for example, an LTPA token. The `EncodingType` indicates how the security token is encoded, for example, Base64Binary. The `BinarySecurityToken` element defines a security token that is binary encoded. The encoding is specified using the `EncodingType` attribute. The value type and space are specified using the `ValueType` attribute. The Web services security implementation for WebSphere Application Server, Version 5.0.2 supports both LTPA and X.509 certificate binary security tokens.
A binary security token has the following attributes that are used to interpret it:

- Value type
- Encoding type

The `ValueType` attribute identifies the type of the security token, for example, an LTPA token. The `EncodingType` indicates how the security token is encoded, for example, Base64Binary. The `BinarySecurityToken` element defines a security token that is binary encoded. The encoding is specified using the `EncodingType` attribute. The value type and space are specified using the `ValueType` attribute. The Web services security implementation for WebSphere Application Server, Version 5.0.2 supports both LTPA and X.509 certificate binary security tokens.

The following is an example of an LTPA binary security token in a Web services security message header:

```
wsse:BinarySecurityToken xmlns:ns7902342339871340177=
                              "http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
              EncodingType="wsse:Base64Binary"
              ValueType="ns7902342339871340177:LTPA">
                    MIZ6LGPt2CzXBQfio9wZTo1VotWov0NW3Za6lU5K7Li78DSnIK6iHj3hxXgrUn6p4wZI
                    8Xg26havepvmSJ8XxiACMihTJuh1t3ufsrjbFQJOqh5VcRvI+AKEaNmnEgEV65jUYAC9
                    C/iwBBWk5U/6DIk7LfXcTT0ZPAd+3D3nCS0f+6tnqMou8EG9mtMeTKccz/pJVTZjaRSo
                    msu0sewsOKfl/WPsjW0bR/2g3NaVvBy18VlTFBpUbGFVGgzHRjBKAGo+ctkl80nlVLIk
                    TUjt/XdYvEpOr6QoddGi4okjDGPyyoDxcvKZnReXww5UsoqlpfXwN4KG9as=
</wsse:BinarySecurityToken></wsse:Security></soapenv:Header>
```

As shown in the example, the token is Base64Binary encoded.

XML token

XML tokens are offered in two formats, Security Assertion Markup Language (SAML) and eXtensible rights Markup Language (XrML).

XML-based security tokens are growing in popularity. Two well-known formats are:

- Security Assertion Markup Language (SAML)
- eXtensible rights Markup Language (XrML)

The extensibility of the `<wsse:Security>` header in XML-based security tokens enables you to directly insert these security tokens into the header.

SAML assertions are attached to Web services security messages using Web Services by placing assertion elements inside the `<wsse:Security>` header. The following example illustrates a Web services security message with a SAML assertion token.

```
S:Envelope xmlns:S="...">&
  <wsse:Security xmlns:wsse="...">
      <saml:Assertion
            MajorVersion="1"
            MinorVersion="0"
            AssertionID="SecurityToken-ef375268"
               Issuer="elliotw1"
               IssueInstant="2002-07-23T11:32:05.6228146-07:00"
               xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
            ...
      </saml:Assertion>
  </wsse:Security>
 </S:Header>
 <S:Body>
 ...
 </S:Body>
</S:Envelope>
```

For more information on SAML and XrML, see Resources for learning.

Security token

A security token represents a set of claims made by a client that might include a name, password, identity, key, certificate, group, privilege, and so on.

Web services security provides a general-purpose mechanism to associate security tokens with messages for single message authentication. A specific type of security token is not required by Web services security. Web services security is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client might provide proof of identity and proof that they have a particular business certification.

A security token is embedded in the SOAP message within the SOAP header. The security token within in the SOAP header is propagated from the message sender to the intended message receiver. On the receiving side, the WebSphere Application Server security handler authenticates the security token and sets up the caller identity on the thread of execution.

Securing Web services using a pluggable token

WebSphere Application Server provides several different methods to secure your Web services; a pluggable token is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication

- Signature authentication
- Pluggable token

Complete the following steps to secure your Web services using a pluggable token:

1. Generate a security token using the JAAS CallbackHandler interface. The Web services security run time uses the Java Authentication and Authorization Service (JAAS) CallbackHandler interface as a plug-in to generate security tokens on the client side or when Web services is acting as client.
2. Configure your pluggable token. To use pluggable tokens to secure your Web services, you must configure both the client request sender and the server request receiver. You can configure your pluggable tokens using either the WebSphere Application Server administrative console or the WebSphere Application Server Toolkit. For more information, see the following topics:
   - Configuring pluggable tokens using the WebSphere Application Server Toolkit
   - Configuring pluggable tokens using the administrative console

Configuring pluggable tokens using the Assembly Toolkit

Prior to completing these steps, it is assumed that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with a Web Services for J2EE (JSR 109) enterprise application. If not, see Developing Web services to create Web services-enabled J2EE with a JSR 109 enterprise application. See either of the following topics for an introduction of how to manage Web services security binding information for the server:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

Important: The pluggable token is required for the request sender and request receiver as they are a pair. The request sender and the request receiver must match for a request to be accepted by the receiver.

You must specify the security constraints in the `ibm-webservicesclient-ext.xmi` and `ibm-webservices-ext.xmi` files for the required tokens using the Assembly Toolkit.

Complete the following steps to configure the request sender using the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Package Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab. The Web Service Client Security Extensions editor displays.
   a. Under Service References, select an existing service reference or click **Add** to create a new one.
   b. Under Port Qname Bindings, select an existing port qualified name for the selected service reference or click **Add** to create a new port name binding.
   c. Under Request Sender Configuration: Login Config, select an exiting authentication method or type in a new one in the editable list box (Lightweight Third-Party Authorization (LTPA) is a supported token generation when Web services is acting as client).
   d. Click **File > Save** to save the changes.
7. Click the **Web Services Client Binding** tab. The Web Services Client Binding editor displays.

a. Under Port Qualified Name Binding, select an existing entry or click **Add** to add a new port name binding. The Web Services Client Binding editor displays for the selected port.
b. Under Login Binding, click **Edit** or **Enable**. This Login Binding dialog box displays.
   1) In the **Authentication Method** field, enter the authentication method. The authentication method that you enter in this field must match the authentication method defined on the **Security Extension** tab for the same Web service port. This field is mandatory.
   2) (Optional) Enter the token value type information in the **URI** and **Local name** fields. These fields are ignored for the BasicAuth, Signature, and IDAssertion authentication methods, but required for other authentication methods. The token value type information is inserted into the `<wsse:BinarySecurityToken>@ValueType` element for binary security token and is used as the namespace for the XML-based token.
   3) Enter an implementation of the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface. This is a mandatory field.
   4) Enter the basic authentication information in the **User ID** and **Password** fields. The basic authentication information is passed to the construct of the CallbackHandler implementation. The usage of the basic authentication information is up to the implementation of the CallbackHandler.
   5) In the **Property** field, add name and value pairs. These pairs are passed to the construct of the CallbackHandler implementation as `java.util.Map`.
   6) Click **OK**.

   Click **Disable** under Login Binding on the **Web Services Client Port Binding** tab to remove the authentication method login binding.
c. Click **File > Save** to save the changes.

8. In the Package Explorer window, right-click the `webservices.xml` file and select **Open With > Web Services Editor**. The Web Services window displays.
   a. Click the **Security Extensions** tab. The Web Service Security Extensions editor displays.
      1) Under Web Service Description Extension, select an existing service reference or click **Add** to create a new extension.
      2) Under Port Component Binding, select an existing port qualified name of the selected service reference or click **Add** to create a new one.
      3) Under Request Receiver Service Configuration Details: Login Config, select an exiting authentication method or click **Add** and enter a new method in the Add AuthMethod field that displays. You can select multiple authentication methods for the request receiver. The security token of the incoming message is authenticated against the authentication methods in the order that they are specified in the list. Click **Remove** to remove the selected authentication method or methods.
   b. Click **File > Save** to save the changes.
   c. Click the **Bindings** tab. The Web Services Bindings editor displays.
      1) Under Web Service Description Bindings, select an existing entry or click **Add** to add a new Web services descriptor.
      2) Click the **Binding Configurations** tab. The Web Services Binding Configurations editor displays for the selected Web services descriptor.
      3) Under Request Receiver Binding Configuration Details: Login Mapping, click **Add** to create a new login mapping or click **Edit** to edit existing selected login mapping. The Login mapping dialog displays.
         a) In the **Authentication method** field, enter the authentication method. The information entered in this field must match the authentication method defined on the **Security Extensions** tab for the same Web service port. This is a mandatory field.
         b) In the **Configuration name** field, enter a JAAS login configuration name. You must define the JAAS login configuration name in the WebSphere Application Server Administrative

Console under **Security > JAAS Configuration > Application Logins**). This is a mandatory field. For more information, see Configuring Java Authentication and Authorization Service login.

c) (Optional) Select **Use Token value type** and enter the token value type information in the **URI** and **Local name** fields. This information is optional for BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is used to validate the `<wsse:BinarySecurityToken>@ValueType` element for binary security tokens and to validate the namespace of the XML-based token.

d) Under Callback Handler Factory, enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface in the **Class name** field. This field is mandatory.

e) Under Callback Handler Factory Property, click **Add** and enter the name and value pairs for the Callback Handler Factory Property. These name and value pairs are passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init()` method. The usage of these name and value pairs is determined by the CallbackHandlerFactory implementation chosen.

f) Under Login Mapping Property, click **Add** and enter the name and value pairs for the Login Mapping Property. These name and value pairs are available to the JAAS Login Module or Modules through the`com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback interface. Click **Remove** to delete selected login mapping.

g) Click **OK**.

d. Click **File > Save** to save the changes.

The previous steps define how to configure the request sender to create security tokens in the Simple Object Access Protocol (SOAP) message and the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

Once you have configured pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:

- Configuring the client for LTPA token authentication: specifying LTPA token authentication
- Configuring the client for LTPA token authentication: collecting the authentication information
- Configuring the server to handle LTPA token authentication
- Configuring the server to validate LTPA token authentication information

Configuring pluggable tokens using the administrative console

Prior to completing these steps, it is assumed that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with a Web Services for J2EE (JSR 109) enterprise application. If not, see Developing Web services to create Web services-enabled J2EE with a JSR 109 enterprise application. See either of the following topics for an introduction of how to manage Web services security binding information for the server:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

Important: The pluggable token is required for the request sender and request receiver as they are a pair. The request sender and the request receiver must match for a request to be accepted by the receiver.

Complete the following steps to configure the client-side request sender (`ibm-webservicesclient-bnd.xmi` file) or server-side request receiver (`ibm-webservices-bnd.xmi` file) using the WebSphere Application Server Administrative Console.

1. Click **Applications > Enterprise Applications >** *enterprise_application*.
2. Under Related Items, click either **EJB Modules >** *Uri* or **Web Modules >** *Uri.* . The *Uri* is the Web services-enabled module
3. Under Additional Properties, click **Web Services: Client Security Bindings** to edit the response sender binding information, if Web services is acting as client.
4. Under Response Sender Binding, click **Edit**.
5. Under Additional Properties, click **Login Binding**.
6. Select **Dedicated Login Binding** to define a new login binding or select **None** to deselect the login binding.
   a. Enter the authentication method in the **Authentication Method** field. This entry must match the authentication method defined in IBM extension deployment descriptor. The authentication method must be unique in the binding file.
   b. Enter an implementation of the JAAS `javax.security.auth.callback.CallbackHandler` interface in the **Callback Handler**field.
   c. Optional: Enter the basic authentication user ID and password in the **Basic Auth User ID** and **Basic Auth Password** fields, respectively. The basic authentication information is passed to the construct of the CallbackHandler implementation. The usage of the basic authentication information is up to the implementation of the CallbackHandler.
   d. Enter the token value type Uniform Resource Identifier (URI) and local name in the **Token Type URI** and **Token Type Local Name** fields. This information is optional for the BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is inserted into the `<wsse:BinarySecurityToken>@ValueType` for binary security token and used as the namespace of the XML based token.
   e. Click **Apply**.
7. Under Additional Properties, click **Properties**. Define the property with name and value pairs. These pairs are passed to the construct of the CallbackHandler implementation as `java.util.Map`.
8. Click **Applications > Enterprise Applications >** *enterprise_application*.
9. Under Related Items, click either **EJB Modules >** *Uri* or **Web Modules >***Uri.*. The *Uri* is the Web services-enabled module
10. Under Additional Properties, click **Web Services: Server Security Bindings** to edit the request receiver binding information.
11. Under Request Receiver Binding, click **Edit**.
12. Under Additional Properties, click **Login Mappings**.
13. Click **New** to create new login mapping. You also can edit an existing login mapping by clicking on the name or delete a login mapping by selecting the box next to the login mapping name and clicking **Remove**.
    a. Enter the authentication method in the **Authentication Method** field. This entry must match the authentication method defined in the IBM extension deployment descriptor. The authentication method must be unique in the login mapping collection of the binding file.
    b. Select a JAAS Login Configuration name from the **JAAS Configuration Name** menu. The JAAS Login Configuration must be defined in **Security > JAAS Configuration > Application Logins**. For more information, see Configuring Java Authentication and Authorization Service login.

   c. Enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface in the **Callback Handler Factory Classname** field. This is a mandatory field.
   d. Enter the token value type Uniform Resource Identifier (URI) and local name in the **Token Type URI** and **Token Type Local Name** fields. This information is optional for the BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is inserted into the `<wsse:BinarySecurityToken>@ValueType` for binary security token and used as the namespace of the XML based token.
   e. Click **Apply**.
14. Under Additional Properties, click **Properties**.
15. Click **New** and enter the name and value pairs in the **Property Name** and **Property Value** fields. These name and value pairs are available to the JAAS Login Module or Modules by `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback. These pairs are available when editing existing login mappings, but not when creating new login mappings.
16. Return to the Additional Properties heading and click **Callback Handler Factory Property**. The **Callback Handler Factory Property** option is located on the same menu where you previously clicked **Properties**.
17. Click **New** and enter the name and value pairs in the **Property Name** and **Property Value** fields. These name and value pairs are passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init()` method. The usage of these name and value pairs is up to the CallbackHandlerFactory implementation.
18. Click **Save** in the upper-left section of the administrative console.

The previous steps define how to configure the request sender to create security tokens in the Simple Object Access Protocol (SOAP) message and the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

Once you have configured pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:
- Configuring the client for LTPA token authentication: specifying LTPA token authentication
- Configuring the client for LTPA token authentication: collecting the authentication information
- Configuring the server to handle LTPA token authentication
- Configuring the server to validate LTPA token authentication information

Pluggable token support

You can extend the WebSphere Application Server login mapping mechanism to handle new types of authentication tokens.WebSphere Application Server provides a pluggable framework to generate security tokens on the sender-side of the message and to validate the security token on the receiver-side of the message. The framework is based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). Pluggable security token support provides plug-in points to support customer security token types including token generation, token validation, and mapping a client identity to a WebSphere Application Server identity that is used by the Java 2, Enterprise Edition (J2EE) authorization engine. Moreover, the pluggable token generation and validation framework allows XML-based tokens to be inserted into the Web service message header and validated on the receiver side.

You can extend the WebSphere Application Server login mapping mechanism to handle new types of authentication tokens.WebSphere Application Server provides a pluggable framework to generate security tokens on the sender-side of the message and to validate the security token on the receiver-side of the message. The framework is based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). Pluggable security token support provides plug-in points to

support customer security token types including token generation, token validation, and mapping a client identity to a WebSphere Application Server identity that is used by the Java 2, Enterprise Edition (J2EE) authorization engine. Moreover, the pluggable token generation and validation framework allows XML-based tokens to be inserted into the Web service message header and validated on the receiver side.

Users can use the javax.security.auth.callback.CallbackHandler implementation to create a new type of security token following these guidelines:

- Use a constructor that takes a user name (a string or null, if not defined), password (a char[] or null, if not defined) and `java.util.Map` (empty, if properties are not defined).
- Use handle() methods that can process the `javax.security.auth.callback.NameCallback`, `javax.security.auth.callback.PasswordCallback`, `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback`, and `com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl` implementations. If either the`javax.security.auth.callback.NameCallback` or the `javax.security.auth.callback.PasswordCallback` implementation is populated with data, then a `<wsse:UsernameToken>` element is created. Otherwise, if `om.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl` is populated, the `<wsse:BinarySecurityToken>` element is created from the `com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl` implementation. Lastly, if `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback` is populated, a XML-based token is created based on the Document Object Model (DOM) element that is returned from the XMLTokenCallback. Encode the token byte by using the security handler and not by using the javax.security.auth.callback.CallbackHandler implementation.

You can implement the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface, which is a factory for instantiating the `javax.security.auth.callback.CallbackHandler`. For your own implementation, you must provide the `javax.security.auth.callback.CallbackHandler` interface. The Web service security run time instantiates the factory implementation class and passes the authentication information from the Web services message header to the factory class through the setter methods. The Web services security run time then invokes the newCallbackHandler() method of the factory implementation class to obtain an instance of the `javax.security.auth.CallbackHandler` object. The object is passed to the JAAS login configuration.

The following is the definition of the `CallbackHandlerFactory` interface:

```
public interface com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory {
      public void setUsername(String username);
      public void setRealm(String realm);
      public void setPassword(String password);
      public void setHashMap(Map properties);
      public void setTokenByte(byte[] token);
      public void setXMLToken(Element xmlToken);
      public CallbackHandler newCallbackHandler();
```

Configuring the client for LTPA token authentication: specifying LTPA token authentication

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a Web service calls a downstream Web service, you can configure the first Web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a Web service acting as a client to a downstream Web service. In order for the downstream Web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify LTPA token as the authentication method:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section.
8. Select **LTPA** as the authentication method. For more conceptual information on LTPA authentication, see LTPA.

Once you have specified LTPA token as the authentication method, you must specify how to collect the LTPA token information. See Configuring the client for LTPA token authentication: collecting the authentication information for more information.

Configuring the client for LTPA token authentication: Collecting the authentication method information

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a Web service calls a downstream Web service, you can configure the first Web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a Web service acting as a client to a downstream Web service. In order for the downstream Web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify how to collect the LTPA token authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
8. Click **Edit** to view the login binding information and select **LTPA**. If LTPA is not already there, enter it as an option. The login binding dialog displays and either select or enter the following information:

   Authentication method

   The authentication method specifies the type of authentication that occurs. Select **LTPA** to use identity assertion.

   Token value type URI and token value type local name

   When you select **LTPA**, you must edit the **token value type URI** and the **local name** fields. These values are specifically for custom authentication types, which are authentication methods not mentioned in the specification. For the **token value type URI** field, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For the **local name** field, enter the following string: `LTPA`.

   Callback handler

   The callback handler specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the LTPA information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler` implementation for LTPA.

Basic authentication user ID and basic authentication password
For LTPA, you can leave these fields empty.

Property name and property value
For LTPA, you can leave these fields empty.

See Configuring the client for LTPA token authentication: specifying LTPA token authentication if you have not previously specified this information.

Configuring the server to handle LTPA token authentication information

This task is used to configure Lightweight Third-Party Authentication (LTPA). LTPA is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first Web service, which authenticated the originating client, to the downstream Web service. Do not attempt to configure LTPA from a pure client. Once the downstream Web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys used by both the sending and receiving servers must be the same.

Complete the following steps to specify that LTPA is authentication method. The authentication method indicated in these steps must match the authentication method specified for the client.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Configuration** section. You can select from the following options:
   - BasicAuth
   - Signature
   - ID assertion
   - LTPA
8. Select **LTPA** to authenticate the client using the LTPA token received from the request.

Once you have specified the authentication method, you must specify the information that the server must validate. See Configuring the server to validate LTPA token authentication information for more information.

Configuring the server to validate LTPA token authentication information

This task is used to configure Lightweight Third-Party Authentication (LTPA). LTPA is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first Web service, which authenticated the originating client, to the downstream Web service. Do not attempt to configure LTPA from a pure client. Once the downstream Web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys used by both the sending and receiving servers must be the same.

Complete the following steps to specify how the server must validate the LTPA token authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the Login Mapping information. The login mapping information displays and either select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that occurs. Select **LTPA** to use LTPA token authentication.

   Configuration name
   This name specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the LTPA authentication method, enter `WSLogin` for the JAAS login configuration name. This configuration understands how to validate an LTPA token.

   Use token value type
   This option determines if you want to specify a custom token type. For LTPA authentication, you must select this option because LTPA is considered a custom type. LTPA is not in the Web Services Security Specification.

   Token value type URI and local name
   If you select **Use Token value type** you must enter data into the **Token value Type URI** and **local name** fields. For URI, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For local name, enter the following string: `LTPA`

   Callback handler factory class name
   This classname creates a JAAS CallbackHandler implementation that understands the following callback handlers:

   - `javax.security.auth.callback.NameCallback`
   - `javax.security.auth.callback.PasswordCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
   - `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

   For any of the default Authentication methods (BasicAuth, IDAssertion, Signature, LTPA), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including LTPA: `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl`

   This implementation creates the correct callback handler for the default implementations.

   Callback handler factory property
   This field is used to specify callback handler properties for custom callback handler factory implementations. The default callback handler factory implementation does not need you to specify any properties. For LTPA, you do not need to enter any properties for this field.

Login mapping property
: This field is used to specify properties for a custom login mapping. For the default implementations including LTPA, you do not need to enter any properties for this field.

See Configuring the server to handle LTPA token authentication if you have not previously specified this information.

Lightweight Third Party Authentication:

When you use the lightweight third party authentication (LTPA) method, the security token generated is `<wsse:BinarySecurityToken>`. On the request sender side, the security token is generated by invoking a callback handler. On the request receiver side, the security token is validated by a Java Authentication and Authorization Service (JAAS) login module.

The token generation and token validation operations are described below.

LTPA token generation

The request sender uses a callback handler to generate an LTPA security token. The callback handler returns a security token that is inserted in the SOAP message. Specify the appropriate callback handler in the `<LoginBinding>` element of the bindings file (`ibm-webservicesclient-bnd.xmi`). The following is a callback handler implementation that can be used with the LTPA authentication method:

- `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`

You can add your own callback handlers that implement `javax.security.auth.callback.CallbackHandler`.

When using the LTPA authentication method (or any authentication method other than BasicAuth, Signature or IDAssertion), the TokenValueType attribute of the `<LoginBinding>` element in the bindings file (`ibm-webservicesclient-bnd.xmi`) must be specified. The values to use for the LTPA TokenValueType are:

- `uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"`
- `localName="LTPA"`

LTPA token validation

The request receiver retrieves the LTPA security token from the SOAP message and validates it using a JAAS login module. The security token, `<wsse:BinarySecurityToken>`, is used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. Subsequently, this Subject is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault.

The appropriate JAAS login configuration to use is specified in the bindings file `<LoginMapping>` element. There are default bindings specified in the `ws-security.xml` file, but these can be overridden using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory, ConfigName and TokenValueType. The CallbackHandlerFactory specifies the name of a class to use to create the JAAS CallbackHandler object. A CallbackHandlerFactory implementation is provided (`com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl`). The ConfigName specifies a JAAS configuration name entry. The Web services security run time first searches the `security.xml` file for a matching entry and if a matching entry is not found, the run time searches the `wsjaas.conf` file. A default configuration entry suitable for the LTPA authentication method is provided (`WSLogin`). There is an appropriate TokenValueType element in the LTPA LoginMapping section of the default `ws-security.xml` file.

Tuning Web services based on Web Services for J2EE

Performance considerations are the same for Web services applications that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) and regular J2EE applications. See Tuning performance for more information about analyzing and tuning J2EE applications.

You can use the Performance Monitoring Infrastructure (PMI) to measure the time required to process Web services requests. To monitor Web services application performance:

1. Enable PMI services in application server through the administrative console. Select the Web Service module, named webServicesModule, in step 7.
2. Monitor performance with Tivoli Performance Viewer In the left-hand pane of the performance view, expand the host and server and select **Web Services**. Run the Web services client application.

Measurements are available for the following items:

- Number of Web services loaded by the application server
- Number of requests received
- Number of requests dispatched to an implementation bean
- Number of requests dispatched with successful replies
- Average time in milliseconds between receiving the request and returning the reply
- Average time in milliseconds between receiving the request and dispatching it to the bean
- Average time in milliseconds between dispatch and receipt of reply from the bean
- Average time in milliseconds between receipt of reply from bean to return of result to client
- Average size of request and reply
- Average size of request
- Average size of reply

Troubleshooting Web services based on Web Services for J2EE

Select the Web services topic area you want to troubleshoot:

- Command-line tools
- Java compiler errors
- Runtime errors and exceptions
- Client runtime errors and exceptions
- Serialization or deserialization errors

Troubleshooting command-line tools for Web services based on Web Services for J2EE

This topic discusses troubleshooting command-line tools for Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

WSDL2Java command-line tool

Emitter failure error occurs when running the WSDL2Java command on a WSDL document containing a JMS-style endpoint URL

If you run the **WSDL2Java** command-line tool on a WSDL document that contains a JMS-style endpoint URL , for example `jms:/...`, the `urlprotocols.jar` file that contains the custom protocol handler for the JMS protocol must be in the CLASSPATH. The error **`WSWS3099E: Error: Emitter failure. Invalid endpoint address in port <x> in service <y>: <jms-url-string>`** can be avoided by making sure the `urlprotocols.jar` file is in the CLASSPATH.

To add the `urlprotocols.jar` file to the CLASSPATH:

On Windows platforms, edit the `install_root\bin\setupCmdLine.bat` and locate the line which sets the *WAS_CLASSPATH* environment variable. Add `%install_root%\lib\urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

On UNIX platforms, edit the `install_root/bin/setupCmdLine.sh` file and add `$install_root/lib/urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

Make sure to use the proper deliminator character for your platform, for example, use a semi-colon (;) for Windows platforms and a colon (:) for UNIX platforms.

Troubleshooting compiled bindings for Web services based on Web Services for J2EE

This topic discusses troubleshooting compiled bindings of Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

Context root not recognized when mapping the default XML namespace to a Java package

When you map the default XML namespace to a Java package the context root is not recognized. If two namespaces are the same up to the first slash, they are mapped to the same Java package. For example, the XML namespaces `http://www.ibm.com/foo` and `http://www.ibm.com/bar` both map to the Java package `www.ibm.com`. Use the -NStoPkg option of the **Java2WSDL** command to specify the package for the fully qualified namespace.

Java code to WSDL mapping cannot be reversed back to the original Java code

If you find that a WSDL file you created with the **Java2WSDL** command-line tool cannot be compiled when regenerated into Java code using the **WSDL2Java** command-line tool it is because the Java API for XML-based remote procedure call (JAX-RPC) mapping from Java code to WSDL is not reversible back to the original Java code.

To troubleshoot this problem review the WSDL file that was generated by the **Java2WSDL** tool using the information in Mapping between Java, WSDL and XML and the JAX-RPC specification available through Web services: Resources for learning. Use this information to determine which elements in the WSDL file are causing the problem. You can modify the WSDL file, or the original Java interface used to generate the WSDL file, and run the **Java2WSDL** command again.

Troubleshooting the run time of Web services based on Web Services for J2EE

This topic discusses troubleshooting the run time of Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can troubleshoot run time errors and exceptions as follows:

- Trace SOAP messages
- Trace the components of Web services based Web Services for J2EE

Tracing SOAP messages

This topic discusses tracing SOAP messages that request Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can trace the SOAP messages exchanged between a client and the server using the **TCPMon** command tool. The **TCPMon** command redirects messages from one port to another and records them.

The WebSphere Application Server listens on port 9080. To trace messages sent to the application server, the **TCPMon** command is configured to listen on port 9088 and redirect them to 9080. The client is redirected to use port 9088 to access the Web service.

Redirecting an application client to a different port is most easily done by changing the SOAP address in the client's Web Services Description Language (WSDL) file to use port 9088 and then running the **wsdeploy** command-line tool on the client enterprise archive (EAR) file to regenerate the service implementation.

You should confirm that the server providing the Web service is running. The following task is performed on the machine providing the Web service.

To trace SOAP messages in Web services:
1. Set up a development and unmanaged client execution environment for Web services based on Web Services for J2EE
2. Run the **java -Djava.ext.dirs=%WAS_EXT_DIRS%** command. A window labeled TCPMonitor displays.
3. Configure the TCPMonitor to listen on port 9088 and forward messages to port 9080.
   a. In the **Listen Port #** field, enter `9088`.
   b. Click **Listener**
   c. In the **TargetHostname** field, enter `localhost`.
   d. In the **Target Port #** field, enter `9080`.
   e. Click **Add**.
   f. Click on the **Port 9088** tab that displays on the top of the page.

The messages exchanged between the client and server appear in the TCPMonitor Request and Response pane.

Save the message data and analyze it.

Tracing Web services components based on Web Services for J2EE

The following are tasks in which you can enable trace for Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.
1. Enable trace for a Web services unmanaged client.
   a. Create a trace properties file by copying *%install_root%*`\WebSphere\AppServer\properties\TraceSettings.properties` file to the same directory as your client application Java archive (JAR) file.
   b. Edit the properties file and change the value of `traceFileName` to output the trace data. For example, `traceFileName=c:\\temp\\myAppClient.trc`.
   c. Edit the properties file to remove `com.ibm.ejs.ras.*=all=enabled` and add `com.ibm.ws.webservices.*=all=enabled`.
   d. Add the option `-DtraceSettingsFile=<`*trace_properties_file*`>` to the Java command- line used to run the client, where *trace_properties_file* represents the name of the properties file created in steps 1-2. For example, **java -DtraceSettingsFile=TraceSettings.properties myApp.myAppMainClass**.
2. Enable trace for a Web services managed client.
   a. Invoke the **launchClient** command-line tool with the following options: -CCtrace=com.ibm.ws.webservices.*=all=enabled-CCtracefile=traceFileName For example:

      ***%install_root%*`\bin\launchClient MyAppClient.ear-`**
      CCtrace=com.ibm.ws.webservices.*=all=enabled -CCtracefile=myAppClient.trc

   See launchClient tool for more information.

3. Enable trace for a Web Services for J2EE server application.
   a. Start WebSphere Application Server.
   b. Open the administrative console.
   a. Click **Servers** >**Application Servers** > *server*.
   a. Click **Diagnostic Trace Service**.
   a. In the **Trace Specification** field, delete the text ***=all=enabled** and add **com.ibm.ws.webservices.*=all=enabled**.
   b. Click **Save** and **Apply**.

   For more information see Enabling trace.

Troubleshooting the run time for a Web services client based on Web Services for J2EE

This topic discusses troubleshooting Web services clients that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

Malformed URL exception displays when running a client that uses a JMS-style endpoint URL

If you are using the **launchClient** command to run a managed or unmanaged client that uses a JMS-style endpoint URL, the `urlprotocols.jar` file that contains the custom protocol handler for the JMS protocol must be in the CLASSPATH. The malformed URL exception can be avoided by making sure the `urlprotocols.jar` file is in the CLASSPATH.

To add the `urlprotocols.jar` file to the CLASSPATH:

On Windows platforms, edit the *`install_root`*`\bin\setupCmdLine.bat` and locate the line which sets the *WAS_CLASSPATH* environment variable. Add `%`*`install_root`*`%\lib\urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

On UNIX platforms, edit the *`install_root`*`/bin/setupCmdLine.sh` file and add `$`*`install_root`*`/lib/urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

Troubleshooting serialization and deserializaton in Web services based on Web Services for J2EE

The following are problems you might encounter performing serialization and deserialization in Web services that are developed and implemented based on the Web Services for Java 2 platform Enterprise Edition (J2EE) specification.

Time zone information in deserialized java.util.Calendar is not as expected

When the client and server are based on Java code and a `java.util.Calendar` is received, the time zone in the received `java.util.Calendar` instance might be different from that of the `java.util.Calendar` instance that was sent.

This occurs because `java.util.Calendar` is encoded as an xsd:dateTime for transmission. An xsd:dateTime is required to encode the correct time (base time plus or minus a time zone offset), but is not required to preserve locale information, including the original time zone.

The fact that the time zone for the current locale is not preserved needs to be accounted for when comparing Calendar instances. The `java.util.Calendar` class equals method checks that the time zones are the same when determining equality. Since the time zone in a deserialized Calendar instance might not match the current locale, the before and after comparison methods should be used to test that two Calendars refer to the same date and time as shown below:

```
java.util.Calendar c1 = ...// Date and time in time zone 1
java.util.Calendar c2 = ...// Same date and equivalent time, but in time zone 2

// c1 and c2 are not equal because their time zones are different
if (c1.equals (c2)) System.out.println("c1 and c2 are equal");

// but c1 and c2 do compare as "not before and not after" since they represent
the same date and time
if (!c1.after(c2) & !c1.before(c2) {
   System.out.println("c1 and c2 are equivalent");
}
```

Mixing Web services client and server bindings causes exceptions

Web Services for J2EE and Java API for XML-based remote procedure call (JAX-RPC) do not support ″round-trip″ mapping between Java code and a Web Services Description Language (WSDL) document for all Java types. For example, you cannot turn (serialize) a Java Date into XML code and then turn it back (deserialize) into a Java Date. It deserializes as Java Calendar.

If you have a Java implementation that you create a WSDL document from, and you generate client bindings from the WSDL document, the client classes can be different from the server classes even though the client classes have the same package and class names. The Web service client classes must be kept separate from the Web service server classes. For example, do not place the Web service server bindings classes in a utility Java archive (JAR) file and then include a Web service client JAR file that references the same utility JAR file.

If you do not keep the Web service client and server classes separate, a variety of exceptions can occur, depending on the Java classes used. The following is a sample stack trace error that can occur:

```
com.ibm.ws.webservices.engine.PivotHandlerWrapper TRAS0014I:
The following exception was loggedjava.lang.NoSuchMethodError:
 com.ibm.wssvt.acme.websvcs.ExtWSPolicyData: method getStartDate()Ljava/util/Date;
 not found
   at com.ibm.wssvt.acme.websvcs.ExtWSPolicyData_Ser.addElements(
     ExtWSPolicyData_Ser.java: 210)
   at com.ibm.wssvt.acme.websvcs.ExtWSPolicyData_Ser.serialize (
     ExtWSPolicyData_Swer.java:29)
   at com.ibm.ws.webservices.engine.encoding.SerializationContextImpl.serializeActual (
     SerializationContextImpl.java 719)
   at com.ibm.ws.webservices.engine.encoding.SerializationContextImpl.serialize (
     SerializationContextImpl.java: 463)
```

The problem is caused by using an interface like the following for the Service Endpoint Interface in the service implementation:

```
package server:
public interface Test_SEI extends java.rmi.Remote {
 public java.util.Calendar getCalendar () throws java.rmi.RemoteException;
 public java.util.Date getDate() throws java.rmi.RemoteException;
}
```

When this interface is compiled and run through the **Java2WSDL** command-line tool, the WSDL document maps the methods as follows:

```
<wsdl:message name="getDateResponse">
 <wsdl:part name="getDateReturn" type="xsd:dateTime"/>
</wsdl:message>

<wsdl:message name="getCalendarResponse">
 <wsdl:part name="getCalendarReturn" type="xsd:dateTime"/>
</wsdl:message>
```

The JAX-RPC mapping implemented by the **Java2WSDL** tool has mapped both java.util.Date and java.util.Calendar to the XML type xsd:dateTime. The next step is to use the generated WSDL file to create a client for the Web service. When you run the **WSDL2Java** command-line tool on the generated WSDL, the generated classes include a different version of server.Test_SEI, for example:

```
package server;
public interface Test_SEI extends java.rmi.Remote {
 public java.util.Calendar getCalendar() throws java.rmi.RemoteException;
 public java.util.Date getDate() throws java.rmi.RemoteException;
}
```

Note: The client version of the `service.Test_SEI` interface is different from the server version in that both `getCalendar` and `getDate` methods return `java.util.Calendar`. The serialization and deserilazation code that the client expects is the client version of the SEI. If the server version inadvertently appears in the client's CLASSPATH, at either compilation or execution time, an exception occurs.

In addition to the`NoSuchMethod` error, the `IncompatibleClassChangeError` and `ClassCastException` can occur, however, almost any run-time exception can occur. The best practice is to be diligent about separating client Web services bindings classes from server Web services bindings classes. The client bindings classes and server bindings classes should never be placed in the same module and, if they are in the same application, should not have bindings classes in utility JAR files that are shared between modules.

Frequently asked questions about Web services based on Web Services for J2EE

This topic presents frequently asked questions about Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

- What IBM development tools work with Web Services for J2EE?
- Is Web Services for J2EE part of the J2EE specification?
- What is the relationship between Web Services for J2EE and the Web Service Invocation Framework (WSIF)?
- What is the relationship between Apache SOAP 2.3 and Web Services for J2EE?
- What is the relationship between the Apache Axis component of the Web services technology preview available with WebSphere Application Server 5.0 and Web Services for J2EE?
- What standards does the Web Services for J2EE component of WebSphere Application Server 5.0. support?
- Does Web Services for J2EE interoperate with other SOAP implementations, like .NET?
- Why can I not use a JavaBean to implement a SOAP Java Messaging Service (JMS) service?
- Does the SOAP JMS support interoperate with other vendors?
- How does two-way messaging with SOAP JMS work? Can it support multiple clients making simultaneous requests?

What IBM development tools work with Web Services for J2EE?

WebSphere Studio Application Developer Version 5.1 and the Assembly Toolkit Version 5.1 both support the use of Web Services for J2EE. The Application Assembly Tool, included with Websphere Application Server, and Websphere Studio Application Developer versions earlier than Version 5.01, do not support Web Services for J2EE.

Is Web Services for J2EE part of the J2EE specification?

For WebSphere Application Server 5.0.2, the Web Services for J2EE Version 1.0 specification is an addition to J2EE 1.3. J2EE 1.4 requires support for Web Services for J2EE Version 1.1. There are minor differences between the J2EE 1.3 Version (JSR-109 Version 1.0) and the J2EE 1.4 Version (JSR-109 Version 1.1).

What is the relationship between Web Services for J2EE and the Web Service Invocation Framework (WSIF)?

Web Services for J2EE and WSIF represent two different programming models for accessing Web services. Web Services for J2EE is standard, Java-centric, and more statically bound to WSDL documents due to the use of generated stubs. WSIF directly models Web Services Description Language (WSDL) documents. WSIF is more suitable when dynamically interpreting WSDL. Future versions of WebSphere Application server will leverage both technologies to achieve dynamic, high performing standards-based Web services implementations.

What is the relationship between Apache SOAP 2.3 and Web Services for J2EE?

Apache SOAP shipped with WebSphere Application Server Versions 4.0 and 5.0. It continues to co-exist with Web Services for J2EE. Apache SOAP is a proprietary API and applications written for it are not portable to other SOAP implementations. Applications written for Web Services for J2EE should be portable to any vendor's implementation that supports Web Services for J2EE.

What is the relationship between the Apache Axis component of the Web services technology preview available with WebSphere Application Server 5.0 and Web Services for J2EE?

The Web services technology preview leveraged the work that IBM contributed to the Apache Axis code base. The Web Services for J2EE support included with WebSphere Application Server 5.0.2 is derived from Apache Axis, but has diverged and contains many IBM-specific features to enhance performance, scalability, reliability, interoperability, and integration with the WebSphere Application Server.

What standards does the Web Services for J2EE component of WebSphere Application Server 5.0. support?

The following standards are supported by the Web Services for J2EE component of WebSphere Application Server 5.0:
- SOAP Version 1.1
- Web Services Description Language (WSDL) Version 1.1
- Web Services for J2EE (JSR-109) Version 1.0
- Java API for XML-Based RPC (JAX-RPC) Version 1.0
- SOAP with attachments API for Java (SAAJ) Version 1.1

Does Web Services for J2EE interoperate with other SOAP implementations, like .NET?

WebSphere Application Server Version 5.0.2 and Version 5.1 support Web services that are consistent with the the WS-I Basic Profile 1.0, and should interoperate with any other vendor conforming to this specification.

Why can I not use a Java bean to implement a SOAP Java Messaging Service (JMS) service?

The SOAP JMS support uses Message Driven Beans (MDB) to implement the JMS endpoint. MDBs can only be used in the EJB container and delegate to an enterprise bean. If you want to use a Java bean instead of an enterprise bean to implement the service endpoint, you must create a ″facade″ enterprise bean that delegates to the Java bean.

Does the SOAP JMS support interoperate with other vendors?

No. There is currently no specification for SOAP JMS, therefore each vendor chooses its own implementation technique.

How does two-way messaging with SOAP JMS work? Can it support multiple clients making simultaneous requests?

Before a client issues a two-way request, it creates a temporary JMS queue to receive the response. This temporary queue is specified as the **replyTo** destination in the outgoing JMS request message. After the server processes the request, it directs the response to the **replyTo** destination specified in the request message. The client deletes the temporary queue after the response has been received. The server is able to handle simultaneous requests from multiple clients since each incoming request message contains the destination to which the reply should be sent.

Web services: Resources for learning

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

- Web services overview

  Including the WebSphere Version 5 Web Services Handbook
- Developing Web services:

  Including developing Web services based on the Java 2 platform, Enterprise Edition (J2EE) and Java API for XML-based remote procedure call (JAX-RPC) specifications.
- Web services gateway

  Including an overview of the gateway and information about the WebSphere Application Server Edge components that are used for distributed gateway deployment.
- Universal Description Discovery and Integration (UDDI)

  Including an overview about UDDI and information about the UDDI Java API.
- Web Services Invocation Framework (WSIF)

  A look into the Apache Software Foundation and its maintenance of WSIF.
- SOAP

  Including an overview about SOAP and information about the SOAP syntax and processing rules.
- Security

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

  Includes WebSphere Application Server Samples Gallery and Samples Central for Web services gateway, UDDI and WSIF.
- Other references

The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

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

Web services overview

- WebSphere Version 5 Web Services Handbook

  This IBM Redbook describes the new concept of Web services from various perspectives. It presents the major building blocks Web services rely on. Well-defined standards and new concepts are presented and discussed.
- IBM Web Services architecture debuts

  Introducing IBM Web services, a distributed software architecture of service components. This brief overview and in-depth interview on IBM DeveloperWorks cover the fundamental concepts of Web services architecture and what they mean for developers. The interview with IBM professional Rod

Smith explores which types of developers Web services targets, how Web services reduces development time, what developers could be doing with Web services now, and takes a glance at the economics of dynamically discoverable services.

- 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, they are not without their problems. Outlined are the difficulties developers face in creating a truly workable distributed system of Web services. This article also outlines author Graham Glass' plan for building peer-to-peer Web applications.

Developing Web services

- JSR 109: Implementing Enterprise Web services

  This document describes the Java 2 platform, Enterprise Edition (J2EE) specification.
- Java API for XML-based RPC (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.
- 5.1 + A developer introduction to JAX-RPC, Part 1: Learn the ins and outs of the JAX-RPC type-mapping system. The Java API for XML-based Remote Procedure Calls (JAX-RPC) is an important step forward in the quest for Web services interoperability. This IBM developerWorks article explains the mapping between WSDL/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.
- 5.1 + A developer introduction to JAX-RPC, Part 2: Mine the JAX-RPC specification to improve Web service interoperability. This IBM developerWorks 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.
- 5.1 + 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 to illustrate their use. The article is intended to give developers a good grasp of how to use JAX-RPC 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.

Web services gateway

- The IBM Web services gateway: Technical Overview. A different version of the gateway is available as a component of a product called IBM WebSphere Business Connection. This brief technical summary from WebSphere Business Connection applies equally well to the version of the gateway in WebSphere Application Server.
- Information center for WebSphere Application Server Edge components. This information center contains a library of PDF online books covering all aspects of the WebSphere Application Server Edge components. Distributed gateway deployment builds upon the load-balancing capabilities of these components.

UDDI

- Universal Description, Discovery and Integration

  This article is a detailed overview of Universal Description, Discovery and Integration (UDDI).
- UDDI4J: Matchmaking for Web services

  Reviewed in this article are the basics of UDDI, the Java API to UDDI, and how you can use this technology to start building, testing, and deploying your own Web services.

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 has been donated by IBM to the Apache Software Foundation, and is maintained here as an Apache project.

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, in a way that enables 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 quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used 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 encyrption technologies.
- JSR 55:Certification Path API

  This document provides a short description of the certification path API.
- XML-Signature Syntax and Processing

  This document specifies XML digital signature processing rules and syntax. XML signatures provide integrity, message authentication, or signer authentication services for data of any type, whether 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's 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 were encrypted before signing, and must not be decrypted, and those that were 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

  A major area of gateway security is based upon this emerging standard.
- 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 IBM WebSphere Developer Domain Web site:
  - The Web services gateway.
  - The IBM private UDDI registry.
  - The Web Services Invocation Framework (WSIF).

Other references
- 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.
- Web services insider, Part 1: Reflections on SOAP

  What is the current state of the *Web services revolution*? Find out at this Web site that features the column *Web services insider, Part 1*. The author answers this question by reviewing the tools and technologies that have emerged over the past year, highlighting their differences and similarities.
- The Web services insider, Part 2: A summary of the W3C Web Services Workshop

  This is a brief summary of a W3C Web services workshop.

Chapter 8. Web Services Invocation Framework (WSIF): Enabling Web services

The Web Services Invocation Framework (WSIF) is a WSDL-oriented Java API. You use this API to invoke Web services dynamically, regardless of the service implementation format (for example enterprise bean (EJB)) or the service access mechanism (for example Java Messaging 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:
- 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 Messaging 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. They can be as small as a simple Web service, or as large as the Web services gateway. 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.

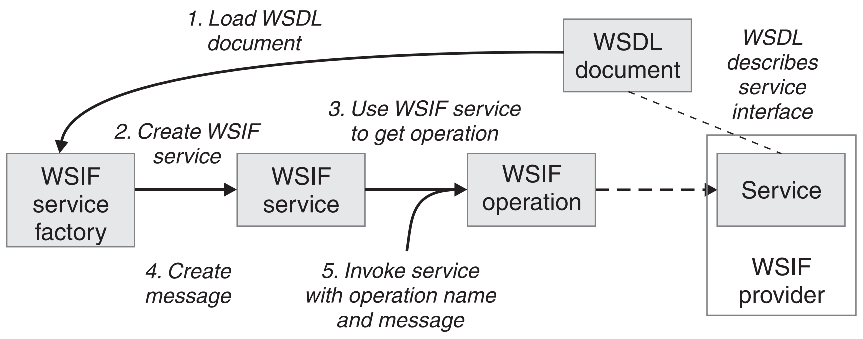
WSIF provides the following features:
- An API that provides binding-independent access to any Web service.
- A close relationship with WSDL, so it can invoke any service that you can describe in WSDL.
- A stubless and completely dynamic invocation of a Web service.
- The capability to plug a new or 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 architecture

The Web Services Invocation Framework (WSIF) architecture is shown in the figure.



The components of this architecture include:

WSDL document
: The Web service WSDL document contains the location of the Web service. The binding document defines the protocol and format for operations and messages defined by a particular portType.

WSIF service
: The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation. For more information, see Finding a port factory or service

WSIF operation
: The run-time representation of an operation, called WSIFOperation is responsible for invoking a service based on a particular binding. For more information, see WSIF API reference: Using ports.

WSIF provider
: A WSIF provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol. WSIF includes SOAP providers, JMS providers, Java providers and EJB providers. For more information, see Using the WSIF providers.

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.

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 Messaging 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.

Using WSIF to invoke Web services

You invoke a Web service dynamically by using the WSIF API directly.

You only specify the location of the WSDL file for the service, the name of the operation to invoke, and any operation arguments. All the information needed to access the Web service (the abstract interface, the binding, and the service endpoint) is available through the WSDL.

This kind of invocation does not require stub classes and does not need a separate compilation cycle.

More information on using the Web Services Invocation Framework (WSIF) to invoke Web services is provided in the following topics:

- Using the WSIF providers.
- Developing a WSIF service.
- Using complex types.
- Using the Java Naming and Directory Interface (JNDI).
- Passing SOAP messages with attachments using WSIF.
- Interacting with the J2EE container in WebSphere Application Server.
- Running WSIF as a client.

Using the WSIF providers

A Web Services Invocation Framework (WSIF) provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol.

Providers implement the interface between the WSIF API and the actual implementation of a service. Providers are pluggable within the WSIF framework, and are registered according to the namespace of the WSDL extension that they implement. Some providers use the Java 2 platform, Enterprise Edition (J2EE) programming model to utilize J2EE services. If a provider is available, but its required class libraries are not, then the provider is disabled.

WebSphere Application Server includes the following WSIF providers:
- SOAP (over HTTP) provider.
- JMS providers (SOAP over JMS, and native JMS).
- Java provider.
- EJB provider.

Using the SOAP provider

The SOAP provider allows WSIF stubs and dynamic clients to invoke SOAP services.

The Web Services Invocation Framework (WSIF) SOAP provider supports SOAP 1.1 over HTTP. The SOAP provider uses Apache SOAP 2.3 for parsing and creating SOAP messages, but it is not limited to invoking services from Apache SOAP.

The WSIF SOAP provider supports:
- SOAP-ENC encoding.
- RPC style and Document style SOAP messages.
- SOAP messages with attachments.

The SOAP provider is not transactional.

If you have a Web service that you expect multiple clients to use connecting over SOAP, then before you deploy the service you must set up your application deployment descriptor file `dds.xml` to handle multiple connections correctly. For more information, see WSIF troubleshooting tips.

For an example of the sort of code changes that need to be made in the WSDL file for a SOAP provider, see the following topics:
- The SOAP over JMS provider - writing the WSDL extension.
- SOAP messages with attachments - Writing the WSDL extensions.

Using the JMS providers

The JMS providers enable a WSIF service to be invoked through JMS.

The Java Messaging Service (JMS) is an API for transport technology. The mapping to a JMS destination is defined during deployment and maintained by the container.

The JMS destination endpoint for a Web service can be realized in any of the following ways:
- The JMS destination for the queue can be the Web service implementation.

- The JMS destination can be (but is not required to be) associated with a message-driven bean by the EJB container, thereby allowing the message-driven bean to be the Web service implementation.
- (For SOAP over JMS) The JMS destination can unwrap the JMS message and route the SOAP message to a Web service that is implemented as a stateless session bean.

The JMS destination endpoint must respect the interaction model expected by the client and defined by the WSDL. It must return a response if one is required.

When the JMS destination endpoint creates the JMS response message the following rules must be followed:
- The response message must be sent to `JMSReplyTo` from the incoming request.
- The `JMSCorrelationID` value of the response message must be set to the `JMSMessageID` value from the request message.
- The response must be sent with a `deliveryMode` value equal to the `JMSDeliveryMode` value of the request message.
- The response must be sent with a `priority` value equal to the `JMSPriority` value of the request message.
- The `timetolive/JMSExpiration` value must be set to a value that equals the `JMSExpiration` value of the request message.

The client does not see any of these headers. The container receives the JMS message and (for SOAP over JMS) removes the SOAP message to send to the client.

See also the following topics:
- Using the SOAP over JMS provider
- Using the native JMS provider
- The JMS providers - Configuring the client and server

Using the SOAP over JMS provider:

For information on working with the Java Messaging Service (JMS) API, see Using the JMS providers.

The SOAP message, including the SOAP envelope, is wrapped with a JMS message and put on the appropriate queue. The container receives the JMS message and removes the SOAP message to send to the client.

For detailed implementation information, see the following topics:
- The SOAP over JMS provider - writing the WSDL extension
- The JMS providers - Configuring the client and server

The SOAP over JMS provider - Writing the WSDL extension:

If a SOAP message contains only XML, then it can be carried on the Java Messaging Service (JMS) transport with the JMS message body type **TextMessage**.

The WSDL binding extension for SOAP over JMS varies only slightly from the SOAP over HTTP binding.

Selecting the SOAP over JMS binding

You set the `transport` attribute of the `<soap:binding>` tag to indicate that JMS is used. If you also set the `style` attribute to `rpc` (Remote Procedure Call), then the Web Services Invocation Framework (WSIF) assumes that an operation is invoked on the Web service endpoint:

```
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/jms"/>
```

Setting the JMS address

For SOAP over JMS, the `<wsdl:port>` tag must contain a `<jms:address>` element. This element provides the information required for a client to connect correctly to the Web service using the

JMS programming model. Typically, it is the stubs generated to support the SOAP over JMS binding that act as the JMS client. Alternatively, the Web service client can use the JMS programming model directly.

The `<jms:address>` element takes this form:

```
<jms:address

    destinationStyle="queue"
    jmsVendorURI="http://ibm.com/ns/mqseries"?
    initialContextFactory="com.ibm.NamingFactory"?
    jndiProviderURL="iiop://something:900/wherever"?
    jndiConnectionFactoryName="orange"
    jndiDestinationName="fred"
/>
```

where attributes marked with a question mark (?) are optional.

The optional `jmsVendorURI` attribute is a string that uniquely identifies the JMS implementation. WSIF ignores this URI, which is used by the client developer and perhaps the client implementation to determine if it has access to the correct JMS provider in the client run-time.

The optional attributes `initialContextFactory` and `jndiProviderURL` can only be omitted if the run-time has a default Java Naming and Directory Interface (JNDI) provider configured.

The `jndiConnectionFactoryName` attribute gives the name of a JMS ConnectionFactory object, which can be looked up within the JNDI context given by the jndiContext attribute. This ConnectionFactory object is used to create a JMS connection to the JMS provider instance that owns the queue. In a simple configuration, the same ConnectionFactory object is used by the server message listener and by the clients. However the server and the clients can use different ConnectionFactory objects, provided that they all create connections to the same JMS provider instance.

Setting the JMS headers and properties

You use the `<jms:property>` tag to set the JMS headers and properties. This tag maps either a message part, or a literal value, into a JMS property:

```
<jms:property name="Priority" {part="requestPriority" | value="fixedValue"}/>
```

If the `<jms:property>` has a literal value, then it can also be nested within the `<jms:address>` tag:

```
<jms:property name="Priority" value="fixedValue" />
```

This form of the `<jms:property>` tag is also used in the native JMS binding.

Here is an example of a WSDL that defines a SOAP over JMS binding:

```
<!-- Example: SOAP over JMS Text Message -->

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        name="StockQuoteInterfaceDefinitions"
        targetNamespace="urn:StockQuoteInterface"
        xmlns:tns="urn:StockQuoteInterface"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <wsdl:message name="GetQuoteInput">
        <part name="symbol" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="GetQuoteOutput">
        <part name="value" type="xsd:float"/>
    </wsdl:message>
    <wsdl:portType name="StockQuoteInterface">
        <wsdl:operation name="GetQuote">
            <wsdl:input message="tns:GetQuoteInput"/>
            <wsdl:output message="tns:GetQuoteOutput"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="StockQuoteSoapJMSBinding" type="tns:StockQuoteInterface">
        <soap:binding style="rpc"
                 transport="http://schemas.xmlsoap.org/soap/jms"/>
        <wsdl:operation name="GetQuote">
            <soap:operation soapAction="urn:StockQuoteInterface#GetQuote"/>
            <wsdl:input>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuoteServicePort"
                      binding="sqi:StockQuoteSoapJMSBinding">
            <jms:address destinationStyle="queue"
                     jndiConnectionFactoryName="myQCF"
                     jndiDestinationName="myQ"
                     initialContextFactory="com.ibm.NamingFactory"
                     jndiProviderURL="iiop://something:900/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Using the native JMS provider:

Using the native JMS provider, WSIF clients can treat a JMS destination as a Web service.

For information on working with the Java Messaging Service (JMS) API, see Using the JMS providers.

For detailed implementation information, see the following topics:
- The native JMS provider - Writing the WSDL extension
- The JMS providers - Configuring the client and server

The native JMS provider - Writing the WSDL extension:

The WSDL extensions for the Java Messaging Service (JMS) are identified with the namespace prefix `jms`. For example, `<jms:binding>`.

Operations

The supported operations are either one-way operations (`send` for JMS point-to-point messaging, or `publish` for JMS publish and subscribe messaging) or request-response operations (`send` and `receive` for JMS point-to-point messaging). The WSDL operations therefore specify either an input message only, or an input and an output message.

Fault messages

Operations that describe message interfaces with a native JMS binding do not have fault messages. No assumptions are made about the message schema or the semantics of message properties, therefore no distinction can be made between output and fault messages.

Setting the JMS message body type

You use the `<jms:binding>` extension to specify the JMS message body type:

```
<wsdl:binding ... >
   <jms:binding type="messageBodyType" />
    ...
</wsdl:binding>
```

where *messageBodyType* is either `ObjectMessage` or `TextMessage`.

Specifying the parts to use for the input and output messages

For JMS text messages and JMS object messages created from one or more WSDL message parts, you use the `<jms:input>` and `<jms:output>` extensions to specify the message parts to use for the JMS messages:

```
<wsdl:input ... >
   <jms:input parts="part1 part2 ..." />
</wsdl:input>

<wsdl:output ... >
   <jms:output parts="part1 part2 ..." />
</wsdl:output>
```

In the next example, the WSDL message has just one part that contains the complete message body. This message body might result from a mapping of some other representation (see **Mapping data types**).

```
<wsdl:input ... >
   <jms:input parts="part1" />
</wsdl:input>
```

If no parts are defined, then all the message parts are used.

Mapping data types

You use the `<format>` extensions to map data types:

```
<wsdl:binding ... >
   <jms:binding type="..." />

   <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="..." formatType="targetType"/>
   </format:typemapping>
   ...
</wsdl:binding>
```

The value of *targetType* is dependent on the JMS message body type (see **Setting the JMS message body type**). For JMS object messages, the target data type implements the java.io.Serializable class. For JMS text messages, the target data type is always java.lang.String.

The `<format>` extensions are also used in other bindings that deal with Java interfaces.

Setting the JMS headers and properties

JMS does not make assumptions about message headers. For example, if the JMS provider is MQSeries then each JMS message carries an RFH2 header. However you can access data in this message header indirectly, by getting and setting JMS message properties.

When you want your application to pass a property into the Web Services Invocation Framework (WSIF) as a part on the WSIF message, you use a `<jms:property>` tag. When you want to hard code an actual property value into the WSDL, you use a `<jms:propertyValue>` tag. The `<jms:propertyValue>` tag contains a specification of a literal value and its associated XML schema type.

You can specify `<jms:property>` and `<jms:propertyValue>` extensions within the `<wsdl:input>` tag in the binding operation, and also within the `<jms:address>` tag. For the`<wsdl:output>` tag in the binding operation, you can only specify the `<jms:property>` extension. Property values that are set in the `<jms:property>` tag take precedence over values set in the `<<jms:propertyValue>` tag, and property values that are set in the binding operation (in the `<input>` and `<output>` tags) take precedence over values set in the `<jms:address>` tag.

Here is an example of the `<jms:property>` and `<jms:propertyValue>` tags nested within the `<input>` and `<output>` tags:

```
<wsdl:input ... >

    <jms:property name="propertyName" part="partName" />

    <jms:propertyValue name="propertyName"
               type="xsdType" value="actualValue" />

</wsdl:input>

<wsdl:output ... >

    <jms:property name="propertyName" part="partName" />

</wsdl:output>
```

where *propertyName* identifies the JMS property that is associated with the header field, and *partName* identifies the message part that is associated with the property.

The JMS property identified by *propertyName* can be user-defined, or it can be one of the following predefined JMS message header fields:

Value	Java type
JMSMessageId	java.lang.String
JMSTimeStamp	long
JMSCorrelationId	byte [ ] or java.lang.String
JMSReplyTo	javax.jms.Destination
JMSDestination	javax.jms.Destination
JMSDeliveryMode	int
JMSRedelivered	boolean
JMSType	java.lang.String
JMSExpiration	long

See the JMS specification for restrictions that apply when setting JMS header field values. Attempts to set restricted values are ignored.

For application-defined JMS message properties, the Java types used in the native JMS binding implementation (used for calls to the corresponding JMS methods) are derived from the XML schema type in the abstract interface (`<wsdl:part>` tag), and from the type mapping information in the format binding (`<format:typemap>` tag).

Handling transactions

Independent of other JMS properties, the asynchronous processing of request-response operations has implications for callers running in a transaction scope. The send request part and the receive response part are separated into two transactions, because the send needs to be committed in order for the request message to become visible. Implementations that process WSDL for asynchronous request-response operations (such as WSIF) must therefore take the following additional actions:

- They must ensure that the send request transaction returns a correlation ID to the user, and provides a **callback** with which users can pass in the response message to process the receive response transaction.
- They might implement their own response message "listener" in order to recognize the arrival of response messages, and to manage the correlation to the request message.

The JMS text message contains a **java.lang.String**. In this example, the WSDL message contains only one part that represents the whole message body:

```
<!-- Example 1: JMS Text Message -->

<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest"> ... </wsdl:message>
   <wsdl:message name="JmsOperationResponse"> ... </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <jms:binding type="TextMessage" />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:String" formatType="String" />
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:input parts="requestMessageBody" />
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:output parts="responseMessageBody" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

As an extension to the previous JMS message example, the following example WSDL describes a request-response operation in which specific JMS property values of the request and response message are set for the request message and retrieved from the response message.

The JMS properties in the request message are set according to the values in the input message. Likewise, selected JMS properties of the response message are copied to the corresponding values of the output message. The direction of the mapping is determined by the appearance of the `<jms:property>` tag in the input or output section, respectively.

```
<!-- Example 2: JMS Message with JMS Properties -->

<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest">
      <wsdl:part name="myInt" type="xsd:int"/>
      ...
   </wsdl:message>

   <wsdl:message name="JmsOperationResponse">
      <wsdl:part name="myString" type="xsd:String"/>
      ...
   </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <!-- the JMS message type may be any of the above -->
      <jms:binding type="..." />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:int" formatType="int" />
         ...
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:property message="tns:JmsOperationRequest" parts="myInt" />
            <jms:propertyValue name="myLiteralString"
                          type="xsd:string" value="Hello World" />
            ...
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:property message="tns:JmsOperationResponse" parts="myString" />
            ...
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
    </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

The JMS providers - Configuring the client and server:

This topic assumes that you installed a Java Messaging Service (JMS) provider when you installed WebSphere Application Server (either the JMS provider that is embedded in WebSphere Application Server, or another provider such as WebSphere MQ). If not, install one now as described in Installing and configuring a JMS provider.

To enable a service to be invoked through JMS by a Web Services Invocation Framework (WSIF) client application, complete the following steps:

1. Use the administrative console to create and configure a queue connection factory and a queue destination as described in Configuring JMS provider resources.
2. Use the administrative console to add the new queue destination to the list of JMS Server destination names for your application server as described in Managing WebSphere internal JMS servers. Ensure that the Initial State is started.
3. Put the JNDI names of the queue destination and queue connection factory, as well as your JNDI configuration, in the WSDL file.

You should also understand the specific ways in which WSIF interacts with JMS:

- Only input JMS properties are supported.
- WSIF needs two queues when invoking an operation: one for the request message and one for the reply. The replyTo queue is by default a temporary queue, which WSIF creates on behalf of the application. You can specify a permanent queue by setting the JMSReplyTo property to the JNDI name of a queue.
- WSIF uses the default values for properties set by the JMS implementation. However in MQSeries and in some other JMS implementations, messages are persistent by default, and the default temporary queue is of type temporary dynamic and cannot have persistent messages written to it. Therefore your JMS listener can fail to write a persistent response message to the temporary replyTo queue.

  Note: If you are using MQSeries, you need to create a temporary model queue that is of type permanent dynamic, then pass this model as the tempmodel of your queue connection factory. This will ensure that persistent messages are written to a temporary replyTo queue that is of type permanent dynamic.

Using the Java provider

Using the WSIF Java provider, WSIF can invoke Java code.

This means that, in a thin-client environment such as a Java Virtual Machine (JVM) or Tomcat test run-time, you can define shortcuts to local Java programs.

The Web Services Invocation Framework (WSIF) Java provider is not intended for use in a Java 2 platform, Enterprise Edition (J2EE) environment. There is a difference between a client using the WSIF Java provider to invoke a Java component, and implementing a Web service as a Java component on the server side.

The Java binding exploits the format binding for type mapping. Using the format binding, your WSDL can define the mapping between XML schema types and Java types.

The Java provider requires that the targeted Java classes reside in the class path of the client. The Java method is invoked synchronously, in-process, in-thread, with the current thread and Object Request Broker (ORB) contexts.

The Java provider is not transactional.

For examples of the code changes that need to be made in the WSDL file, see The Java provider - Writing the WSDL extension.

The Java provider - Writing the WSDL extension:

The Java provider supports the invocation of a method on a local Java object.

To use the Java provider, you need the following binding specified in the WSDL:

```
<!-- Java binding -->
<binding .... >
    <java:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
    <operation>*
        <java:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            methodType="instance|constructor" />
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
```

In this example:
- A question mark (?) means optional, and an asterisk (*) means 0 or more.
- The name attribute of the <format:typeMap> element is a qualified name of a simple or complex type used by one of the Java operations.
- The formatType attribute of the <format:typeMap> element is the fully qualified class name for the Java class to which the element specified by name maps.
- The methodName attribute of the <java:operation> element is the name of the method on the Java object that is called by the operation.
- The parameterOrder attribute of the <java:operation> element contains a white space-separated list of part names that define the order in which they are passed to the Java object method.
- The methodType attribute of the <java:operation> element must be set to either instance or constructor. The value specifies whether the method that is invoked on the object is an instance method or a constructor for the object.

In the next example, the className attribute of the <java:address> element specifies the fully qualified class name of the object containing the method to invoke:

```
 <service ... >
     <port>*
         <java:address
             className="nmtoken"/>
     </port>
</service>
```

Using the EJB provider

Using the EJB provider, WSIF clients can invoke enterprise beans.

The EJB client JAR file must be available in the client run-time with the current provider. The enterprise bean is invoked using normal EJB invocation methods, using Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP), with the current security and transaction contexts. If the EJB provider is invoked within a transaction, the transaction is passed to the onward service and the standard EJB transaction attribute applies.

If there are multiple implementations of the service, it is up to the service providers to make sure that every implementation offers the same semantics. For example, in the case of transactions, the bean deployer must specify TX_REQUIRES_NEW to force a new transaction.

For examples of the code changes that need to be made in the WSDL file, see The EJB provider - Writing the WSDL.

The EJB provider - Writing the WSDL extension:

The EJB provider supports the invocation of an enterprise bean through Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP).

To use the EJB provider, you need the following binding specified in the WSDL:

```
<!-- EJB binding -->
<binding .... >
    <ejb:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
    <operation>*
        <ejb:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            interface="remote|home" />
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
```

In this example:
- A question mark (?) means optional, and an asterisk (*) means 0 or more.
- The `name` attribute of the `<format:typeMap>` element is a qualified name of a simple or complex type used by one of the EJB operations.
- The `formatType` attribute of the `<format:typeMap>` element is the fully qualified class name for the Java class to which the element specified by `name` maps.
- The `methodName` attribute of the `<ejb:operation>` element is the name of the method on the enterprise bean that is called by the operation.
- The `parameterOrder` attribute of the `<ejb:operation>` element contains a white space-separated list of part names that define the order in which they are passed to the EJB method.
- The `interface` attribute of the `<ejb:operation>` element must be set to either `remote` or `home`. The value specifies the interface of the enterprise bean on which the method named by the `methodName` attribute is accessible.

In the next example:
- The `className` attribute of the `<ejb:address>` element specifies the fully qualified class name of the home interface class of the enterprise bean.
- The `jndiName` attribute of the `<ejb:address>` element specifies the full Java Naming and Directory Interface (JNDI) name that is used to look up the enterprise bean.
- The `initialContextFactory` attribute of the `<ejb:address>` element is optional and specifies the initial context factory class.
- The `jndiProviderURL` attribute of the `<ejb:address>` element is optional and specifies the JNDI provider Web address.

```
 <service ... >
     <port>*
         <ejb:address
             className="nmtoken"
             jndiName="nmtoken"
             initialContextFactory="nmtoken" ?
             jndiProviderURL="nmtoken" ? />
     </port>
</service>
```

Developing a WSIF service

A Web Services Invocation Framework (WSIF) service is a Web service that uses WSIF.

To develop a WSIF service, develop the Web service (or use an existing Web service), then develop the WSIF client based on the WSDL document for that Web service.

There are also two pre-built WSIF Samples available for download from the Samples Central page of the DeveloperWorks WebSphere Web site:
- The Address Book Sample.
- The Stock Quote Sample.

For more information on using the pre-built Samples, see the documentation that is included in the download package.

To develop a WSIF service, complete the following steps:
1. Develop the Web service.

   Use Web services tools to discover, create, and publish the Web service. You can develop Java bean, enterprise bean, and URL Web services. You can use Web service tools to create skeleton Java code and a sample application from a WSDL document. For example, an enterprise bean can be offered as a Web service, using Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) as the access protocol. Or you can use a Java class as a Web service, with native Java invocations as the access protocol.

   You can use the WebSphere Studio Application Developer to create a Web service from a Java application, as described in its StockQuote service tutorial. The Java application that you use in this scenario returns the last trading price from the Internet Web site www.xmltoday.com, given a stock symbol. Using the Web service wizard, you generate a binding WSDL document named `StockQuoteService-binding.wsdl` and a service WSDL document named `StockQuoteService-service.wsdl` from the `StockQuoteService.java` bean. You then deploy the Web service to a Web server, generate a client proxy to the Web service, and generate a sample application that accesses the StockQuoteService through the client proxy. You test the StockQuote Web service, publish it using the IBM UDDI Explorer, and then discover the StockQuote Web service in the IBM UDDI Test Registry.
2. Develop the WSIF client. The information you need to develop a WSIF client is provided in the following topics:
   - Developing the WSIF client - the Address Book Sample gives example code to show how you define a Web service in WSDL.
   - Using the WSIF providers describes the available providers, and gives example code of how their WSDL extensions are coded.
   - WSIF API defines the main interfaces that your client uses to support the invocation of Web services defined in WSDL.

   The Address Book Sample is written for synchronous interaction. If you are using a JMS provider, your WSIF client might need to act asynchronously. WSIF provides two main features that meet this requirement:
   - A **correlation service** that assigns identifiers to messages so that the request can match up with the (eventual) response.
   - A **response handler** that picks up the response from the Web service at a later time.

   For more information, see the WSIF API topic WSIFOperation - Asynchronous interactions reference.

Developing the WSIF client - the Address Book Sample

The code fragments in this topic show you how to use the Web Services Invocation Framework (WSIF) API to invoke the AddressBook Sample Web service dynamically.

This is example code for dynamic invocation of the AddressBook sample Web service using WSIF:

```
try {
    String wsdlLocation="clients/addressbook/AddressBookSample.wsdl";

    // The starting point for any dynamic invocation using wsif is a
    // WSIFServiceFactory. We create ourselves one via the newInstance
```

```
        // method.
        WSIFServiceFactory factory = WSIFServiceFactory.newInstance();

        // Once we have a factory, we can use it to create a WSIFService object
        // corresponding to the AddressBookService service in the wsdl file.
        // Note: since we only have one service defined in the wsdl file, we
        // do not need to use the namespace and name of the service and can pass
        // null instead. This also applies to the port type, although values have
        // been used below for illustrative purposes.
        WSIFService service = factory.getService(
            wsdlLocation,    // location of the wsdl file
            null,            // service namespace
            null,            // service name
           "http://www.ibm.com/namespace/wsif/samples/ab", // port type namespace
            "AddressBookPT" // port type name
        );

        // The AddressBook.wsdl file contains the definitions for two complexType
        // elements within the schema element. We will now map these complexTypes
        // to Java classes. These mappings are used by the Apache SOAP provider
        service.mapType(
            new javax.xml.namespace.QName(
                "http://www.ibm.com/namespace/wsif/samples/ab/types",
                "address"),
            Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"));
        service.mapType(
            new javax.xml.namespace.QName(
                "http://www.ibm.com/namespace/wsif/samples/ab/types",
                "phone"),
            Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFPhone"));
            // We now have a WSIFService object. The next step is to create a WSIFPort
            // object for the port we wish to use. The getPort(String portName) method
            // allows us to generate a WSIFPort from the port name.
        WSIFPort port = null;

        if (portName != null) {
            port = service.getPort(portName);
        }
        if (port == null) {
            // If no port name was specified, attempt to create a WSIFPort from
            // the available ports for the port type specified on the service
            port = getPortFromAvailablePortNames(service);
        }

        // Once we have a WSIFPort, we can create an operation. We are going to execute
        // the addEntry operation and therefore we attempt to create a WSIFOperation
        // corresponding to it. The addEntry operation is overloaded in the wsdl ie.
  // there are two versions of it, each taking different parameters (parts).
  // This overloading requires that we specify the input and output message
  // names for the operation in the createOperation method so that the correct
  // operation can be resolved.
        // Since the addEntry operation has no output message, we use null for its name.
        WSIFOperation operation =
            port.createOperation("addEntry", "AddEntryWholeNameRequest", null);

        // Create messages to use in the execution of the operation. This should
        // be done by invoking the createXXXXXMessage methods on the WSIFOperation.
        WSIFMessage inputMessage = operation.createInputMessage();
        WSIFMessage outputMessage = operation.createOutputMessage();
        WSIFMessage faultMessage = operation.createFaultMessage();

        // Create a name and address to add to the addressbook
        String nameToAdd="Chris P. Bacon";
        WSIFAddress addressToAdd =
            new WSIFAddress (1,
                "The Waterfront",
                "Some City",
                "NY",
                47907,
                new WSIFPhone (765, "494", "4900"));

        // Add the name and address to the input message
        inputMessage.setObjectPart("name", nameToAdd);
        inputMessage.setObjectPart("address", addressToAdd);

        // Execute the operation, obtaining a flag to indicate its success
        boolean operationSucceeded =
            operation.executeRequestResponseOperation(
                inputMessage,
                outputMessage,
                faultMessage);

        if (operationSucceeded) {
            System.out.println("Successfully added name and address to addressbook\n");
        } else {
            System.out.println("Failed to add name and address to addressbook");
        }

        // Start from fresh
        operation = null;
        inputMessage = null;
        outputMessage = null;
        faultMessage = null;

        // This time we will lookup an address from the addressbook.
        // The getAddressFromName operation is not overloaded in the
        // wsdl and therefore we can simply specify the operation name
        // without any input or output message names.
        operation = port.createOperation("getAddressFromName");

        // Create the messages
        inputMessage = operation.createInputMessage();
        outputMessage = operation.createOutputMessage();
        faultMessage = operation.createFaultMessage();

        // Set the name to find in the addressbook
        String nameToLookup="Chris P. Bacon";
        inputMessage.setObjectPart("name", nameToLookup);

        // Execute the operation
        operationSucceeded =
            operation.executeRequestResponseOperation(
                inputMessage,
                outputMessage,
                faultMessage);

        if (operationSucceeded) {
            System.out.println("Successful lookup of name '"+nameToLookup+"' in addressbook");

            // We can obtain the address that was found by querying the output message
            WSIFAddress addressFound = (WSIFAddress) outputMessage.getObjectPart("address");
            System.out.println("The address found was:");
            System.out.println(addressFound);
        } else {
            System.out.println("Failed to lookup name in addressbook");
        }

    } catch (Exception e) {
        System.out.println("An exception occurred when running the sample:");
        e.printStackTrace();
    }
}
```

The preceding code refers to the following Sample method:

```
    WSIFPort getPortFromAvailablePortNames(WSIFService service)
            throws WSIFException {
        String portChosen = null;

        // Obtain a list of the available port names for the service
        Iterator it = service.getAvailablePortNames();
        {
            System.out.println("Available ports for the service are: ");
            while (it.hasNext()) {
                String nextPort = (String) it.next();
                if (portChosen == null)
                    portChosen = nextPort;
                System.out.println(" - " + nextPort);
            }
        }
        if (portChosen == null) {
            throw new WSIFException("No ports found for the service!");
        }
        System.out.println("Using port " + portChosen + "\n");

        // An alternative way of specifying the port to use on the service
        // is to use the setPreferredPort method. Once a preferred port has
        // been set on the service, a WSIFPort can be obtained via getPort
        // (no arguments). If a preferred port has not been set and more than
        // one port is available for the port type specified in the WSIFService,
        // an exception is thrown.
        service.setPreferredPort(portChosen);
        WSIFPort port = service.getPort();
        return port;
    }
```

The Web service uses the following classes:

WSIFAddress:

```
public class WSIFAddress implements Serializable {

    //instance variables
    private int streetNum;
    private java.lang.String streetName;
    private java.lang.String city;
    private java.lang.String state;
    private int zip;
    private WSIFPhone phoneNumber;

    //constructors
    public WSIFAddress () { }

    public WSIFAddress (int streetNum,
                        java.lang.String streetName,
                        java.lang.String city,
                        java.lang.String state,
                        int zip,
                        WSIFPhone phoneNumber) {
          this.streetNum = streetNum;
          this.streetName = streetName;
          this.city = city;
          this.state = state;
          this.zip = zip;
          this.phoneNumber = phoneNumber;
    }

    public int getStreetNum() {
          return streetNum;
    }

    public  void setStreetNum(int streetNum) {
          this.streetNum = streetNum;
    }

    public java.lang.String getStreetName() {
          return streetName;
    }

    public  void setStreetName(java.lang.String streetName) {
          this.streetName = streetName;
    }

    public java.lang.String getCity() {
          return city;
    }

    public  void setCity(java.lang.String city) {
          this.city = city;
    }

    public java.lang.String getState() {
          return state;
    }

    public  void setState(java.lang.String state) {
          this.state = state;
    }

    public int getZip() {
          return zip;
    }

    public  void setZip(int zip) {
          this.zip = zip;
    }

    public WSIFPhone getPhoneNumber() {
          return phoneNumber;
    }

    public  void setPhoneNumber(WSIFPhone phoneNumber) {
          this.phoneNumber = phoneNumber;
    }
}
```

WSIFPhone:

```
public class WSIFPhone implements Serializable {

      //instance variables
      private int areaCode;
      private java.lang.String exchange;
      private java.lang.String number;

      //constructors
      public WSIFPhone () { }

      public WSIFPhone (int areaCode,
                        java.lang.String exchange,
                        java.lang.String number) {
            this.areaCode = areaCode;
            this.exchange = exchange;
            this.number = number;
      }

      public int getAreaCode() {
            return areaCode;
      }
```

```
    public  void setAreaCode(int areaCode) {
          this.areaCode = areaCode;
}

    public java.lang.String getExchange() {
          return exchange;
    }

    public  void setExchange(java.lang.String exchange) {
          this.exchange = exchange;
    }

    public java.lang.String getNumber() {
          return number;
    }

    public  void setNumber(java.lang.String number) {
          this.number = number;
    }
}
```

WSIFAddressBook:

```
public class WSIFAddressBook {
    private Hashtable name2AddressTable = new Hashtable();

    public WSIFAddressBook() {
    }

    public void addEntry(String name, WSIFAddress address)
    {
        name2AddressTable.put(name, address);
    }

    public void addEntry(String firstName, String lastName, WSIFAddress address)
    {
        name2AddressTable.put(firstName+" "+lastName, address);
    }

    public WSIFAddress getAddressFromName(String name)
        throws IllegalArgumentException
    {

        if (name == null)
        {
            throw new IllegalArgumentException("The name argument must not be " +
                                               "null.");
        }
        return (WSIFAddress)name2AddressTable.get(name);
    }

}
```

The following code is the corresponding WSDL file for the Web service:

```
<?xml version="1.0" ?>

<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:typens="http://www.ibm.com/namespace/wsif/samples/ab/types"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
             xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
             xmlns:ejb="http://schemas.xmlsoap.org/wsdl/ejb/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
<types>
  <xsd:schema
     targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab/types"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:complexType name="phone">
      <xsd:element name="areaCode" type="xsd:int"/>
      <xsd:element name="exchange" type="xsd:string"/>
      <xsd:element name="number" type="xsd:string"/>
    </xsd:complexType>

    <xsd:complexType name="address">
      <xsd:element name="streetNum" type="xsd:int"/>
      <xsd:element name="streetName" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:int"/>
      <xsd:element name="phoneNumber" type="typens:phone"/>
    </xsd:complexType>

  </xsd:schema>
</types>

<message name="AddEntryWholeNameRequestMessage">
  <part name="name" type="xsd:string"/>
  <part name="address" type="typens:address"/>
</message>

<message name="AddEntryFirstAndLastNamesRequestMessage">
  <part name="firstName" type="xsd:string"/>
  <part name="lastName" type="xsd:string"/>
  <part name="address" type="typens:address"/>
</message>

<message name="GetAddressFromNameRequestMessage">
  <part name="name" type="xsd:string"/>
</message>

<message name="GetAddressFromNameResponseMessage">
  <part name="address" type="typens:address"/>
</message>

<portType name="AddressBookPT">
  <operation name="addEntry">
    <input name="AddEntryWholeNameRequest"
     message="tns:AddEntryWholeNameRequestMessage"/>
  </operation>
  <operation name="addEntry">
    <input name="AddEntryFirstAndLastNamesRequest"
     message="tns:AddEntryFirstAndLastNamesRequestMessage"/>
  </operation>
  <operation name="getAddressFromName">
    <input name="GetAddressFromNameRequest"
     message="tns:GetAddressFromNameRequestMessage"/>
    <output name="GetAddressFromNameResponse"
      message="tns:GetAddressFromNameResponseMessage"/>
  </operation>
</portType>

<binding name="SOAPHttpBinding" type="tns:AddressBookPT">
  <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryWholeNameRequest">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryFirstAndLastNamesRequest">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="getAddressFromName">
    <soap:operation soapAction=""/>
    <input name="GetAddressFromNameRequest">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output name="GetAddressFromNameResponse">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>

<binding name="JavaBinding" type="tns:AddressBookPT">
  <java:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
           formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="name address"
       methodType="instance"/>
    <input name="AddEntryWholeNameRequest"/>
  </operation>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="firstName lastName address"
       methodType="instance"/>
    <input name="AddEntryFirstAndLastNamesRequest"/>
  </operation>
  <operation name="getAddressFromName">
    <java:operation
       methodName="getAddressFromName"
       parameterOrder="name"
       methodType="instance"
       returnPart="address"/>
    <input name="GetAddressFromNameRequest"/>
    <output name="GetAddressFromNameResponse"/>
  </operation>
</binding>

<binding name="EJBBinding" type="tns:AddressBookPT">
  <ejb:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
           formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>
  <operation name="addEntry">
    <ejb:operation
```

```
         methodName="addEntry"
         parameterOrder="name address"
         interface="remote"/>
      <input name="AddEntryWholeNameRequest"/>
    </operation>
    <operation name="addEntry">
      <ejb:operation
         methodName="addEntry"
         parameterOrder="firstName lastName address"
         interface="remote"/>
      <input name="AddEntryFirstAndLastNamesRequest"/>
    </operation>
    <operation name="getAddressFromName">
      <ejb:operation
         methodName="getAddressFromName"
         parameterOrder="name"
         interface="remote"
         returnPart="address"/>
      <input name="GetAddressFromNameRequest"/>
      <output name="GetAddressFromNameResponse"/>
    </operation>
  </binding>
  <service name="AddressBookService">
    <port name="SOAPPort" binding="tns:SOAPHttpBinding">
      <soap:address
       location="http://localhost/wsif/samples/addressbook/soap/servlet/rpcrouter"/>
    </port>
    <port name="JavaPort" binding="tns:JavaBinding">
      <java:address className="services.addressbook.WSIFAddressBook"/>
    </port>
    <port name="EJBPort" binding="tns:EJBBinding">
      <ejb:address className="services.addressbook.ejb.AddressBookHome"
        jndiName="ejb/samples/wsif/AddressBook"
  classLoader="services.addressbook.ejb.AddressBook.ClassLoader"/>
    </port>
  </service>

</definitions>
```

Using complex types

WSIF supports user-defined complex types through the mapping of complex types to Java classes.

You specify this mapping manually or automatically as described in the following sections:
- Manual mapping of complex types.
- Automatic mapping of complex types.

Any calls to the WSIFService mapType and mapPackage methods used for manual mapping override any equivalent mapping information that is produced automatically. This override helps to maintain backwards compatibility, and also accommodates less standard mappings.

Manual mapping of complex types

The method to use when you create these mappings manually depends on the provider that is used. For the Java and EJB providers, the mappings are specified in the WSDL file in the binding element. The following example provides the syntax for specifying the mapping:

```
<binding .... >
    <ejb:binding|java:binding/>
      <format:typeMapping style="Java" encoding="Java"/>?
          <format:typeMap name="qname" formatType="nmtoken"/>*
      </format:typeMapping>
  ...
</binding>
```

In this example:

- A question mark (“?”) means “optional” and an asterisk (“*”) means “0 or more”.
- The format:typeMap **name** attribute is a qualified name of a complex type or simple type used by one of the operations.
- The format:typeMap **formatType** attribute is the fully qualified class name for the Java class to which the element specified by **name** maps.

If you use the Apache SOAP provider then you specify the mapping of a complex type to a Java class in the client code through two methods on the org.apache.wsif.WSIFService interface:

```
public void mapType(QName elementType, Class javaType)
```

and

```
public void mapPackage(String namespaceURI, String packageName)
```

Use the **mapType** method to specify a mapping between an XML schema element and a Java class. The method takes a QName representing the complex type or simple type, and the corresponding Java class to which it maps.

Use the **mapPackage** method to specify a more general mapping between a namespace and a Java package. Any custom, complex or simple type whose namespace matches that of the mapping is mapped to a Java class in the corresponding package. The name of the actual class is derived from the name of the complex type using standard XML to Java naming conventions.

Automatic mapping of complex types

For complex types defined in the WSDL, where a generated bean is used to represent this type in Java, the Web Services Invocation Framework (WSIF) programming model requires that a call is made to the WSIFService.mapType() method. This call tells WSIF the package and class name of the bean representing the XML schema type that is identified with a QName. To make things easier, the WSIFService.mapPackage() method provides a mechanism to specify a wildcard version of this, where any class within a specified package is mapped to the namespace of a QName. This is a mechanism for manually mapping an XML schema type to a Java class and back again (one mapping entry provides a bidirectional mapping).

There are many ways to convert a QName representing an XML schema type name to a Java package name and class. To enable automatic type mapping, set the WSIF_FEATURE_AUTO_MAP_TYPES feature on the WSIFServiceFactory instance:

```
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
factory.setFeature(WSIFConstants.WSIF_FEATURE_AUTO_MAP_TYPES, new Boolean(true));
```

WSIF maps types by converting the URI part of the XML schema type <tt>QName</tt> to a package name, and converting the local part to a class name. WSIF does this mapping using the WSIFUtils methods <tt>getPackageNameFromNamespaceURI</tt> and <tt>getJavaClassNameFromXMLName</tt>.

Using the Java Naming and Directory Interface (JNDI)

This example task shows you how to use WSIF to bind a reference to a Web service, then look up the reference using JNDI.

You access a Web service through information provided in the WSDL document for the service. If you do not know where to find the WSDL document for the service, but you know that it has been registered in a UDDI registry, then you look it up in the registry. Java programs access Java objects and resources in a similar manner, but using a JNDI interface.

The following example shows how, using the Web Services Invocation Framework (WSIF), you can bind a reference to a Web service then look up the reference using JNDI.

Specifying the argument values for the Web service

The Web service is represented in WSIF by an instance of the org.apache.wsif.naming.WSIFServiceRef class. This simple Referenceable object has the following constructor:

```
public WSIFServiceRef(
        String WSDL,
        String sNS,
        String sName,
        String ptNS,
        String ptName)
{
    [...]
}
```

In this example
- *WSDL* is the location of the WSDL file containing the definition of the service.
- *sNS* is the full namespace for the service definition (you can specify `null` if only one service is defined in the WSDL file).
- *sName* is the local name for the service definition (you can specify `null` if only one service is defined in the WSDL file).
- *ptNS* is the full namespace for the port type within the service that you want to use (you can specify `null` if only one port type is available for the service).
- *ptName* is the local name for the port type (you can specify `null` if only one port type is available for the service).

For example, if the WSDL file for the Web service is available from the Web address `http://localhost/WSDL/Example.WSDL` and contains the following service and port type definitions:

```
  <definitions targetNamespace="http://hostname/namespace/example"
               xmlns:abc="http://hostname/namespace/abc"
[...]
    <portType name="ExamplePT">
      <operation name="exampleOp">
        <input name="exampleInput" message="tns:ExampleInputMsg"/>
      </operation>
    </portType>
[...]
    <service name="abc:ExampleService">
[...]
    </service>
[...]
  </definitions>
```

You can specify the following argument values for the WSIFServiceRef class:
- *WSDL* is `http://localhost/WSDL/Example.WSDL`
- *sNS* is `http://hostname/namespace/abc`
- *sName* is `ExampleService`
- *ptNS* is `http://hostname/namespace/example`
- *ptName* is `ExamplePT`

Binding the service using JNDI

To bind the service reference in the naming directory using JNDI, you can use the com.ibm.websphere.naming.JndiHelper class in WebSphere Application Server:

```
[...]
    import com.ibm.websphere.naming.JndiHelper;
    import org.apache.wsif.naming.*;
```

```
[...]
    try {
       Context startingContext = new InitialContext();
       WSIFServiceRef ref = new WSIFServiceRef("http://localhost/WSDL/Example.WSDL",
                                               "http://hostname/namespace/abc"
                                               "ExampleService",
                                               "http://hostname/namespace/example",
                                               "ExamplePT");
       JndiHelper.recursiveRebind(startingContext,
            "myContext/mySubContext/myServiceRef", ref);

    }
    catch (NamingException e) {
       // Handle  error.
    }
[...]
```

Looking up the service using JNDI

The following code fragment shows the lookup of a service using JNDI:

```
[...]
    try {
[...]
       InitialContext ic = new InitialContext();
       WSIFService myService =
        (WSIFService) ic.lookup("myContext/mySubContext/myServiceRef");
[...]
    }
    catch (NamingException e) {
       // Handle error.
    }
[...]
```

Passing SOAP messages with attachments using WSIF

The W3C SOAP Messages with Attachments document describes a standard way to associate a SOAP message with one or more attachments in their native format (for example GIF or JPEG) by using a multipart MIME structure for transport. It defines specific use of the “Multipart/Related” MIME media type, and rules for the use of URI references to entities bundled within the MIME package. It thereby outlines a technique for carrying a SOAP 1.1 message within a MIME multipart/related message in such a way that the SOAP processing rules for a standard SOAP message are not changed.

The Web Services Invocation Framework (WSIF) supports passing attachments in a MIME message using the SOAP provider. The attachment is a javax.activation.DataHandler object. The mime:multipartRelated, mime:part and mime:content tags are used to describe the attachment in the WSDL.

For more information, see the following topics:
- SOAP messages with attachments - Writing the WSDL extensions.
- SOAP messages with attachments - Passing attachments to WSI.
- SOAP messages with attachments - Working with types and type mappings.

The following scenarios are not supported:
- Using DIME.
- Passing in `javax.xml.transform.Source` and `javax.mail.internet.MimeMultipart`.
- Using the mime:mimeXml WSDL tag.
- Nesting a mime:multipartRelated tag inside a mime:part tag.
- Using types that extend `DataHandler`, `Image`, and so on.
- Using types that contain `DataHandler`, `Image`, and soon.
- Using Arrays or Vectors of `DataHandlers`, `Images`, and so on.
- Using multiple in/out or output attachments.

The MIME headers from the incoming message are not preserved for referenced attachments. The outgoing message contains new MIME headers for Content-Type, Content-Id and Content-Transfer-Encoding that are created by WSIF.

SOAP messages with attachments - Writing the WSDL extensions

The following example WSDL illustrates a simple operation that has one attachment called `attch`:

```
<binding name="MyBinding" type="tns:abc" >
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="MyOperation">
    <soap:operation soapAction=""/>
    <input>
      <mime:multipartRelated>
        <mime:part>
          <soap:body use="encoded" namespace="http://mynamespace"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </mime:part>
        <mime:part>
          <mime:content part="attch" type="text/html"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
  </operation>
</binding>
```

In this type of WSDL extension:
- There must be a `part` attribute (in this example `attch`) on the input message for the operation (in this example `MyOperation`). There can be other input parts to `MyOperation` that are not attachments.
- In the binding input there must either be a `<soap:body>` tag or a `<mime:multipartRelated>` tag, but not both.
- For MIME messages, the `<soap:body>` tag is inside a `<mime:part>` tag. There must only be one `<mime:part>` tag that contains a `<soap:body>` tag in the binding input and that must not contain a `<mime:content>` tag as well, because a content type of `text/xml` is assumed for the `<soap:body>` tag.
- There can be multiple attachments in a MIME message, each described by a `<mime:part>` tag.
- Each `<mime:part>` tag that does not contain a `<soap:body>` tag contains a `<mime:content>` tag that describes the attachment itself. The `type` attribute inside the `<mime:content>` tag is not checked or used by the Web Services Invocation Framework (WSIF). It is there to suggest to the application using WSIF what the attachment contains. Multiple `<mime:content>` tags inside a single `<mime:part>` tag means that the backend service expects a single attachment with a type specified by one of the `<mime:content>` tags inside that `<mime:part>` tag.
- The `parts="..."` attribute (optional) inside the `<soap:body>` tag is assumed to contain the names of all the MIME parts as well as the names of all the SOAP parts in the message.

SOAP messages with attachments - Passing attachments to WSIF

The following code fragment can invoke the service described by the example WSDL in the topic writing the WSDL extensions:

```
import javax.activation.DataHandler;
. . .
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
WSIFOperation op = service.getPort().createOperation("MyOperation");
WSIFMessage in = op.createInputMessage();
in.setObjectPart("attch",dh);
op.executeInputOnlyOperation(in);
```

The associated type mapping in the `DeploymentDescriptor.xml` file depends upon your SOAP server. For example if you use Tomcat with SOAP 2.3, then the `DeploymentDescriptor.xml` file contains the following type mapping:

```
<isd:mappings>
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:x="http://mynamespace"
 qname="x:datahandler"
 javaType="javax.activation.DataHandler"
 java2XMLClassName="org.apache.soap.encoding.soapenc.MimePartSerializer"
 xml2JavaClassName="org.apache.soap.encoding.soapenc.MimePartSerializer" />
</isd:mappings>
```

In this case, the backend service is invoked with the following signature:

```
public void MyOperation(DataHandler dh);
```

You can also use stubs to pass attachments into the Web Services Invocation Framework (WSIF):

```
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
MyInterface stub = (MyInterface)service.getStub(MyInterface.class);
stub.MyOperation(dh);
```

Attachments can also be returned from an operation, but at present only one attachment can be returned as the return parameter.

SOAP messages with attachments - Working with types and type mappings

By default, attachments are passed into the Web Services Invocation Framework (WSIF) as DataHandler objects. If the part on the message that is the DataHandler object maps to a `<mime:part>` tag in the WSDL, then WSIF automatically maps the fully qualified name of the WSDL type to the DataHandler class and sets up that type mapping with the SOAP provider.

In your WSDL, you might have defined a schema for the attachment (for instance as a `binary[]` type). WSIF silently ignores this mapping and treats the attachment as a DataHandler object, unless you explicitly issue a `mapType()` method. WSIF lets the SOAP provider set the MIME content type based on the type of the DataHandler object, instead of the `type` attribute specified for the `<mime:content>` tag in the WSDL.

Interacting with the J2EE container in WebSphere Application Server

Interaction with a container is limited to the following aspects:

- Using the application server administrative console to define Web services to WebSphere Application Server. This task is described in Using the Java Naming and Directory Interface (JNDI) and WSIF system management and administration. As part of the definition of a service, the administrator might define a “preferred port”.
- Using the Web Services Invocation Framework (WSIF) to make log and trace calls to the JRAS services in WebSphere Application Server, as described in Trace and logging for WSIF.
- Using WSIF providers to access Java 2 platform, Enterprise Edition (J2EE) services. For example using the EJB provider to access the Java Naming and Directory Interface (JNDI) and make calls to remote enterprise beans.
- Using WSIF to wrap the use of container services so that, when WSIF is run in an unmanaged (thin) environment, the operation can succeed.

Running WSIF as a client

The Web Services Invocation Framework (WSIF) runs in the WebSphere Application Server application client container, and in similar clients from other suppliers.

To simplify the process of launching client applications in the WebSphere Application Server application client, use the `launchClient` tool as described in Running application clients.

WSIF system management and administration

The Web Services Invocation Framework (WSIF) is provided as a stand-alone JAR file named `wsif.jar`. The JAR file contains the core WSIF classes, and the Java, EJB, SOAP over HTTP and SOAP over JMS providers. Additional providers are packaged as separate JAR files.

When you install WebSphere Application Server, the `wsif.jar` file is put on the WebSphere or Java Virtual Machine (JVM) class path.

WSIF requires no further configuration. WSIF is a thin abstraction layer between application code and the relevant invocation infrastructure.

For specific information on other aspects of managing your WSIF system, see the following topics:
- Maintaining the WSIF properties file
- Enabling security for WSIF
- Trace and logging for WSIF
- Troubleshooting the Web Services Invocation Framework
- WSIF (Web Services Invocation Framework) messages

Maintaining the WSIF properties file

The Web Services Invocation Framework (WSIF) properties are stored in the `wsif.jar` file, in a properties file named `wsif.properties`. This properties file is kept on the class path, so that WSIF can find it and the client administrator can use it to configure WSIF.

Here is a copy of the initial contents of the `wsif.properties` file. All the possible properties are listed and described.

```
# Two properties are used to override which WSIFProvider is selected when there
# exists multiple providers supporting the same namespace URI. These properties are:
#
#    wsif.provider.default.CLASSNAME=N
#    wsif.provider.uri.M.CLASSNAME=URI
#
# CLASSNAME is the WSIFProvider class name
# N is the number of following default wsif.provider.uri.M.CLASSNAME properties
# M is a number from 1 to N to uniquely identify each wsif.provider.uri.M.CLASSNAME
#   property key.
# For example the following two properties would override the default SOAP provider
# to be the Apache SOAP provider:
#
# wsif.provider.default.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=1
# wsif.provider.uri.1.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=\
# http://schemas.xmlsoap.org/wsdl/soap/
#

# maximum number of milliseconds to wait for a response to a synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined on invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

Enabling security for WSIF

The Web Services Invocation Framework (WSIF) interacts with a security manager in the following ways:
- WSIF runs in the Java 2 platform, Enterprise Edition (J2EE) security context without modification.
- When WSIF is run under a J2EE container, port implementations can use the security context to pass on security tokens or credentials as necessary.

- WSIF implementations can automatically convert J2EE security context into appropriate context for onward services.

For WSIF to interact effectively with the WebSphere Application Server security manager, you must set the following permissions in the `server.policy` file:

Permission	Required by SOAP and Java portion?	Required by EJB portion?	Additional notes
FilePermission to load the WSDL	-	-	This permission is only required when a WSDL file is referred to using the file:/// protocol
RuntimePermission "getClassLoader" for the context class loader for the current thread	Yes	No	
RuntimePermission "accessDeclaredMembers"	Yes	Yes	This permission is required by both portions for handling enterprise beans
PropertyPermission for system properties	Yes (read and write access)	Yes (write access only)	This permission is required by SOAP and many others
NetPermission "specifyStreamHandler"	Yes	Yes	This permission must be in either the SOAP and Java portion, or the EJB portion, but it need not be in both.
SocketPermission "*host_name*", "resolve"	No	Yes	Where *host_name* is your host name (for example `localhost`)
SocketPermission "*host_name*:*port_no*", "connect"	Yes	Yes	Where *host_name* is your host name (for example `localhost`) and *port_no* is your port number (for example `9080`).

Troubleshooting the Web Services Invocation Framework

For information on resolving WebSphere-level problems, see Diagnosing and fixing problems.

To identify and resolve Web Services Invocation Framework (WSIF)-related problems, you can use the standard WebSphere Application Server trace and logging facilities. If you encounter a problem that you think might be related to WSIF, you can check for error messages in the WebSphere Application Server administrative console, and in the application server `stdout.log` file. You can also enable the application server debug trace to provide a detailed exception dump.

A list of the WSIF run-time system messages, with details of what each message means, is provided in Message reference for WSIF.

Here is a checklist of major WSIF activities, with advice on common problems associated with each activity:

Create service
Handcrafted WSDL can cause numerous problems. To help ensure that your WSDL is valid, use a tool such as WebSphere Studio to create your service.

Define transport mechanism
: For the Java Messaging Service (JMS), check that you have set up the Java Naming and Directory Interface (JNDI) correctly, and created the necessary connection factories and queues.

  For SOAP, make sure that the deployment descriptor file `dds.xml` is correct - preferably by creating it using WebSphere Studio or similar tooling.

Create client - Java code
: Follow the correct format for creating a WSIF service, port, operation and message. For examples of correct code, see the Address Book Sample.

Compile code (client and service)
: Check that the build path against code is correct, and that it contains the correct levels of JAR files.

  Create a valid EAR file for your service in preparation for deployment to a Web server.

Deploy service
: When you install and deploy the service EAR file, check carefully any messages given when the service is deployed.

Server setup and start
: Make sure that the WebSphere Application Server `server.policy` file (in the `/properties` directory) has the correct security settings. For more information, see Enabling security for WSIF.

WSIF setup
: Check that the `wsif.properties` file is correctly set up. For more information, see Maintaining the WSIF properties file.

Run client
: Either check that you have defined the class path correctly to include references to your client classes, WSIF JAR files and any other necessary JAR files, or (preferably) run your client using the WebSphere Application Server launchClient tool.

Here is a list of common errors, and information on their probable causes:

- **“No class definition” errors received when running client code.**

  This problem usually indicates an error in the class path setup. Check that the relevant JAR files are included.
- **“Cannot find WSDL” error.**

  Some likely causes are:
  - The application server is not running.
  - The server location and port number in the WSDL are not correct.
  - The WSDL is badly formed (check the error messages in the application server `stdout.log` file).
  - The application server has not been restarted since the service was installed.

  You might also try the following checks:
  - Can you load the WSDL into your Web browser from the location specified in the error message?
  - Can you load the corresponding WSDL binding files into your Web browser?
- **Your Web service EAR file does not install correctly onto the application server.**

  It is likely that the EAR file is badly formed. Verify the installation by completing the following steps:
  - For an EJB binding, run the WebSphere Application Server tool `\bin\dumpnamespace`. This tool lists the current contents of the JNDI directory.
  - For a SOAP over HTTP binding, open the `http://pathToServer/WebServiceName/admin/list.jsp` page (if you have the SOAP administration pages installed). This page lists all currently installed Web services.
  - For a SOAP over JMS binding, complete the following checks:
    - Check that the queue manager is running.
    - Check that the necessary queues are defined.
    - Check the JNDI setup.
    - Use the “display context” option in the `jmsadmin` tool to list the current JNDI definitions.
    - Check that the Remote Procedure Call (RPC) router is running.

- **There is a permissions problem or security error.**

  Check that the WebSphere Application Server `server.policy` file (in the `/properties` directory) has the correct security settings. For more information, see Enabling security for WSIF.
- **Using WSIF with multiple clients causes a SOAP parsing error.**

  Before you deploy a Web service to WebSphere Application Server, you must decide on the scope of the Web service. The deployment descriptor file `dds.xml` for the Web service includes the following line:

  ```
  <isd:provider type="java" scope="Application" ......
  ```

  You can set the `Scope` attribute to `Application` or `Session`. The default setting is `Application`, and this value is correct if each request to the Web service does not require objects to be maintained for longer than a single instance. If `Scope` is set to `Application` the objects are not available to another request during the execution of the single instance, and they are released on completion. If your Web service needs objects to be maintained for multiple requests, and to be unique within each request, you must set the scope to `Session`. If `Scope` is set to `Session`, the objects are not available to another request during the life of the session, and they are released on completion of the session. If scope is set to `Application` instead of `Session`, you might get the following SOAP error:

```
SOAPException: SOAP-ENV:ClientParsing error, response was:
FWK005 parse may not be called while parsing.;
nested exception is:

[SOAPException: faultCode=SOAP-ENV:Client; msg=Parsing error, response was:

FWK005 parse may not be called while parsing.;
       targetException=org.xml.sax.SAXException:
FWK005 parse may not be called while parsing.]
```

- 5.1 + **Using the same names for JMS messaging queues and queue connection factories that run on application servers on different machines can cause JNDI lookup errors.** You should not use the same names for messaging queues and queue connection factories that run on application servers on different machines, because WSIF always looks first for JMS destinations locally, and only uses the full JNDI reference if it cannot find the destination locally. For example, if you run a Web service on a remote machine, and have an application server running locally that uses the same names for the messaging queues and queue connection factories, then WSIF will find and use the local queues even if the remote JNDI destination is provided in full in the WSDL service definition.

Trace and logging for WSIF

If you want to enable trace for the Web Services Invocation Framework (WSIF) API within WebSphere Application Server, and have trace, stdout and stderr for the application server written to a well-known location, see Enabling trace.

WSIF offers trace points at the opening and closing of ports, the invocation of services, and the responses from services.

To trace the WSIF API, you need to specify the following trace string:

```
wsif=all=enabled
```

WSIF also includes a `SimpleLog` utility through which you can run trace when using WSIF outside of WebSphere Application Server. To enable this utility, complete the following steps:

1. Create a file named `commons-logging.properties` with the following contents:

```
org.apache.commons.logging.LogFactory=org.apache.commons.logging.impl.LogFactoryImpl
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

2. Create a file named `simplelog.properties` with the following contents:

```
org.apache.commons.logging.simplelog.defaultlog=trace
org.apache.commons.logging.simplelog.showShortLogname=true
org.apache.commons.logging.simplelog.showdatetime=true
```

3. Put both these files, and the `commons-logging.jar` file, on the class path.

The `SimpleLog` uitility writes trace to the `System.err` file.

WSIF (Web Services Invocation Framework) messages

This topic contains a list of the WSIF run-time system messages, with details of what each message means.
WebSphere system messages are logged from a variety of sources, including application server components and applications. Messages logged by application server components and associated IBM products start with a unique message identifier that indicates the component or application that issued the message.

For more information about the message identifier format, see the topic Message reference.

WSIF0001E: An extension registry was not found for the element type "{0}"

Explanation: Parameters: {0} element type. No extension registry was found for the element type specified.

User Response: Add the appropriate extension registry to the port factory in your code.

WSIF0002E: A failure occurred in loading WSDL from "{0}"

Explanation: Parameters: {0} location of the WSDL file. The WSDL file could not be found at the location specified or did not parse correctly

User Response: Check that the location of the WSDL file is correct. Check that any network connections required are available. Check that the WSDL file contains valid WSDL.

WSIF0003W: An error occurred finding pluggable providers: {0}

Explanation: Parameters: {0} specific details about the error. There was a problem locating a WSIF pluggable provider using the J2SE 1.3 JAR file extensions to support service providers architecture. The WSIF trace file will contain the full exception details.

User Response: Verify that a META-INF/services/org.apache.wsif.spi.WSIFProvider file exists in a provider jar, that each class referenced in the META-INF file exists in the class path, and that each class implements org.apache.wsif.spi.WSIFProvider. The class in error will be ignored and WSIF will continue locating other pluggable providers.

WSIF0004E: WSDL contains an operation type "{0}" which is not supported for "{1}"

Explanation: Parameters: {0} name of the operation type specified. {1} name of the portType for the operation. An operation type which is not supported has been specified in the WSDL.

User Response: Remove any operations of the unsupported type from the WSDL. If the operation is required then make sure all messages have been correctly specified for the operation.

WSIF0005E: An error occurred when invoking the method "{1}" . ("{0}")

Explanation: Parameters: {0} name of communication type. For example EJB or Apache SOAP. {1} name of the method that failed. An error was encountered when invoking a method on the Web service using the communication shown in brackets.

User Response: Check that the method exists on the Web service and that the correct parts have been added to the operation as described in the WSDL. Network problems might be a cause if the method is remote and so check any required connections.

WSIF0006W: Multiple WSIFProvider found supporting the same namespace URI "{0}" . Found ("{1}")

Explanation: Parameters: {0} the namespace URI. {1} a list of the WSIFProvider found.. There are multiple org.apache.wsif.spi.WSIFProvider classes in the service provider path that support the same namespace URI.

User Response: A following WSIF0007I message will be issued notifying which WSIPFProvider will be used. Which WSIFProvider is chosen is based on settings in the wsif.properties file, or if not defined in the properties, the last WSIFProvider found will be used. See the wsif.properties file for more details on how to define which provider should be used to support a namespace URI.

WSIF0007I: Using WSIFProvider "{0}" for namespaceURI "{1}"

Explanation: Parameters: {0} the classname of the WSIFProvider being used. {1} the namespaceURI the provider will be used to support.. Either a previous WSIF0006W message has been issued or the SetDynamicWSIFProvider method has been used to override the provider used to support a namespaceURI.

User Response: None. See also WSIF0006W.

WSIF0008W: WSIFDefaultCorrelationService removing correlator due to timeout. ID:"{0}"

Explanation: Parameters: {0} the ID of the correlator being removed from the correlation service. A stored correlator is being removed from the correlation service due to its timeout expiring.

User Response: Determine why no response has been received for the asynchronous request within the timeout period. The wsif.asyncrequest.timeout property of the wsif.properties file defines the length of the timeout period.

WSIF0009I: Using correlation service - "{0}"

Explanation: Parameters: {0} the name of the correlation service being used. This identifies the name of the correlation service that will be used to process asynchronous requests.

User Response: None. If a correlation service other than the default WSIF supplied one is required, ensure that it is correctly registered in the JNDI java:comp/wsif/WSIFCorrelationService namespace.

WSIF0010E: Exception thrown while processing asynchronous response - "{0}"

Explanation: Parameters: {0} the error message string of the exception. While processing the response from an executeRequestResponseAsync call an exception was thrown.

User Response: Use the exception error message string to determine the cause of the error. The WSIF trace will have more details on the error including the exception stack trace.

WSIF0011I: Preferred port "{0}" was not available

Explanation: Parameters: {0} the user's preferred port. The preferred port set by the user on org.apache.wsif.WSIFService is not available

User Response: None unless this message appears for long periods of time in which case the user might want to pick a different port as their preferred port.

WSIF API

The Web Services Invocation Framework (WSIF) API supports the invocation of services defined in WSDL. WSIF is intended for use in both WSIF clients and Web service intermediaries.

The WSIF API is driven by the abstract service description in WSDL; it is completely independent of the actual binding used. This independence makes the API more natural to work with because it uses WSDL terms to refer to message parts, operations, and so on.

The WSIF API was designed for the WSDL usage model: Pick a port that supports the port type needed, then invoke the operation by providing the necessary abstract input message consisting of the required parts, without worrying about how the message is mapped to a specific binding protocol.

Other Web service APIs, for example SOAP APIs, are not designed on WSDL, but for a specific binding protocol with its associated syntax; for example, target URIs and encoding styles.

The WSIF API main interfaces are described in the following topics:
- Creating a message for sending to a port (the WSIFMessage interface).
- WSIF API reference: Finding a port factory or service (the WSIFService interface and the WSIFServiceFactory class).
- WSIF API reference: Using ports (the WSIFPort interface and the WSIFOperation interface).

Note: You must ensure that your application uses only one thread to call WSIF.

For additional technical details of the WSIF API, see the WSIF Javadoc.

WSIF API reference: Creating a message for sending to a port

For message management (that is, message construction and parsing) the underlying API is modeled on WSDL semantics. There is a simple and direct mapping from the WSDL model to the Web Services Invocation Framework (WSIF) classes.

In WSDL, a message describes the abstract type of the input or output to an operation. The corresponding WSIF class is WSIFMessage, which represents in memory the actual input or output of an operation. A WSIFMessage class is a container for a set of named parts. The WSIFMessage interface separates the actual representation of the data from the abstract type defined by WSDL. WSDL defines messages as XML schema types. There are two natural ways to represent a WSDL message in a run-time environment:

- The generated Java class, based on a WSDL to Java mapping such as that provided by a Java API for XML-based remote procedure call (JAX-RPC).
- The XML representation of the data, for example using SOAP Encoding.

Each option offers benefits in different scenarios. The Java class is the natural approach when WSIF is used in a standard Java client. However, in other scenarios where WSIF is used in an intermediary, it might be more efficient to keep a WSDL message in the SOAP encoded format.

The style used to define messages must be consistent within the message, so all the parts in one message must be consistent. A string - `getRepresentationStyle()` - always returns `null`. This indicates that parts on this WSIFMessage class are represented as Java objects.

You add parts to a WSIFMessage class with the setObjectPart or set*Type*Part methods. Each part is named. Part names within a message are unique. If you set a part more than once, the last setting is the one that is used.

You retrieve parts by name from a WSIFMessage class with the getObjectPart or get*Type*Part methods. If the named part does not exist, the method returns a WSIFException exception.

You can use Iterators to retrieve parts from the message through the getParts() and getPartNames() methods.

The order in which you set the parts is not important, but the message implementation might be more efficient if the parts are set in the parameter order specified by WSDL.

WSIFMessage classes are cloneable and serializable. If the parts set are not cloneable, the implementation can try to clone them using serialization. If the parts are not serializable either, then a CloneNotSupportedException exception is thrown if cloning is attempted.

WSIFMessage classes can be sent between Java Virtual Machines (JVMs).

In addition to the containing parts, a WSIFMessage class also has a message name. This is required for operation overloading, which is supported by WSDL and WSIF.

Here is the Javadoc for the WSIFMessage interface.

WSIF API reference: Finding a port factory or service

To find a port you use the WSIFService interface, which is a factory for ports.

The port factory models and supports the WSDL approach in which a service is available on one or more ports. The factory hides the implementation of the port from the user. The Web Services Invocation Framework (WSIF) supports dynamic ports that are based on a particular protocol and transport, and configured using the WSDL at run-time. For example, the dynamic SOAP port can invoke any SOAP service based on the WSDL description of that service. Using this service you can hide and modify the set of available ports at run-time.

Here is the WSIFService interface.

To find a service from a WSDL document at a Web address, or from a code-generated code base, you can use the WSIFServiceFactory class.

WSIFService interface

The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation.
The Web Services Invocation Framework (WSIF) service stores a list of providers that can each generate a WSIF operation for a particular WSDL binding. This service looks up providers by the provider type. For example the service knows about one provider that handles SOAP ports and other providers that handle Java ports that you define. In a managed environment, the container can configure the WSIFService interface.

Here is the Javadoc for the WSIFService interface.

A WSIFService implementation can choose a preferred port based on a number of criteria. The WSIFService implementation can set the preferred port, or it can be set by calling the setPreferredPort method.

The getPort method returns an instance of the WSIFPort class that is used to invoke a service on the port. Variants of the getPort method are used to define the characteristics of the port to be created:

- the getPort method with no arguments returns the preferred port.
- the getPort method with a string argument returns the port named by the string containing the WSDL identifier for the selected port.

The return value is `null` if the port name is not valid.

If a port is chosen (either by the WSIFService implementation, or by the setPreferredPort method), then the WSIFService implementation validates that the relevant provider exists and is configured. If the provider fails this validation check, the WSIFService interface chooses any other port for which a provider is defined. For example, if the preferred port is SOAP over JMS but the JMS libraries are not available, then WSIF chooses another port. If no preferred port is set, or the preferred port is not available, the WSIF implementation chooses the first available port listed in the WSDL.

The getAvailablePortNames() method returns, as an iteration of strings, the list of WSDL port names filtered by the set of available providers.

The getDefinition() method returns the WSDL definition for the service. If the WSDL definition is not available, this method returns `null`.

WSIFServiceFactory class

To find a service from a WSDL document at a Web address, or from a code-generated code base, you can use the WSIFServiceFactory class.

Note: When you create a WSIFService interface from a WSIFServiceFactory class, you can specify a ClassLoader object to use in locating the WSDL file. You need to specify this object when the WSDL file is in a JAR file. In such a case, specify the location of the WSDL file relative to the root of the JAR file, using forward slashes (/) with the preceding slash removed.

For example:

```
com/myCompany/wsdl/MyWSDLFile.wsdl
```

rather than

```
/com/myCompany/wsdl/MyWSDLFile.wsdl
```

Here is the Javadoc for the WSIFServiceFactory class.

The WSIFServiceFactory class returns `null` if no service is found with that identifier.

WSIF API reference: Using ports

A WSIFPort interface handles the details of invoking an operation. The port provides access to the actual implementation of the service.

A WSDL document can provide many different WSDL bindings, and these bindings can drive multiple ports. The client can choose a port, the service stub can choose a port, or the Web Services Invocation Framework (WSIF) can choose a default port.

The port offers an interface to retrieve an Operation object. A WSIFOperation interface offers the ability to execute the given operation.

If the port is serialized and deserialized at a later time, then WSIF ensures that the client provides the correct information to the server to identify the instance. If the server instance is no longer available, then it is up to the server to decide whether to throw a fault or provide a new instance. That behavior can depend on the type of service.

For example, for an enterprise bean the WSIFPort interface stores the EJB Home, and uses it to select the bean before each invocation. It is the responsibility of the client to serialize or maintain the port instance if it wants instance support. The client must create a new operation and messages for each invocation.

Here is the WSIFPort interface.

Here is the WSIFOperation interface.

WSIFPort interface

The port implements a factory method for the WSIFOperation interface.

Here is the Javadoc for the WSIFPort interface.

The createOperation(String) method returns a new instance of a WSIFOperation object. If the operationName value is not valid or the operation is overloaded, then the method throws an exception.

The createOperation(String, String, String) method supports overloaded WSDL operations. You can overload based on the input parameters, but not on the output parameters.

It is the duty of the client to call the close method when a port is no longer in use. In many cases, where the transport is sessionless, like HTTP, this has no effect. However, if the port is using a session-based protocol such as MQSeries, Java Messaging Service (JMS), or External Call Interface (ECI), this supports the port in caching an open connection to the server and then closing it as required. Responsibly-written applications will call the close method if appropriate.

WSIFOperation interface

You use the WSIFOperation interface to invoke a service based on a particular binding.

The WSIFOperation interface is the run-time representation of an operation. This interface provides methods to create input, output, and fault messages, and to invoke the operation.

Here is the Javadoc for the WSIFOperation interface.

createInputMessage, createOutputMessage and createFaultMessage

These are factory methods to create the messages required by the invocation methods. All invocation methods require an input message.

executeRequestResponseOperation
This method invokes “In Out” operations.

executeInputOnlyOperation
This method invokes “In only” operations.

executeRequestResponseOperation
If this method is used for invocation, then an output and a fault message are instantiated and passed on the call to the method. If the method returns `true`, then the output message contains the response message. If the message returns `false`, then a fault occurred and is returned in the fault message.

executeRequestResponseAsync
This method allows “In Out” operations to be invoked with the reply handled using an alternate thread. Use of this method is discussed further in WSIFOperation - Asynchronous interactions.

setContext and getContext
Use of these methods is discussed in WSIFOperation - Context.

All of the **execute*Nnnn*** methods fail with an exception if there is an error in processing the request in the WSIF provider.

Setting the timeouts for synchronous and asynchronous operations is discussed in WSIFOperation - Synchronous and asynchronous timeouts.

WSIFOperation - Context: Although WSDL does not define context, a number of uses of the Web Services Invocation Framework (WSIF) require the ability to pass context to the port that is invoking the service. For example, a SOAP over HTTP port might require an HTTP user name and password. This information is specific to the invocation, but is not a parameter of the service. In general, context is defined as a set of name-value pairs. However, because Web services tend to define the types of data using XML schema types, WSIF represents the name-value pairs of the context using the same representation that WSIFMessage classes use; that is a set of named parts, each of which equates to an instance of an XML schema type.

You use the WSIFOperation interface setContext and getContext methods to pass context information to the binding. The port implementation can use this context, for example to update a SOAP header. There is no definition of how a port can utilize the context.

The parameter of the setContext and getContext methods is a WSIFMessage interface, and this interface has named parts defining the context information. The WSIFConstants class defines constants for the part names that can be set in a context WSIFMessage interface.

The following code fragment shows how to set the user name and password for HTTP basic authentication:

```
// set a basic authentication header
   WSIFMessage headers = new WSIFDefaultMessage();
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_USER, "user name" );
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_PSWD, "password" );
   operation.setContext( headers );
```

The WSIFOperation interface ignores context parts that it does not support. For example, the previous code is ignored by the WSIF Java provider.

The WSIFConstants class includes the following constants that can be used for context part names:
- CONTEXT_HTTP_USER
- CONTEXT_HTTP_PSWD
- CONTEXT_SOAP_HEADERS

The HTTP header values are expected to be of type String, and the SOAP header value is expected to be of type java.util.List, which should contain entries of type org.w3c.dom.Element.

WSIFOperation - Asynchronous interactions reference: The Web Services Invocation Framework (WSIF) supports asynchronous operation. In this mode of operation, the client puts the request message as part of one transaction, and carries on with the thread of execution. The response message is then handled by a different thread, with a separate transaction. The SOAP over JMS and native JMS providers are the only WSIF providers that currently support asynchronous operation.

The WSIFPort class uses the supportsAsync method to test if asynchronous operation is supported.

An asynchronous operation is initiated with the WSIFOperation interface executeRequestResponseAsync method. This method lets a Remote Procedure Call (RPC) method be invoked asynchronously. The method returns before the operation is completed, and the thread of execution continues.

The response to the asynchronous request is processed by the WSIFOperation interface fireAsyncResponse or processAsyncResponse methods.

To initiate the request, there are two forms of the executeRequestResponseAsync method:

```
public WSIFCorrelationId executeRequestResponseAsync
                         (WSIFMessage input, WSIFResponseHandler handler)
```

and

```
public WSIFCorrelationId executeRequestResponseAsync (WSIFMessage input)
```

executeRequestResponseAsync(WSIFMessage input, WSIFResponseHandler handler)
: This method takes an input message and a WSIFResponseHandler handler. The handler is invoked on another thread when the operation completes. When using this method the client listener calls the fireAsyncResponse method, which then calls the WSIFResponseHandler interface executeAsyncResponse method. Here is the Javadoc for the WSIFResponseHandler interface.

executeRequestResponseAsync(WSIFMessage input)
: This method only takes an input message, and does not use a response handler. The client listener processes the response by calling the WSIFOperation interface processAsyncResponse method. This process updates the WSIFMessage output and fault messages with the result of the request.

WSIF supports correlation between the asynchronous request and response. When the request is sent, the WSIFOperation object is serialized into the WSIFCorrelationService object. The executeRequestResponseAsync methods return a WSIFCorrelationId object which identifies the serialized WSIFOperation object. The client listener can use this to match a response to a particular request.

The correlation service is located with the WSIFCorrelationServiceLocator class getCorrelationService() method in the org.apache.wsif.utils package.

In a managed container a default correlation service is defined in the default Java Naming and Directory Interface (JNDI) namespace using the name: java:comp/wsif/WSIFCorrelationService. If this correlation service is not available, then WSIF uses the WSIFDefaultCorrelationService.

Here is the Javadoc for the WSIFCorrelationService interface.

and this is the correlator ID:

```
 public interface WSIFCorrelator extends Serializable {
    public String getCorrelationId();
 }
```

The client must implement its own response message listener or message data base so that it can recognize the arrival of response messages. This client implementation manages the correlation of the response message to the request and call of one of the asynchronous response processing methods. As an example of the requirement for a client listener, the following code fragment shows what can be in the onMessage method of a Java Messaging Service (JMS) listener:

```
public void onMessage(Message msg) {
     WSIFCorrelationService cs = WSIFCorrelationServiceLocator.getCorrelationService();
           WSIFCorrelationId cid = new JmsCorrelationId( msg.getJMSCorrelationID() );
           WSIFOperation op = cs.get( cid );
           op.fireAsyncResponse( msg );
}
```

WSIFOperation - Synchronous and asynchronous timeouts reference:

When you use the Web Services Invocation Framework (WSIF) with the Java Messaging Service (JMS) you can set timeouts for synchronous and asynchronous operations.

Default values for these timeouts are defined in the WSIF properties file:

```
# maximum number of milliseconds to wait for a response to a synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined on invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

If you use these default values, a synchronous request (such as a WSIFOperation interface executeRequestResponseOperation method call) times out after ten seconds, and an asynchronous request (such as a WSIFOperation interface executeRequestResponseAsync method call) times out after sixty seconds.

Note:

The code that processes both of these timeout values uses milliseconds as its unit of time. The WSIFProperties class getAsyncTimeout method multiplies the wsif.asyncrequest.timeout value by 1000, to convert the value from seconds to milliseconds.

You can override these default values for a given request by setting a JMS property on the operation request with the `<jms:property>` and `<jms:propertyValue>` WSDL elements. Set the name of the property to be the name of the timeout from the WSIF properties file.

The following example sets synchronous requests to time out after two minutes (120 seconds):

```
<jms:propertyValue name="wsif.syncrequest.timeout" type="xsd:string" value="120000"/>
```

and the following example disables asynchronous timeouts (a value of zero means wait forever):

```
<jms:propertyValue name="wsif.asyncrequest.timeout" type="xsd:string" value="0"/>
```

When an asynchronous timeout expires, no listener or message data base waiting for the response is notified. The asynchronous timeout is only used to tell the correlation service that the stored WSIFOperation can be deleted.

Chapter 9. Class loading

Class loaders affect the packaging of applications and the run-time behavior of packaged applications deployed on application servers.

1. Read about class loaders. The article ″“Class loading: Resources for learning” on page 504″ points to additional sources.
2. If necessary, migrate class-loader Module Visibility Mode settings for Version 4.0.x applications to Version 5.0 application or WAR class-loader policies.
3. If an application module uses a resource, create a resource provider that specifies the directory name of the resource drivers. Do not specify the resource JAR file names. All JAR files in the specified directory will be added into the class path of the WebSphere Application Server extensions class loader.
4. Configure class loaders of an application server for the run-time environment.
   a. Click **Servers > Application Servers >***server_name* and, on the settings page for an application server, set the application class-loader policy and application class-loader mode.

      The application class-loader policy controls the isolation of applications running in the system. When set to SINGLE, applications are not isolated; a single application class loader is used to contain all EJB modules, dependency JAR files, and shared libraries in the system. When set to MULTIPLE, applications are isolated from each other; each application receives its own class loader to load that application's EJB modules, dependency JAR files, and shared libraries.

      The application class-loader mode specifies the class-loader mode when the application class-loader policy is SINGLE. PARENT_FIRST causes the class loader to first delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. PARENT_LAST causes the class loader to first attempt to load classes from its local class path before delegating the class loading to its parent. This allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

   b. On the settings page for an application server, click **Classloader**. On the Classloader page, click **New**.
   c. On the settings page for a class loader, specify the class-loader 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 classpath. PARENT_LAST causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Then, click **OK**.
   d. On the settings page for a class loader, click **Libraries**. From the Library Ref page, click **Add**. On the settings page for a library reference, specify variables for the library reference as needed and click **OK**. Repeat the previous step until you define a library reference instance for each library file that your application needs. To define a library reference, you must first define one or more shared libraries.
5. When configuring an installed enterprise application for deployment in the run-time environment, set the class-loader mode and the WAR class-loader policy.
6. When configuring an installed Web module for deployment in the run-time environment, set the class-loader mode.

Class loaders

Class loaders are part of the Java virtual machine (JVM) code and are responsible for finding and loading class files. Class loaders affect the packaging of applications and the run-time behavior of packaged applications deployed on application servers.

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 JVM.

The bootstrap class loader uses the boot classpath (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 J2EE APIs of the WebSphere Application Server product (inside j2ee.jar). Because the J2EE APIs are in this class loader, you can add libraries that depend on J2EE APIs to the classpath system property to extend a server's classpath. However, a preferred method of extending a server's classpath is to add a shared library.

2. A **WebSphere-specific extensions class loader**.

   The WebSphere extensions class loader loads the WebSphere 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 used to load classes. Each directory in the ws.ext.dirs classpath and every JAR file or ZIP file in these directories is added to the classpath used by this class loader.

   The WebSphere extensions class loader also loads resource provider classes into a server if an application module installed on the server refers to a resource that is associated with the provider and if the provider specifies the directory name of the resource drivers.
3. One or more **application module class loaders** that load elements of enterprise applications running in the server.

   The application elements can be Web modules, EJB modules, resource adapters, and dependency JAR files. Application class loaders follow J2EE class-loading rules to load classes and JAR files from an enterprise application. The WebSphere run time enables you to associate a shared library classpath with an application.

Each class loader is a child of the class loader above it. 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 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 occurs. Once a class is loaded by a class loader, any new classes that it tries to load reuse the same class loader or go up the precedence list until the class is found.

Class-loader isolation policies

The number and function of the application module class loaders depends on the class-loader policies 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 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 application's WAR class-loader policy.

  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.

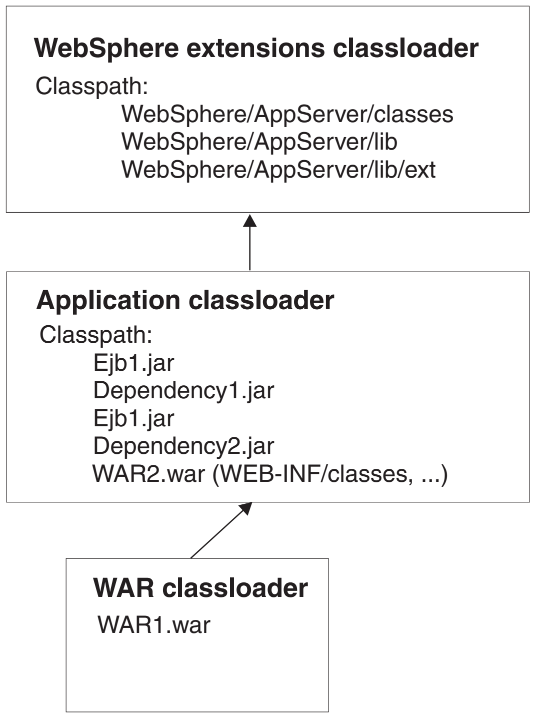
Note: WebSphere server class loaders never load application client modules.

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 used for loading that application's EJB modules, dependency JAR files, and shared libraries.

This application class loader can load each application's Web modules if that WAR module's class-loader policy is also set to APPLICATION. If the WAR module's class-loader policy is set to APPLICATION, then the application's loader loads the WAR module's classes. If the WAR class-loader policy is set to MODULE, then each WAR module receives its own class loader.

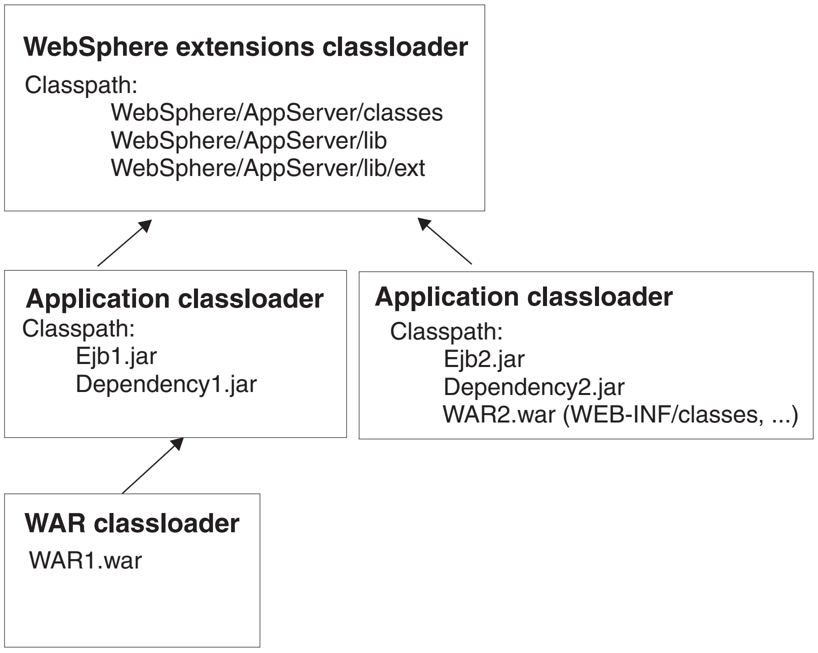
The following example shows that when the application class-loader policy is set to SINGLE, a single application class loader loads all EJB modules, dependency JAR files, and shared libraries of all applications on the server. The single application class loader can also load Web modules if an application has its WAR class-loader policy set to APPLICATION. Applications having 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's 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

There are two possible values for a class-loader mode:

PARENT_FIRST
: The PARENT_FIRST class-loader mode causes the class loader to first delegate the loading of classes to its parent class loader before attempting to load the class from its local classpath. This is the default for class-loader policy and for standard JVM class loaders.

PARENT_LAST
: The PARENT_LAST class-loader mode causes the class loader to first attempt to load classes from its local classpath before delegating the class loading to its parent. This policy allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

The following settings determine a class loader's mode:

- 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.

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***> Classloader**.

Classloader ID

States a string unique to the server identifying the class-loader instance. The product assigns the identifier.

Classloader Mode

Specifies the class-loader mode when the application class-loader policy is SINGLE. 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_LAST causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent; this allows 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* **> Classloader >***class_loader_ID*.

Classloader ID

States a string unique to the server identifying the class-loader instance. The product assigns the identifier.

Data type	String

Classloader Mode

Specifies the class-loader mode when the application class-loader policy is SINGLE. 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_LAST causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent; this allows 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

Migrating the class-loader Module Visibility Mode setting

WebSphere Application Server Version 4.0.x had a server-wide configuration setting called **Module Visibility Mode**. For Version 5.0, you use application or WAR class-loader policies instead of module visibility modes. The Version 5.0 policies provide additional flexibility because you can configure applications running in a server for an application class-loader policy of SINGLE or MULTIPLE and for a WAR class-loader policy of APPLICATION or MODULE.

To migrate module visibility modes in your Version 4.0.x applications to their equivalents in Version 5.0, change the settings for your Version 4.0.x applications and modules to the Version 5.0 values shown in the table below.

Version 4.0.x module visibility mode	**Version 5.0 application class-loader policy**	**Version 5.0 WAR class-loader policy**
Server	SINGLE	APPLICATION
Compatibility	SINGLE	MODULE
Application	MULTIPLE	APPLICATION
Module*	MULTIPLE	MODULE
J2EE	MULTIPLE	MODULE

*There is no exact equivalent for the Version 4.0.x Module mode because it isolated EJB modules within an application.

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.

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.

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^{TM} Platform Specification
- Sun's J2EE^{TM} Extension Mechanism Architecture

Chapter 10. 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. Queries specified in the deployment descriptor are compiled into SQL during deployment. Dynamic queries require the interface provided by WebSphere Application Server Enterprise.

WebSphere's EJB query language is compliant with the EJB QL defined in Sun's EJB 2.0 specification and has additional capabilities as listed in the topic Comparison of EJB 2.0 specification and WebSphere Query Language.

In your WebSphere application, you can define an EJB query in the following ways:
- **Application Assembly Tool.** When assembling an EJB 2.0 entity bean, specify the `<ejb-ql>` tag for the finder() or select() method.
- **WebSphere Studio 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. The dynamic query API is provided as an Enterprise Extension to WebSphere Application Server.

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 EJBs 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 6. DeptBean schema

Entity bean name (EJB name)	DeptEJB (not used in query)

Table 6. DeptBean schema (continued)

Abstract schema name	DeptBean
Implementation class	com.acme.hr.deptBean (not used in query)
Persistent attributes (cmp fields)	• deptno - Integer (key) • name - String • budget - BigDecimal
Relationships	• emps - 1:Many with EmpEJB • mgr - Many:1 with EmpEJB

Table 7. EmpBean schema

Entity bean name (EJB name)	EmpEJB (not used in query)
Abstract schema name	EmpBean
Implementation class	com.acme.hr.empBean (not used in query)
Persistent attributes (cmp fields)	• empid - Integer (key) • name - String • salary - BigDecimal • bonus - BigDecimal • hireDate - java.sql.Date • birthDate - java.util.Calendar • address - com.acme.hr.Address
Relationships	• dept - Many:1 with DeptEJB • manages - 1:Many with DeptEJB

Address is a serializable object used as cmp field in EmpBean. The definition of address is as follows:

```
   public class com.acme.hr.Address  extends Object implements Serializable {
public String street;
public String state;
public String city;
public Integer zip;
     public double distance (String start_location) { ... } ;
     public  String format ( ) { ... } ;
}
```

The following query returns all departments:

```
SELECT OBJECT(d) FROM DeptBean d
```

The following query returns departments whose name begins with the letters ″Web″. Sort the result by name:

```
SELECT OBJECT(d) FROM DeptBean d WHERE  d.name LIKE  'Web%' ORDER BY d.name
```

The keywords SELECT and FROM are shown in uppercase in the examples but are case insensitive. If a name used in a query is a reserved word, the name must be enclosed in double quotes to be used in the query. There is a list of reserved words later in this document. 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 referes to the corresponding value of the finder or select method. Query parameters are numbered starting with 1:

```
SELECT OBJECT (e) FROM EmpBean e WHERE e.dept.mgr.name= ?1
```

This query shows navigation of a multivalued relationship and returns all departments that have an employee that earns at least 50000 but not more than 90000:

```
SELECT OBJECT(d) FROM DeptBean d,  IN (d.emps) AS e
WHERE e.salary BETWEEN 50000 and 90000
```

There is a join operation implied in this query between each department object and its related collection of employees. If a department has no employees, the department does not appear in the result. If a department has more than one employee that earns more than 50000, that department appears multiple times in the result.

The following query eliminates the duplicate departments:

```
SELECT DISTINCT OBJECT(d) from DeptBean d,  IN (d.emps) AS e  WHERE e.salary > 50000
```

Find employees whose bonus is more than 40% of their salary:

```
SELECT OBJECT(e) FROM EmpBean e where e.bonus > 0.40 * e.salary
```

Find departments where the sum of salary and bonus of employees in the department exceeds the department budget:

```
SELECT OBJECT(d) FROM DeptBean d where d.budget <
( SELECT SUM(e.salary+e.bonus) FROM IN(d.emps) AS e )
```

A query can contain DB2 style date-time arithmetic expressions if you use java.sql.* datatypes as CMP fields and your datastore is DB2. Find all employees who have worked at least 20 years as of January 1st, 2000:

```
SELECT OBJECT(e) FROM EmpBean e where year(  '2000-01-01' - e.hireDate ) >= 20
```

If the datastore is not DB2 or if you prefer to use java.util.Calendar as the CMP field, then you can use the java millsecond value in queries. The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate <  631180800232
```

Find departments with no employees:

```
SELECT OBJECT(d) from DeptBean d where d.emps IS EMPTY
```

Find all employees whose earn more than Bob:

```
SELECT OBJECT(e) FROM EmpBean e, EmpBean b
WHERE b.name = 'Bob' AND e.salary + e.bonus > b.salary + b.bonus
```

Find the employee with the largest bonus:

```
SELECT OBJECT(e) from EmpBean e  WHERE e.bonus =
(SELECT MAX(e1.bonus) from EmpBean e1)
```

The above queries all return EJB objects. A finder method query must always return an EJB Object for the home. A select method query can in addition return CMP fields or other EJB Objects not belonging to the home.

The following would be valid select method queries for EmpBean. Return the manager for each department:

```
SELECT  d.mgr FROM DeptBean d
```

Return department 42 manager's name:

```
SELECT  d.mgr.name FROM DeptBean d WHERE  d.deptno = 42
```

Return the names of employees in department 42:

```
SELECT e.name FROM EmpBean e WHERE  e.dept.deptno=42
```

Another way to write the same query is:

```
SELECT e.name from DeptBean d, IN (d.emps) AS e WHERE d.deptno=42
```

Finder and select queries allow only a single CMP field or EJBObject in the SELECT clause.

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 datastore.

A numeric literal can be any of the following:
- an exact value such as 57, -957, +66
- any value supported by Java long
- a decimal literal such as 57.5, -47.02
- an approximate numeric value such as 7E3, -57.4E-2

A decimal or approximate numeric value must be in the range supported by the underlying persistent datastore.

A boolean literal can be the keyword TRUE or FALSE and is case insensitive.

Input parameters

Input parameters are designated by the question mark followed by a number; For example: `?2`

Input parameters are numbered starting at 1 and correspond to the arguments of the finder or select method; therefore, a query must not contain an input parameter that exceeds the number of input arguments.

An input parameter can be a primitive type of byte, short, int, long, float, double, boolean, char or wrapper types of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Char, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp or an EJBObject.

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.

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

In some datastores, a zero length string value (”) is treated as a null value and affects the results of queries. Some datastores perform division between two integer values using integer arithmetic rules and other datastores use non integer rules. This also can affect the results of queries. For portability, avoid the use of zero length string values and division of integer values in an EJB query.

Null value semantics: The following describe the semantics of NULL values:
- Comparison or arithmetic operations with an unknown (NULL) value yield an unknown value
- Path expressions that contain NULL evaluate to NULL
- 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 evalution 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
```

If dept or budget is NULL evaluation of `e.dept.budget` is null returns TRUE and the employee object is returned.

```
select object(e) from EmpBean e ,  in (e.dept.emps) e1 where e1.salary > 10
```

If dept is NULL, then the multivalued path expression `e.dept.emps` results in an empty collection (not a collection that contains a NULL value). An employee with a null dept value will not be returned.

```
select object(e) from EmpBean e where  e.dept.emps is empty
```

If dept is NULL the evaluation of the predicate in unknown and the employee object is not returned.

```
select object(e) from EmpBean e , EmpBean e1  where e member of  e1.dept.emps
```

If dept is NULL evaluation of the member of predicate returns unknown and the employee is not returned.

Date time arithmetic and comparisons: DATE, TIME and TIMESTAMP values may be compared with another value of the same type. Comparisons are chronological. Date time values can also be incremented, decremented, and subtracted.

If the datastore is DB2, then DB2 string representation of DATE, TIME and TIMESTAMP types can also be used. A string representation of a date or time can use ISO, USA, EUR or JIS format. A string representation of a timestamp uses ISO format.

Format	Date format	Date examples	Time format	Time examples
ISO	yyyy-mm-dd	1987-02-24 1987-2-24	hh.mm.ss	13.50.00 13.50
USA	mm/dd/yyyy	2/24/1987	hh:mm AM or PM	1:50 pm 02:10 AM
EUR	dd.mm.yyyy	24.02.1987 24.2.1987	hh.mm.ss	13.50.00 13.55
JIS	yyyy-mm-dd	1987-02-24	hh:mm:ss	13:50 13:50:05

Example 1: Date time arithmetic comparisons

```
e.hiredate > '1990-02-24'
```

The timestamp of February 24th, 1990 1:50 pm can be represented as follows:

```
'1990-02-24-13.50.00.000000'  or
'1990-02-24-13.50.00'
```

If the datastore is DB2, DB2 decimal durations can be used in expressions and comparisons. A date duration is a decimal(8,0) number that represents the difference between two dates in the format YYYYMMDD. A time duration is a decimal(6,0) number that represents the difference between two time values as HHMMSS. A timestamp duration is a decimal(20,6) number representing the differences between two timestamp values as YYYYMMDDHHMMSS.ZZZZZZ (ZZZZZZ is the number of microseconds and is to the right of the decimal point) .

Two date values (or time values or timestamp values) can be subtracted to yield a duration. If the second operand is greater than the first the duration is a negative decimal number. A duration can be added or subtracted from a datetime value to yield a new datetime value.

Example 2: Date time arithmetic comparisons

`DATE('3/15/2000') - '12/31/1999'` results in a decimal number 215 which is a duration of 0 years, 2 months and 15 days.

Durations are really decimal numbers and can be used in arithmetic expressions and comparisons.

`( DATE('3/15/2000') - '12/31/1999' ) + 14 > 215` evaluates to TRUE.

`DATE('12/31/1999') + DECIMAL(215,8,0)` results in a date value 3/15/2000.

`TIME('11:02:26') - '00:32:56'` results in a decimal number 102930 which is a time duration of 10 hours, 29 minutes and 30 seconds.

`TIME('00:32:56') + DECIMAL(102930,6,0)` results in a time value of 11:02:26.

`TIME('00:00:59') + DECIMAL(240000,6,0)` results in a time value of 00:00:59.

`e.hiredate + DECIMAL(500,8,0) > '2000-10-01'` means compare the hiredate plus 5 months to the date 10/01/2000.

Basic predicates

Basic predicates can be of two forms

```
expression-1  comparison-operator  expression-2
expression-3  comparison-operator ( subselect )
```

The subselect must not return more than one value and the subselect can not return a type of an EJB reference. Boolean types and reference types only support = and <> comparisons.

Example: Basic predicates

```
d.name='Java Development'
e.salary > 20000
e.salary > ( select avg(e.salary) from EmpBean e)
```

Quantified predicates

A quantified predicate compares a value with a set of values produced by a subselect.

```
expression   comparison-operator   SOME  |  ANY  |   ALL    ( subselect )
```

The expression must not evaluate to a reference type.

When SOME or ANY is specified the result of the predicate is as follows:

- TRUE if the comparison is true for at least one value returned by the subselect.
- FALSE if the subselect is empty or if the comparison is false for every value returned by the subselect.
- UNKNOWN if the comparison is not true for all of the values returned by the subselect and at least one of the comparisons is unknown because of a null value.

When ALL is specified the result of the predicate is as follows:

- TRUE if the subselect returns empty or if the comparison is true to every value returned by the subselect.
- FALSE if the comparison is false for at least one value returned by the subselect.
- UNKNOWN if the comparison is not false for all values returned by the subselect and at least one comparison is unknown because of a null value.

BETWEEN predicate

The BETWEEN predicate determines whether a given value lies between two other given values.

```
expression  [NOT]  BETWEEN expression-2  AND expression-3
```

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 built-in 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
- SQRT
- CONCAT
- LENGTH
- LOCATE
- SUBSTRING

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 >
```

```
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 following function converts the argument to an integer n by truncation and returns the date that is n-1 days after January 1, 0001:

```
DATE ( < date string > ) -> Date
DATE (  < any numeric datatype>) -> Date
```

The following function returns date portion of a timestamp:

```
DATE( timestamp ) -> Date
DATE ( < timestamp-string > ) -> Date
```

The following function converts number to decimal with optional precision p and scale s.

```
DECIMAL ( < any numeric datatype > [, p [ ,s ] ] ) -> Decimal
```

The following function converts string to decimal with optional precision p and scale s.

```
DECIMAL ( < string > [ , p [ , s ] ] ) -> Decimal
```

```
DOUBLE ( < any numeric datatype > ) -> Double
DOUBLE ( < string > ) -> Double
```

```
FLOAT ( < any numeric datatype > ) -> Double
FLOAT ( < string > ) -> Double
```

Float is a synonym for DOUBLE.

```
INTEGER ( < any numeric datatype > ) -> Integer
INTEGER ( < string > ) -> Integer
```

```
REAL ( < any numeric datatype > ) -> Float
```

```
SMALLINT ( < any numeric datatype ) -> Short
SMALLINT ( < string > ) -> Short
```

```
TIME ( < time > ) -> Time
TIME ( < time-string  > ) -> Time
TIME ( < timestamp > ) -> Time
TIME ( < timestamp-string  > ) -> Time
```

```
TIMESTAMP ( < timestamp > ) -> Timestamp
TIMESTAMP ( < timestamp-string > ) -> Timestamp
```

String functions

```
CONCAT ( <string>, <string>  ) -> String
```

The following function returns a character string representing absolute value of the argument not including its sign or decimal point. For example, `digits( -42.35)` is ″4235″.

```
DIGITS ( Decimal d  ) -> String
```

The following function returns the length of the argument in bytes. If the argument is a numeric or datetime type, it returns the length of internal representation.

```
LENGTH ( < string >  ) -> Integer
```

The following function returns a copy of the argument string where all upper case characters have been converted to lower case.

```
LCASE ( < string > ) -> String
```

The following function returns the starting position of the first occurrence of argument 1 inside argument 2 with optional start position. If not found, it returns 0.

```
LOCATE ( String s1 , String s2  [, Integer start ] ) -> Integer
```

The following function returns a substring of s beginning at character m and containing n characters. If n is omitted, the substring contains the remainder of string s. The result string is padded with blanks if needed to make a string of length n.

```
SUBSTRING ( String s ,  Integer m [ , Integer n ] ) -> String
```

The following function returns a copy of the argument string where all lower case characters have been converted to upper case.

```
UCASE ( < string > ) -> String
```

Date - time functions

The following function returns the day portion of its argument. For a duration, the return value can be -99 to 99.

```
DAY (  Date ) ->  Integer
DAY ( < date-string > ) ->  Integer
DAY ( < date-duration > ) -> Integer
DAY ( Timestamp  ) ->  Integer
DAY ( < timestamp-string > ) ->  Integer
DAY ( < timestamp-duration > ) -> Integer
```

The following function returns one more than number of days from January 1, 0001 to its argument.

```
DAYS ( Date  ) ->  Integer
DAYS ( < Date-string > ) ->  Integer
DAYS ( Timestamp  ) ->  Integer
DAYS ( < timestamp-string > ) ->  Integer
```

The following function returns the hour part of its argument. For a duration, the return value can be -99 to 99.

```
HOUR ( Time ) -> Integer
HOUR ( < time-string > ) -> Integer
HOUR ( < time-duration > ) ->  Integer
HOUR ( Timestamp ) -> Integer
HOUR ( < timestamp-string > ) -> Integer
HOUR ( < timestamp-duration >  ) -> Integer
```

The following function returns the microsecond part of its argument.

```
MICROSECOND ( Timestamp ) -> Integer
MICROSECOND ( < timestamp-string > ) -> Integer
MICROSECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the minute part of its argument. For a duration, the return value can be -99 to 99.

```
MINUTE ( Time ) -> Integer
MINUTE ( < time-string > ) -> Integer
MINUTE ( < time-duration > ) ->  Integer
MINUTE ( Timestamp ) -> Integer
MINUTE ( < timestamp-string > ) -> Integer
MINUTE ( < timestamp-duration >  ) -> Integer
```

The following function returns the month portion of its argument. For a duration, the return value can be -99 to 99.

```
MONTH (  Date ) ->  Integer
MONTH ( < date-string > ) ->  Integer
MONTH ( < date-duration > ) -> Integer
MONTH ( Timestamp  ) ->  Integer
MONTH ( < timestamp-string > ) ->  Integer
MONTH ( < timestamp-duration > ) -> Integer
```

The following function returns the second part of its argument. For a duration, the return value can be -99 to 99.

```
SECOND ( Time ) -> Integer
SECOND ( < time-string > ) -> Integer
SECOND ( < time-duration > ) ->  Integer
SECOND ( Timestamp ) -> Integer
SECOND ( < timestamp-string > ) -> Integer
SECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the year portion of its argument. For a duration, the return value can be -9999 to 9999.

```
YEAR (  Date ) ->  Integer
YEAR ( < date-string > ) ->  Integer
YEAR ( < date-duration > ) -> Integer
YEAR ( Timestamp  ) ->  Integer
YEAR ( < timestamp-string > ) ->  Integer
YEAR ( < timestamp-duration > ) -> Integer
```

Aggregation functions

Queries that return aggregate values can only be used with the dynamic query interface available in WebSphere Application Server Enterprise. However, aggregation functions can be used in non-dynamic queries if the aggregation function is used in a subselect or HAVING clause.

Aggregation functions operate on a set of values to return a single scalar value. The following is an example of an aggregation:

```
SELECT  SUM (e.salary + e.bonus) FROM EmpBean e WHERE e.dept.deptno =20
```

This computes the total salary and bonus for department 20.

The aggregation functions are avg, count, max, min and sum. The syntax of an aggregation function is as follows:

```
aggregation-function  (    [ ALL |  DISTINCT ]  expression )
```

or:

```
COUNT( * )
```

The DISTINCT option eliminates duplicate values before applying the function. ALL is the default and does not eliminate duplicates. Null values are ignored in computing the aggregate function except for COUNT(*) which returns a count of all elements in the set.

MAX and MIN can apply to any numeric, string or datetime datatype and returns the same datatype. SUM and AVG take a numeric type as input. SUM and AVG return numeric type. The actual numeric type returned by SUM and AVG depends on the underlying datastore. COUNT can take any datatype and returns an integer.

If you are using an Informix datastore, the argument to COUNT must be an asterisk or a single valued path expression. The argument to SUM, AVG, MIN, or MAX used with DISTINCT must be a single valued path expression.

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 subquery. This set can be divided into groups and the aggregation function applied to each group. This is done by using a GROUP BY clause in the subquery. The GROUP BY clause defines grouping members which is 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.

Finder or select queries can not return aggregation function values. In other words, aggregation functions can not appear in the top level SELECT of a finder or select query but can be used in subqueries.

Example: Aggregation functions

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

The above query computes the average salary for each department.

In dividing a set into groups, a NULL value is considered equal to another NULL value.

Just as the WHERE clause filters tuples 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 average salary for departments that have more than 3 employees and the department number is greater than 5.

It is possible to have 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 as follows:

```
SELECT  [  ALL | DISTINCT  ]
 { single-valued-path-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. The keyword DISTINCT can be used to eliminate duplicate references.

For a query that defines a finder method the query must return an object type consistent with the home for which the finder method associated with the query. In other words, 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 above query returns a collection of name values for those departments having employees earning less than 20000.

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
```

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 restrictions

An EJB query is compiled into an SQL query and executed against the underlying datastore based on schema mapping of the abstract bean to the datastore schema. The semantics of comparison and arithmetic operations are that of the underlying datastore. In the case of SQL, note that two strings are equal if the shorter string padded with blanks equals the longer string. For example, 'A' is equal to 'A '. This differs from the equality of strings in the Java language. Arithmetic overflow operations are an error in SQL.

A cmp field can not be used in comparison operations or predicates (except for LIKE) if that cmp field is mapped to a long varchar or large objects (LOB) column or any other column type for which the database server does not support predicates or comparison operations.

A cmp field of any type can be used in a SELECT clause. Fields that can be used in predicates, grouping, or ordering operations must be of the types listed below:

- Primitive types : byte, short, int, long, float, double, boolean, char
- Object types: Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Char, java.util.Calendar , java.util.Date
- JDBC types: java.sql.Date, java.sql.Time, java.sql.Timestamp

The field must be mapped to a table column that is compatible in type either by using a ″top-down″ default mapping generated by the WebSphere deploy tool, or using a ″meet-in-the-middle″ mapping between compatible types.

In order to search on attributes of a cmp field that is a user-defined value object, you should use a ″meet-in-the-middle″ mapping and use a composer to map each attribute to a compatible column. The default ″top-down″ mapping stores the object as a serialized object in a column of type blob, which does not allow searching.

If a cmp field is mapped to a column using a ″meet-in-the-middle″ mapping with a converter, that field can only be used with the NULL predicate or with basic predicates of the following form:

```
 path-expression  <comparison>  literal_value
 path-expression  <comparison>  input_parameter
```

In this situation, the converter method toData( ) is called to convert the right-hand side of the predicate to an SQL value.

Example of allowable predicate on a cmp field with user defined converter:

```
e.name = 'Chris'
e.name > ?1
e.name IS NULL
```

Examples of unallowable predicates:

```
substring( e.name, 1, 3 ) = 'ABC'
e.salary >  d.budget
```

A converter should preserve equality, collating sequence and null values when doing a conversion. Otherwise cmp fields created by the converter should not be used in WHERE, GROUP, HAVING or ORDER clauses of a query.

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]
         [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) }

select_clause_eex ::= SELECT { ALL | DISTINCT }  [ selection , ]*  selection

selection  ::= { expression [[AS] id ]  |  subselect }

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

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 ) |
        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 2.0 specification and WebSphere query language

Item	EJB 2.0 specification	WebSphere Query	WebSphere Enterprise (Dynamic) Query
Bean methods	no	no	yes
Calendar comparisons	yes	yes	yes
Delimited identifiers	no	yes	yes
Dependent Value attributes	no	yes	yes
Dependent Value methods	no	no	yes
Dynamic Query APIs	no	no	yes
EXISTS predicate	no	yes	yes
Inheritance	no	yes	yes
Multiple element select clauses	no	no	yes
Order by	no	yes	yes
Scalar functions	yes *	yes	yes
Select clause	required	optional	optional
SQL Date/time expressions	no	yes	yes
String comparisons	= and <> only	= <> > <	= <> > <

Item	EJB 2.0 specification	WebSphere Query	WebSphere Enterprise (Dynamic) Query
Subqueries, aggregations, group by, and having clauses	no	yes	yes

* EJB 2.0 defines the following scalar functions: abs, sqrt, concat, length, locate and substring. WebSphere query and dynamic query support additional scalar functions as listed in the topic, EJB query: Scalar functions.

Chapter 11. Internationalizing applications

For your application to be used in multiple regions around the world, its user interfaces will need to support multiple locales and time zones. IBM WebSphere Application Server supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

- If you are new to internationalization, read ″Internationalization″ before you continue.
- For general information about internationalization, see ″Resources for learning.″

1. Identify localizable text in your application.
2. Create the message catalogs necessary for the locales to be supported by your application.
3. In your application code, compose the language-specific strings for output.
4. 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 JAR.
6. Using the Assembly Toolkit, assemble the application modules and the deployment JAR into a J2EE application.
7. Deploy and manage the application.

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. Internationalization techniques have traditionally been expensive and difficult to implement, so they have been applied only to major development efforts. However, given the rise in distributed computing and in the use of the World Wide Web, application developers have been pressured to internationalize a much wider variety of applications. This requires making internationalization techniques much more accessible to application developers.

In an application that is not internationalized, the interface that the user sees is unalterably written into the application code. On the other hand, localizing the displayed strings adds a layer of abstraction into the design of the application. Instead of simply printing an error message, an internationalized 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, then, the application looks up the key in the configured message catalog. Each message catalog is a list of keys with associated strings. Different message catalogs provide strings for the different languages supported. The application looks up the key in the appropriate catalog, retrieves the corresponding error message in the requested language, and prints this 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 itself (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 itself needs no further modification.

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. In a localized application, the locale also indicates the message catalog from which an application is to retrieve message strings. 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.

Identifying localizable text

1. Determine which elements of the application need to be translated. 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 be used 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 itself 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 subclass of java.util.ResourceBundle 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 in without modifying the application code.

1. For each string identified for localization, add a line to the message catalog that lists the string's 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
```

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 535.

Localization API support

The package *com.ibm.websphere.i18n.localizabletext* 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:

- 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 WebSphere 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 object's format method. 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 call, LocalizableTextFormatterEJBFinder, is made to retrieve a reference to the formatter bean.
3. Information from the LocalizableTextFormatter instance, including the client's time zone and locale, 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 package com.ibm.websphere.i18n.localizabletext, 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.

LocalizableTextFormatter extends java.lang.Object 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

LocalizableTextFormatter supports the following constructors:
- LocalizableTextFormatter()
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)

The LocalizableTextFormatter instance must have certain values, such as resource-bundle name, key, and the name of the formatting application. If you do not pass these values in by using the second constructor listed previously, you can set them separately by making the following calls:
- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)

You can use a fourth method, setArguments(Object[] args), to set optional localization values after construction. See Processing of application-specific values at the end of this article. For a usage example, see ″Composing complex strings.″

API for formatting text

The formatting methods in LocalizableTextFormatter 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 macine. All four methods throw LocalizableException objects.

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, 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 apppropriate under some circumstances. In order 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 the issue of 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*; this refers to the name of the formatting application, which is not necessarily the name of the application in which the value is set.

Caching of messages

LocalizableTextFormatter can optionally cache formatted messages so that they do not have to be reformatted when needed again. By default, caching is not enabled, but use `LocalizableTextFormatter.setCacheSetting(true)` to enable caching. When caching is enabled and the format method is called, the method determines whether the message has already been 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 have been cached, those messages remain in the cache until the cache is cleared by a call to LocalizableTextformatter.clearCache(). 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 can be ignored unless the LocalizableTextFormatter instance throws 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 one 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 LocalizableTextFormatter or independently implement some or all of the same localizable-text interfaces. As a minimum, your class must implement java.io.Serializable 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 application that is requesting a localized message can specify the locale and time zone for which the message is to be formatted, 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 LocalizableTextFormatter, 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:

- Compose complex strings from variable substrings
- Customize the formatting of strings, taking into account 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 by calling the setArguments(Object[]) method. For a usage example, see ″Composing complex strings.″

   Note: 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 by key. Variable components are designated by curly 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 itself be localized, you can create a nested LocalizableTextFormatter instance for it 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 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 itself can be localized. You can do this 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 localization values as needed.

1. If needed, customize the formatting behavior.
2. In application code, call the appropriate format method.

You can provide fallback behavior for use if the appropriate message catalog is not available at formatting time.

The following code generates a localized string. If the formatting fails, the application retrieves and uses a fallback string instead of the localized string:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType){
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ...
   ltf.setFallBackString("Enter account number: ");
   try {
      msg = new Label(ltf.format(this.applicationLocale), Label.CENTER);
   }
   catch (LocalizableException le) {
      msg = new Label(ltf.getFallBackString(), Label.CENTER);
   }
   ...
}
```

When the application is finished, deploy your application.For more information, see "Preparing the localizable-text package for deployment."

Customizing the behavior of a formatting method

You can customize formatting behavior by passing your own formatter classes into a LocalizableTextFormatter instance through an array of optional values. This enables you to take variables other than locale and time zone into account 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 format() 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 java.text.DateFormat as well as constants defined within LocalizableTextDateTimeArgument itself.

Preparing the localizable-text package for deployment

Write code to compose the language-specific strings.

The LocalizableTextEJBDeploy tool is used to create a deployment JAR for the Localizable Text service. You must deploy the enterprise bean in each enterprise application that requires support for localized text.

1. Make sure the **LocalizableTextEJBDeploy** tool (ltext.jar) exists in the lib directory under the product's installation root directory.
2. Set up a working directory for the **LocalizableTextEJBDeploy** tool to use. You will 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, the generated JNDI name will be incorrect.

   To deploy the bean on multiple hosts and servers, run the tool for each host/server combination. This generates a unique JNDI name for each deployment. After the tool is run, a deployment JAR is located in the working directory you specified.

Using the Assembly Toolkit, assemble the deployment JAR in an enterprise application with other application components.

As part of preparing for deployment, verify the following:
- Add the resource bundles for your application to the EAR as files.
- Add the location of the EAR to the server's class path. This is so the resource bundles can be located on the virtual host and server.

The same deployment JAR 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 takes place. If the name cannot be resolved at run time, the format method throws 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.

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.

Programming specifications
- Java 2 SDK, Standard Edition Documentation: Internationalization

  The Java internationalization documentation from Sun Microsystems, including a list of supported locales and encodings.
- Making the WWW truly World Wide

  The W3C's effort to make World Wide Web technology work with the many writing systems, languages, and cultural conventions of the global community:
- developerWorks - Unicode

  Articles on various subjects relating to Unicode, from IBM's developerWorks.

Chapter 12. 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

Transaction support in 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).

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 (where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) use bean-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 other OTS 1.2 compliant transaction managers (for example J2EE 1.3 application servers). WebSphere applications can also be configured 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 available.

  Last participant support (of WBI Server Foundation) 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.

The ActivitySession service (of WBI Server Foundation) 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 through its LocalTransaction interface for the period of a client-scoped ActivitySession rather than just the duration of an EJB method.

Resource manager local transaction (RMLT)

A resource manager local transaction (RMLT) is a resource manager's view of a local transaction; that is, it represents a unit of recovery on a single connection that is managed by the resource manager.

Resource managers include:

- Enterprise Information Systems that are accessed through a resource adapter, as described in the J2EE Connector Architecture 1.0.
- Relational databases that are accessed through a JDBC datasource.
- JMS queue and topic destinations.

Resource managers offer specific interfaces to enable control of their RMLTs. J2EE connector resource adapters that include support for local transactions provide a LocalTransaction interface to enable applications to request that the resource adapter commit or rollback RMLTs. JDBC datasources provide a Connection interface for the same purpose.

The boundary at which all RMLTs must be complete is defined in WebSphere Application Server by a local transaction containment (LTC).

Global transactions

If an application uses two or more resources, then an external transaction manager is needed to coordinate the updates to both resource managers in a global tansaction.

Global transaction support is available to web and enterprise bean J2EE components. Enterprise bean components can be subdivided into beans that exploit container-managed transactions (CMT) or bean-managed transactions (BMT).

BMT enterprise beans and web components can use the Java Transaction API (JTA) UserTransaction interface to define the demarcation of a global transaction. The UserTransaction interface is obtained by a JNDI lookup of `java:comp/UserTransaction`. 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.
- Client applications running outside the Web and EJB containers.

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, 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, EJB 2.0 applications can use only the java:comp/UserTransaction location.

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 Specification.)

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.

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.

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 (where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) use bean-managed transactions.

You configure whether a component uses container- or bean-managed transactions by setting an appropriate value on the Transaction type deployment attribute, as described in Configuring transactional deployment attributes using the Assembly ToolkitorConfiguring transactional deployment attributes using the Application Assembly Tool. 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 bean-managed transactions.

Similarly, 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 bean-managed transactions.

Configuring transactional deployment attributes using the Assembly Toolkit

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.

This topic describes the use of the Assembly Toolkit 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 using the Assembly Toolkit, see Assembling applications with the Assembly Toolkit.

To set transactional attributes in the deployment descriptor for an application component (enterprise bean or servlet), complete the following steps:
1. Start the Assembly Toolkit.

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 Toolkit. 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***->** **Session Beans** then select the bean instance.
   - For a servlet, expand **Web Modules->** *web_application***->** *web component* then select the servlet instance.

   A property dialog notebook for the component is displayed in the property pane.
4. 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:
   - For a session bean to use container-managed transactions, set Container
   - For a session bean to use bean-managed transactions, set Bean
   - For an entity bean, set Container
   - For a Web component (servlet), set Bean
5. In the property pane, select the IBM Extensions tab.
6. Under **WebSphere Extensions**, configure J2EE component extensions attributes for extended local transaction containment. To enable management of local transaction containments, configure the following EJB 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.

   This property can be changed on WBI Server Foundation only.

   Resolver
   Specifies how the local transaction is to be resolved before the local transaction context ends: by the application through user code or by the EJB 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**.
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 navigation pane, click the Assembly Descriptor tab. The **Container Transactions** box displays a table of the methods for enterprise beans.
   b. For each method of the enterprise bean set the **Transaction attribute** attribute to an appropriate value.
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. Generate code for deployment for EJB modules or for enterprise applications that use EJB modules.
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 Assembly Toolkit to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important

    Important: Use **Run On Server** for unit testing only. Assembly Server Toolkit controls the WebSphere Application Server installation and, when an application is published remotely, the Toolkit overwrites the server configuration file for that server. Do not use on production servers.

    For instructions on remote testing, see the article "Setting Up a Remote WebSphere Application Server in WebSphere Studio V5" at http://www7b.boulder.ibm.com/wsdd/techjournal/0303_yuen/yuen.html.

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 bean-managed transactions

This topic describes how to enable a session bean or servlet to use bean-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 or servlet to use bean-managed transactions, complete the following steps:
1. Set the **Transaction type** attribute in the component's deployment descriptor to `Bean`, as described in Setting transactional attributes in the deployment descriptor.
2. Write the component code to actively manage transactions When writing the code required by a component to manage its own transactions, remember the following basic rules:
    - An instance of a stateless session bean cannot reuse the same transaction context across multiple methods called by an EJB client.
    - An instance of a stateful session bean can reuse the same transaction context across multiple methods called by an EJB client.

    The following code extract shows the standard code required to obtain an object encapsulating the transaction context. There are three basics steps involved:
    - The component class must set the value of the javax.ejb.SessionContext object reference in the setSessionContext method.
    - A javax.transaction.UserTransaction object is created by calling a lookup on "java:comp/UserTransaction".
    - The UserTransaction object is used to participate in the transaction by calling transaction methods such as begin and commit as needed. If an enterprise bean 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.*;
...
public class MyStatelessSessionBean implements SessionBean {
private SessionContext mySessionCtx =null;
...
public void setSessionContext (SessionContext ctx)throws EJBException {
mySessionCtx =ctx;
}
...
    public float doSomething(long arg1)throws FinderException,EJBException {
        UserTransaction userTran = (UserTransaction)initCtx.lookup(
           "java:comp/UserTransaction");
        ...
        //User userTran object to call transaction methods
        userTran.begin ();
        //Do transactional work
        ...
        userTran.commit ();
        ...
    }
    ...
}
```

Configuring transaction properties for an application server

Use this task to configure the transaction properties for an application server; for example, to define the location of the directory that contains the transaction log or to change default timeouts associated with transactions.

To configure the transaction properties for an application server, complete the following steps:

1. Start the Administrative console
2. In the navigation pane, select **Servers-> Manage Application Servers->** ***your_app_server*** This displays the properties of the application server, *your_app_server*, in the content pane.
3. Select the Transaction Service tab, to display the properties page for the transaction service, as two notebook pages:

   Configuration
   The values of properties defined in the configuration file. If you change these properties, the new values are applied when the application server next starts.

   Runtime
   The runtime values of properties. If you change these properties, the new values are applied immediately, but are overwritten with the Configuration values when the application server next starts.
4. Select the Configuration tab, to display the transaction-related configuration properties.
5. If you want to change the directory in which transaction logs are written, type the full pathname of the directory in the **Transaction log directory** field. You can check the current runtime value of **Transaction log directory**, by clicking the Runtime tab.

   You can also specify a size for the transaction logs, as described in the following step.

   Note: If you change the transaction log directory, you should apply the change and restart the application server as soon as possible, to minimize the risk of problems caused that might occur before the application server is restarted. For example, if a problem causes the server to fail (with in-flight transactions), the server next starts with the new log directory and is unable to automatically resolve in-flight transactions that were recorded in the old log directory.
6. If you want to change the default file size of transaction log files, modify the **Transaction log directory** field to include a file size setting, in the following format:

   ```
   directory_name;file_size
   ```

   Where
   - *directory_name* is the name of the transaction log directory
   - *file_size* is the new default size specified in bytes. The *n*K or *n*M suffix can be used to indicate kilobytes or megabytes. If you do not specify a file size value, the default value of 1M is used.

   For example, `c:\tranlogs;2M` indicates the files are to be created with 2M bytes size and stored in the directory c:\tranlogs.

In a non-production environment, you can use the transaction log directory value of `;0` to disable transaction logging. (There must be no directory name element before the size element of 0.) You should not disable transaction logging in a production environment, because this prevents recovery after a system failure and, therefore, data integrity cannot be guaranteed.

7. In the **Total transaction lifetime timeout** field, type the number of seconds a transaction can remain inactive before it is ended by the transaction service. A value of 0 (zero) indicates that there is no timeout limit.
8. In the **Client inactivity timeout** field, type the number of seconds after which a client is considered inactive and the transaction service ends any transactions associated with that client. A value of 0 (zero) indicates that there is no timeout limit.
9. Click **OK**.
10. Stop then restart the application server.

    If you change the transaction log directory configuration property to an incorrect directory name, the application server will restart but be unable to open the transaction logs. You should change the configuration property to a valid directory name, then restart the application server.

Transaction service settings

Use this page to modify transaction service settings.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Transaction Service**.

Transaction log directory

Specifies the name of a directory for this server where the transaction service stores log files for recovery. A blank value in the server configuration is expanded by the transaction log at startup as the directory (*install_root*)/tranlog/(*server_name*).

When the application running on the WebSphere product accesses more than one resource, the WebSphere product stores transaction information to properly coordinate and manage the distributed transaction. In a higher transaction load, this persistence slows down performance of the application server due to its dependency on the operating system and the underlying storage systems.

To achieve better performance, move the transaction log files to a storage device with more physical disk drives, or preferably RAID disk drives. When the log files are moved to the file systems on the raided disks, the task of writing data to the physical media is shared across the multiple disk drives. This allows more concurrent access to persist transaction information and faster access to that data from the logs. Depending upon the design of the application and storage subsystem, performance gains can range from 10% to 100%, or even more in some cases.

This change is applicable only to the configuration where the application uses distributed resources or XA transactions, for example, multiple databases and resources are accessed within a single transaction. Consider setting this property when the application server shows one or more of following signs:

- CPU utilization remains low despite an increase in transactions
- Transactions fail with several time outs
- Transaction rollbacks occur with *unable to enlist transaction* exception
- Application server hangs in middle of a run and requires the server to be restarted
- The disk on which an application server is running shows higher utilization

Data type	String
Default	Initial value is the *%WAS_HOME%*/tranlog/(*server_name*) directory and a default size of 1MB.

Recommended	Create a file system with at least 3-4 disk drives raided together in a RAID-0 configuration. Then, create the transaction log on this file system with the default size. When the server is running under load, check the disk input and output. If disk input and output time is more then 5%, consider adding more physical disks to lower the value. If disk input and output is low, but the server is still high, consider increasing the size of the log files.

Total transaction lifetime timeout

Specifies the maximum duration, in seconds, for transactions on this application server.
Any transaction that is not requested to complete before this time-out is rolled back. If set to 0, there is no time-out limit.

Data type	Integer
Units	Seconds
Default	120
Range	0 to 2 147 483 647

Client inactivity timeout

Specifies the maximum duration, in seconds, between transactional requests from a remote client.
Any period of client inactivity that exceeds this timeout results in the transaction rolling back in this application server. If set to 0, there is no timeout limit.

Data type	Integer
Units	Seconds
Default	60
Range	0 to 2 147 483 647

Enable logging for heuristic reporting

Select this property to enable the 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.
This property enables logging for heuristic reporting. If applications are configured to allow one-phase commit resources to participate in two-phase commit transactions, reporting of heuristic outcomes that occur at application server failure requires extra information to be written to the transaction log. If enabled, one additional log write is performed for any transaction that involves both one- and two-phase commit resources. No additional records are written for transactions that do not involve a one-phase commit resource.

Data type	String
Default	Cleared
Range	**Cleared** The application server does not log ″about to commit one-phase resource″ events from transactions that involve a one-phase commit resource and two-phase commit resources. **Selected** The application server does log ″about to commit one-phase resource″ events from transactions that involve a one-phase commit resource and two-phase commit resources.

Maximum Transaction Timeout

Specifies the maximum duration, in seconds, that transactions started by or propagated into this application server are allowed to execute.

Data type	Integer
Units	Seconds
Default	300
Range	0 to 2 147 040

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 unspecific transaction context as they can under a global transaction. An enterprise bean, for example, runs under an unspecified transaction context if it is deployed with a **Transaction** attribute of `Not supported` or `Never`.

The extended local transaction support provides a container-managed, implicit local transaction boundary within which application updates can be committed and their connections cleaned up by the container. Applications can then be designed with a greater degree of independence from deployment concerns. This makes using a **Transaction** attribute of `Supports` much simpler, for example, when the business logic may be called either with or without a global transaction context.

An application can follow a get-use-close pattern of connection usage regardless of whether or not the application runs under a transaction. The application can depend on the close behaving in the same way and not causing a rollback to occur on the connection if there is no global transaction.

There are many scenarios where ACID coordination of multiple resource managers is not needed. In such scenarios running business logic under a **Transaction** policy of `Not supported` performs better than if it had been run under a `Required` policy. This benefit is exploited through the **Local Transactions - Resolution-control** extended deployment setting of `ContainerAtBoundary`. With this setting, application interactions with resource providers (such as databases) are managed within implicit RMLTs that are both started and ended by the container. The RMLTs are committed by the container at the configured **Local Transactions - Boundary**; for example at the end of a method. If the application returns control to the container by an exception, the container rolls back any RMLTs that it has started.

This usage applies to both servlets and enterprise beans.

Use local transactions in a managed environment that guarantees clean-up.

Applications that want to control RMLTs, by starting and ending them explicitly, can use the default **Local Transactions - Resolution-control** extended deployment setting of `Application`. In this case, the container ensures connection cleanup at the boundary of the local transaction context.

J2EE specifications that describe application use of local transactions do so in the manner provided by the default setting of **Local Transactions - Resolution-control**=`Application` and **Local Transactions - Unresolved-action**=`Rollback`. By configuring the **Local Transactions - Unresolved-action** extended deployment setting to `Commit`, then any RMLTs started by the application but not completed when the local transaction containment ends (for example, when the method ends) are committed by the container. This usage applies to both servlets and enterprise beans.

Extend the duration of a local transaction beyond the duration of an EJB component method.
The J2EE specifications restrict the use of RMLTs to single EJB methods. This restriction is because the specifications have no scoping device, beyond a container-imposed method boundary, to which an RMLT can be extended. In WBI Server Foundation, 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.

An ActivitySession is a WBI Server Foundation programming model extension that provides 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`.

Points specific to **WBI Server Foundation**

- 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 of WBI Server Foundation.

Managing active transactions

Use this task to manage transactions that are active on an application server.

You can use this task to display a snapshot of all the transactions currently running on an application server. For each transaction, the following properties are shown: its local ID, global ID, and current status. The transaction status is shown as an integer value. The values correspond to the following status:

```
0 - active
1 - marked for rollback
2 - prepared
3 - committed
4 - rolled back
5 - unknown
6 - none
7 - preparing
8 - committing
9 - rolling back
```

You can also choose to finish transactions manually.

Under normal circumstances, transactions should run and complete (commit or rollback) automatically, without the need for intervention. However, in some circumstances, you may need to finish a transaction manually. For example, you may want to finish a transaction that has become stuck polling a resource manager that you know will not become available again within the desired timeframe.

Note: If you choose to finish a transaction on an application server, it is recorded as having completed in the transaction service logs for that server, so will not be eligible for recovery during server start up. If you finish a transaction, you are responsible for cleaning up any in-doubt transactions on the resource managers affected.

To manage the active transactions for an application server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Manage Application Servers** This displays a list of application servers in the content pane.
2. In the content pane, click *your_app_server* This displays the properties of the application server, *your_app_server*.

3. In the content pane, click the **Runtime** tab. This displays the runtime properties of the application server.
4. In the Additional Properties table, select **Transaction Service** This displays the runtime properties of the Transaction Service.
5. Click **Manage Transactions**. This displays a snapshot of all the transactions currently running on the server. For each transaction, the following properties are shown: its local ID, current status, and global ID.
6. If you want to finish one or more transactions, select the checkbox provided on the entry for the transaction, then click **Finish**. Alternatively, to finish all transactions, select the checkbox in the header of the transactions table, then click **Finish**.

Managing transaction logging for optimum server availability

This topic describes some considerations and actions that you can use to manage transaction logging to help ensure that the availability of your application servers is optimized.

The transaction service writes information to the transaction log for every global transaction which involves two or more resources or is distributed across multiple servers. The transaction log is stored on disk and is used by the transaction service for recovery after a system or server crash. The transaction log for each application server consists of multiple subdirectories and files held in a single directory. You can change the directory that an application server uses to store the transaction log, as described in Configuring transaction properties for an application server.

When a global transaction is completed, the information in the transaction log is not needed anymore so is marked for deletion. Periodically, this redundant information is garbage collected and the space reused by new transactions. The log files are created of fixed size at server startup, thus no further disk space allocation is required during the lifetime of the server. The default allocation is suitable for around 4000 concurrent transactions.

If all the log space is in use when a transaction needs to save information, the transaction is rolled back and the message “WTRN0083W: The transaction log is full. Transaction rolled back.” is reported to the system error log. No more transactions can commit until more log space is made available when existing active transactions complete.

You can monitor the number of concurrent global transactions by using the performance monitoring counters for transactions. The “Global transaction commit time” counter is a measure of how long a transaction takes to complete and, therefore, how long the log is in use by a transaction. If this value is high, then transactions are taking a long time to complete, which can be due to resource manager or network failures. If you ensure this value is low, the log is more efficiently used and unlikely to become full.

You can change the default size of log files by updating the transaction log settings as described in Configuring transaction properties for an application server.

Configuring transaction aspects of servers for optimum availability

This topic describes some considerations and actions that you can take to configure transaction-related aspects of application servers for optimum availability.

To configure transaction-related aspects of application servers for optimum availability, complete the following steps:

1. Store the transaction log files on a fast disk in a highly-available file system, such as a RAID device. The transaction log may need to be accessed by every global transaction and be used for transaction recovery after a crash. Therefore, the disk the log files are being written to should be on a highly-available file system, such as a RAID device.

The performance of the disk also directly affects the transaction performance. In general, a global transaction makes two disk writes, one after the prepare phase when the outcome of the transaction is known (this information is forced to disk) and a further disk write at transaction completion. Therefore, the transaction logs should be placed on the fastest disks available and not make use of network mounted devices.

2. Mirror the transaction log files by using hardware disk mirroring or dual-ported disks. If log files have been mirrored or can be recovered, they can be used when restarting a failed server or moved to an another machine and another server started there to perform recovery.

   Hardware disk mirroring or dual-ported disks can be used by specifiying the appropriate file system directory for the transaction logs using the WebSphere Administrative Console.

3. Specify the optimum location of the transaction log directory for application servers. By default, an application server places transaction log files in a subdirectory of the installed WebSphere Application Server, where the subdirectory name is the same as the server name. For example, the default directory for an application server named `server1` on Windows is `c:\WebSphere\AppServer\tranlog\server1`.

   Note: The TRANLOG_ROOT variable was used in previous versions of WebSphere Application Server to override the default location of the transaction log, but is now deprecated. In a future version of WebSphere Application Server, transaction and compensation logs will move to a different default location and the variable TRANLOG_ROOT will be removed.

   You can define a specific location for the transaction log directory for an application server by setting the **Transaction Log Directory** property for the server.

4. Never allow more than one application server to concurrently use the same set of log files. Because the transaction logs record the state of global transactions within a server, if the logs become lost or corrupt, then transactions that are in the prepared state before failure can leave resources in an in-doubt state and prevent further updates or access to the resources by other users or servers. These transactions may need to be manually resolved by either committing or rolling back the transactions at the affected resource managers. The failed server can then be cold-started, which creates new empty transaction logs.

   If log files have been mirrored or can be recovered, they can be used when restarting the failed server or moved to an alternate server or machine and another server restarted to perform recovery, as described in the related tasks.

   Never allow more than one application server to concurrently use the same set of log files, because each server will destroy the information recorded by the other, resulting in corrupt log files that are unusable for future recovery purposes.

5. Configure application servers to always use the same listening port address at each startup. If you are running distributed transactions between multiple application servers, the remote object references saved in the transaction log need to be redirected to the originating server on recovery.

   On Application Server Network Deployment, the node agents automatically redirect such remote object references to the appropriate application servers on recovery. However, if the distributed transaction is between application servers that are not on Application Server Network Deployment, then you must handle the redirection of remote object references for transaction recovery to complete. For example, you must do this is if an application server is deployed on WebSphere Application Server (not the Network Deployment edition) and runs distributed transactions with non-WebSphere EJB or Corba servers.

   In particular, the default restart action of an application server not on Application Server Network Deployment is to use a different listening port address to the port when the server shut down. This prevents transaction recovery completing. To overcome this, you should always configure application servers to always use the same listening port address at each startup (see the ORB property com.ibm.CORBA.ListenerPort in ORB service settings that can be added to the administrative console). You may need to make similar configuration changes to other application servers involved in transactions, to be able to access those servers during recovery.

Moving a transaction log from one server to another

This topic describes some considerations and actions that you can take to move the transaction logs for an application server to another server.

To move transaction logs from one application server to another, consider the following steps:

1. Move all the transaction log files for the application server.

   5.1 + The transaction log directory for each server contains a number of files and subdirectories. When moving transaction logs from one server to another you must move all of the files and subdirectories together as a single unit; otherwise recovery may not complete resulting in data inconsistency.
2. For a server configuration where there are no distributed transactions, move the transaction logs to any server that has access to the same resource managers. For a single server or network-deployed server configuration where it is known there are no distributed transactions present in the logs, the transaction logs can be moved to any server (on any node) that has access to the same resource managers as the original server. For example, the server needs communication and valid security access to databases or message queues.

   All the transaction log files for the original server need to be moved to a directory accessible by the new server. This can be accomplished by either renaming the transaction log directory or copying all thecontents to the new server's transaction log directory before starting the new server.

   Note: To complete transaction recovery, the application server uses the resource manager configuration information in the transaction logs. However, for the application server to continue to do new work with the same resource managers, the server must have an appropriate resource manager configuration (as for the original server).
3. For a network-deployed server configuration where there are distributed transactions, move the transaction logs to a server that has the same name and host IP address, and access to the same resource managers. For a network-deployed server configuration, when it is known there are distributed transactions present in the logs, there are more restrictions. Distributed transactions that access multiple servers log information about each server involved in the transaction. This information includes the server name and the IP address of the machine on which the server is running. When recovery is taking place on server restart, the server uses this information to contact the distributed servers and similarly, the distributed servers try to contact the server with the same original name. So, if a server fails and the logs need to the recovered on an alternative server, that alternative server needs to have the same name and host IP address as the original server. The alternative server also needs to have access to the same resource managers as the original server. For example, the server needs communication and valid security access to databases or message queues.

   Note: All servers within a cell must have unique names.

   Note: To complete transaction recovery, the application server uses the resource manager configuration information in the transaction logs. However, for the application server to continue to do new work with the same resource managers, the server must have an appropriate resource manager configuration (as for the original server).

Restarting an application server on a different host

This topic describes some considerations and actions that you can take with transaction logs to restart an application server on a different host.

Moving transactions logs to a different host is similar to moving logs from one server to another, as described in Moving transaction logs from one server to another.

This involves moving an original application server on one host to an alternative server, which has access to the same resource managers, on another host. For a network-deployed server configuration, the alternative server must have the same name and host IP address as the original server.

Note: To complete transaction recovery, the application server uses the resource manager configuration information in the transaction logs. However, for the application server to continue to do new work with the same resource managers, the server must have an appropriate resource manager configuration (as for the original server).

To restart an application server on a different host, complete the following steps:
1. Ensure that the alternative application server is stopped.
2. Move all the transaction logs for the original server to the alternative application server, according to the considerations described in Moving transaction logs from one server to another.
3. Restart the alternative application server.

Interoperating transactionally between application servers

This topic describes some considerations and actions that you can take to interoperate transactionally between different types of application servers.

To interoperate transactionally with a non-WebSphere application server, WebSphere Application Server switches dynamically between WebSphere-optimized native transaction contexts and interoperable OTS contexts depending on the capability of the partner with which it is interoperating.

5.1 + To interoperate transactionally between WebSphere Application Server version 5.1 (or later) and WebSphere Application Server before version 5.0, you need to set the following system properties on application servers before version 5.0:

```
com.ibm.ejs.jts.jts.ControlSet.nativeOnly=false
com.ibm.ejs.jts.jts.ControlSet.interoperabilityOnly=true
```

For example, if you want to interoperate between application servers at WebSphere Application Server version 5.1 and WebSphere Application Server version 4.0.n, you need to set the system properties on the version 4.0.n application servers.

Troubleshooting transactions

Use this overview task to help resolve a problem that you think is related to the Transaction service.

To identify and resolve transaction-related problems, you can use the standard WebSphere Application Server RAS facilities. If you encounter a problem that you think might be related to transactions, complete the following steps:
1. Check for transaction messages in the administrative console. The Transaction service produces diagnostic messages prefixed by “WTRN”. The error message indicates the nature of the problem and provides some detail. The associated message information provides an explanation and any user actions to resolve the problem.
2. Check for Transaction messages in the activity log. Activity log messages produced by the Transaction service are accompanied by Log Analyzer descriptions.
3. Check for more messages in the application server’s stdout.log. For more information about a problem, check the stdout.log file for the application server, which should contain more error messages and extra details about the problem.
4. Check for messages related to the application server’s transaction log directory when the problem occurred.

Note: If you changed the transaction log directory and a problem caused the application server to fail (with in-flight transactions) before the server was restarted properly, the server will next start with the new log directory and be unable to automatically resolve in-flight transactions that were recorded in the old log directory. To resolve this, you can copy the transaction logs to the new directory then stop and restart the application server.

Transaction service exceptions

This topic lists the exceptions that can be thrown by the WebSphere Application Server transaction service. The exceptions are listed in the following groups:
- Standard exceptions
- Heuristic exceptions

If the EJB container catches a system exception from the business method of an enterprise bean, and the method is running within a container-managed transaction, the container rolls back the transaction before passing the exception on to the client. For more information about how the container handles the exceptions thrown by the business methods for beans with container-managed transaction demarcation, see the section *Exception handling* in the Enterprise JavaBeans 2.0 specification. That section specifies the container's action as a function of the condition under which the business method executes and the exception thrown by the business method. It also illustrates the exception that the client receives and how the client can recover from the exception.

Standard exceptions

The standard exceptions such as TransactionRequiredException, TransactionRolledbackException, and InvalidTransactionException are defined in the Java Transaction API (JTA) 1.0.1 Specification.

InvalidTransactionException
This exception indicates that the request carried an invalid transaction context.

TransactionRequiredException exception
This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.

TransactionRolledbackException exception
This exception indicates that the transaction associated with processing of the request has been rolled back, or marked for roll back. Thus the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

Heuristic exceptions

A heuristic decision is a unilateral decision made by one or more participants in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the Transaction Service. Heuristic decisions are an issue only after the participant has been prepared and the second phase of commit processing is underway. Heuristic decisions are normally made only in unusual circumstances, such as repeated failures by the transaction manager to communicate with a resource manage during two-phase commit. If a heuristic decision is taken, there is a risk that the decision differs from the consensus outcome, resulting in a loss of data integrity.

The following list provides a summary of the heuristic exceptions. For more detail, see the Java Transaction API (JTA) 1.0.1 Specification.

HeuristicRollback exception
This exception is raised on the commit operation to report that a heuristic decision was made and that all relevant updates have been rolled back.

HeuristicMixed exception
This exception is raised on the commit operation to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

UserTransaction interface - methods available

For details about the methods available with the UserTransaction interface, see the WebSphere Application Server application programming interface reference information (Javadoc) or the Java Transaction API (JTA) 1.0.1 Specification.

Chapter 13. 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 following steps outline the context of Naming in the overall application development and deployment process. Steps for this task follow:

1. Develop your application using either JNDI or CosNaming (CORBA) 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 theAssembly Toolkit. 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, see ″Lookup names support in deployment descriptors and thin clients.″
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, see ″Lookup names support in deployment descriptors and thin clients.″ For more information on configured name bindings, see ″Configured name bindings.″
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.

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.

© Copyright IBM Corp. 2003

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.

Version 5 features for name space support

The following are features of the WebSphere Application Server new to naming implementation as of Version 5:

- **Name space is distributed.**

  For additional scalability, the name space for a cell is distributed among various servers. Every server has a name server. In previous releases, there was only one name server for an entire administrative domain.

  In WebSphere Application Server versions prior to V5, all servers shared the same default initial context, and everything was bound relative to that same initial context. In WebSphere Application Server V5, the default initial context for a server is its server root. System artifacts, such as EJB homes and resources, are bound to the server root of the server with which they are associated.
- **Transient and persistent partitions.**

  The name space is partitioned into transient areas and persistent areas. Server roots are transient. System-bound artifacts such as EJB homes and resources are bound under server roots. There is a cell persistent root, which you can use for cell-scoped persistent bindings, and a node persistent root, which you can use to bind objects with a node scope.
- **System name space structure.**

  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 means of a system name space. You can use the system name space structure to traverse to any context in the cell name space.
- **Configured bindings.**

  You can use the configuration graphical interface and script interfaces to configure bindings in various root contexts within the name space. These bindings are read-only and are bound by the system at server startup.
- **Support for CORBA Interoperable Naming Service (INS) object URLs.**

WebSphere Application Server V5 contains support for Common Object Request Broker Architecture (CORBA) object URLs (corbaloc and corbname) as Java Naming and Directory Interface (JNDI) provider URLs and lookup names.

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.

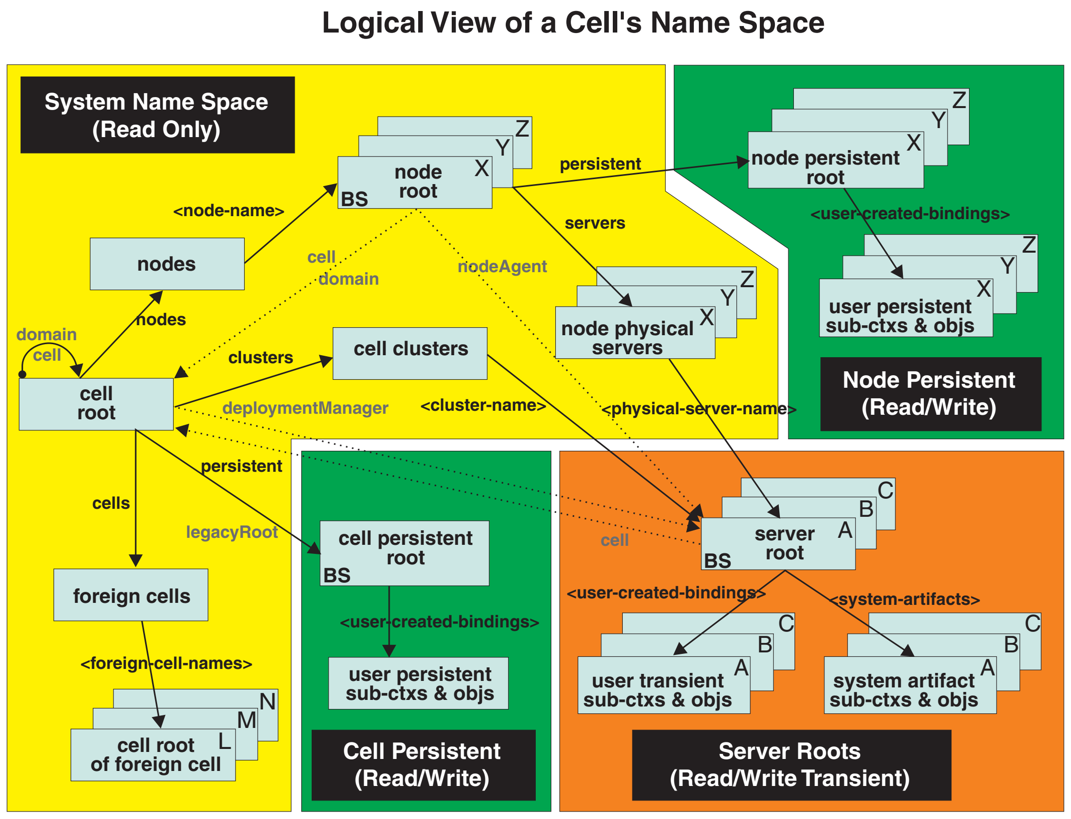


Figure 16. Name Space Logical View

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.

In WebSphere Application Server Network Deployment cells, the cell and node persistent areas can be read even if the deployment manager and respective node agent are not running. However, the deployment manager must be running to update the cell persistent segment, and a node agent must be running to update its respective node persistent segment.

Name space partitions

There are four major partitions in a cell name space:
- System name space partition
- Server roots partition
- Cell persistent partition
- Node persistent partition

System name space partition

The system name space contains a structure of contexts based on the cell topology. The system structure supports traversal to all parts of a cell name space and to the cell root of other cells, which are configured as foreign cells. The root of this structure is the cell root. In addition to the cell root, the system structure contains a node root for each node in the cell. You can access other contexts of interest specific to a node from the node root, such as the node persistent root and server roots for servers configured in that node.

All contexts in the system name space are read-only. You cannot add, update, or remove any bindings.

Server roots partition

Each server in a cell has a server root context. A server root is specific to a particular server. You can view the server roots for all servers in a cell as being in a transient read/write partition of the cell name space. System artifacts, such as EJB homes for server applications and resources, are bound under the server root context of the associated server. A server application can also add bindings under its server root. These bindings are transient. Therefore, the server application creates all required bindings at application startup, so they exist anytime the application is running.

A server cluster is composed of many servers that are logically equivalent. Each member of the cluster has its own server root. These server roots are not replicated across the cluster. In other words, adding a binding to the server root of one member does not propagate it to the server roots of the other cluster members. To maintain the same view across the cluster, you should create all user bindings under the server root by the server application at application startup so that the bindings are present under the server root of each cluster member. Because of Workload Management (WLM) behavior, a JNDI client outside a cluster has no control over which cluster member's server root context becomes the target of the JNDI operation. Therefore, you should execute bind operations to the server root of a cluster member from within that cluster member process only.

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.

The cell persistent area can be read even if the deployment manager is not running. However, the deployment manager must be running to update the cell persistent segment. Because every server contains its own copy of the cell persistent partition, any server can look up locally objects bound in the cell persistent partition.

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.

The node persistent area for a node can be read from any server in the node even if the respective node agent is not running. However, the node agent must be running to update the node persistent area, or for any server outside the node to read from that node persistent partition. Because every server in a node contains its own copy of the node persistent partition for its node, any server in the node can look up locally objects bound in that node persistent partition.

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.

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); ...
javax.naming.Context initialContext = new javax.naming.InitialContext(env);
```

2. Optional: Specify the name syntax

   Most WebSphere applications use JNDI to look up EJB objects and do not need to look up objects bound by CORBA applications. Therefore, the default name syntax used for JNDI names is the most convenient. If your application needs to look up objects bound by CORBA applications, you may need to change your name syntax so that all CORBA CosNaming names can be represented.

   JNDI clients can set the name syntax by setting a property. The property setting is applied by the initial context factory when you instantiate a new java.naming.InitialContext object. Names specified in JNDI operations on the initial context are parsed according to the specified name syntax.

   You can set the property:

   - From the command line by entering the actual string value. For example:

     ```
      java -Dcom.ibm.websphere.naming.name.syntax=ins
     ```
   - In a jndi.properties file by creating a file named jndi.properties as a text file with the desired properties settings. For example:

     ```
     ...
     com.ibm.websphere.naming.name.syntax=ins
     ...
     ```

     Include the file 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
     ...
     javax.naming.Context initialContext = new javax.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 -CCBootstrapHost -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 may 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 run-time 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 InitalContext 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 an 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)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

  Server cluster

  The example below shows a lookup of an EJB home which is running in the cluster, MyCluster. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
   // Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "cell/clusters/MyCluster/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

- **Fixed qualified names**

  If the target object has a cell-scoped fixed name defined for it, you can use its qualified form instead of the topology-based qualified name. Even though the topology-based name works, the fixed name does not change with the specific cell topology or with the movement of the target object to a different server. An example lookup with a qualified fixed name is shown below.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
```

```
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
     "cell/persistent/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
   }
catch (NamingException e) { // Error getting the home interface
...
}
```

JNDI lookup with a corbaname URL

A corbaname can be useful at times as a lookup name. If, for example, the target object is not a member of the federated name space and cannot be located with a qualifiied name, a corbaname can be a convenient way to look up the object. A lookup with a corbaname URL follows.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using a corbaname URL
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "corbaname:iiop:someHost:2809#com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

Example: Looking up a JavaMail session with JNDI

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 previous releases of WebSphere Application Server and with non-WebSphere Application Server JNDI clients. Also, the way resources from MQSeries must be bound to the name space has changed and is described below.

Interoperability with previous WebSphere Application Server Releases

- **EJB clients running on WebSphere Application Server V3.5 or V4.0 accessing EJB applications running on WebSphere Application Server V5**

  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 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.

  Note: EJB clients running in version 3.5 must be running in version 3.5.5 or above, or in version 3.5.3 or 3.5.4 with e-fix PQ51387 installed.

- **EJB clients running on WebSphere Application Server V5 accessing EJB applications running on WebSphere Application Server V3.5 or V4.0 servers**

The default initial context for a WebSphere Application Server v3.5 or 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 clients to access version 3.5.x and 4.0.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 servers

When an EJB application running in WebSphere Application Server V5 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 boostrapping to the cluster itself, but it will work. The prefix is needed, however, when looking up enterprise beans in other clusters. Name bindings under the **clusters** context are implemented on the name server to resolve to the server root of a running cluster member during a lookup; thus avoiding a single point of failure.
- **Without CORBA object URL support**

  If the JNDI initial context factory being used does not support CORBA object URLs, the initial context can be obtained from the server, and the lookup can be performed on the initial context as follows:

  ```
  Hashtable env = new Hashtable();
  env.put(CONTEXT.PROVIDER_URL, "iiop://myHost:2809");
  Context ic = new InitialContext(env);
  Object o = ic.lookup("cell/clusters/myCluster/myEJB");
  ```

Binding resources from MQSeries 5.2

In previous releases of WebSphere Application Server, the MQSeries jmsadmin tool could be used bind resources to the name space. When used with a WebSphere Application Server V5 name space, the resource will be bound within a transient partition in the name space and will 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 left panel on the console

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 max cache life for the cache is reached, or the max 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. 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.

Usually, cached objects are relatively static entities, and objects becoming stale are not a problem. However, you can set timeout values on cache entries or on a cache so that cache contents are periodically refreshed.

JNDI cache settings

Various cache property settings follow. Ensure that all property values are string values.

com.ibm.websphere.naming.jndicache.cachename

The name of the cache to associate with an initial context instance can be specified with this property. It is possible to create multiple InitialContext instances, each operating on the name space of a different name server. By default, objects from each bootstrap address are cached separately, since they each involve independent name spaces and name collisions could occur if they used the same cache. The provider URL specified when the initial context is created by default serves as the basis for the cache name. With this property, a JNDI client can specify a cache name. Valid options for cache names follow:

Valid options	Resulting cache behavior

providerURL (default)	Use the value for java.naming.provider.url property as the basis for the cache name. Cache names are based on the bootstrap host and port specified in the URL. The boostrap host is normalized to a fully qualfied name, if possible. For example, ″corbaname:iiop:server1:2809#some/starting/context″ and ″corbaloc:iiop://server1″ are normalized to the same cache name. If no provider URL is specified, a default cache name is used.
Any string	Use the specified string as the cache name. You can use any arbitrary string with a value other than ″providerURL″ as a cache name.

com.ibm.websphere.naming.jndicache.cacheobject

Turn caching on or off and clear an existing cache with this property.

By default, when an InitialContext is instantiated, it is associated with an existing cache or, if one does not exist, a new one is created. An existing cache is used with its existing contents. In some circumstances, this behavior is not desirable. For example, when objects that are looked up change frequently, they can become stale in the cache. Other options are available. The following table lists these other options along with the corresponding property value.

Valid values	Resulting cache behavior
populated (default)	Use a cache with the specified name. If the cache already exists, leave existing cache entries in the cache; otherwise, create a new cache.
cleared	Use a cache with the specified name. If the cache already exists, clear all cache entries from the cache; otherwise, create a new cache.
none	Do not cache. If this option is specified, the cache name is irrelevant. Therefore, this option will not disable a cache that is already associated with other InitialContext instances. The InitialContext that is instantiated is not associated with any cache.

com.ibm.websphere.naming.jndicache.maxcachelife

Impose a limit to the age of a cache with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to ″cleared″. This property enables a JNDI client to set the maximum life of a cache. This property differs from the maxentrylife property (below) in that the entire cache is cleared when the cache lifetime is reached. The table below lists the various maxcachelife values and their affect on cache behavior:

Valid options	Resulting cache behavior
0 (default)	Make the cache lifetime unlimited.
Positive integer	Set the maximum lifetime of the entire cache, in minutes, to the specified value. When the maximum lifetime for the cache is reached, the next attempt to read any entry from the cache causes the cache to be cleared

com.ibm.websphere.naming.jndicache.maxentrylife

Impose a limit to the age of individual cache entries with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to `cleared`. This property enables a JNDI client to set the maximum lifetime of individual cache entries. This property differs from the maxcachelife property in that individual entries are refreshed individually as their maximum lifetime reached. This might avoid any noticeable change in performance that might occur if the whole cache is cleared at once. The table below lists the various maxentrylife values and their effect on cache behavior:

Valid options	Resulting cache behavior
0 (default)	Lifetime of cache entries is unlimited.

Positive integer	Set the maximum lifetime of individual cache entries, in minutes, to the specified value. When the maximum lifetime for an entry is reached, the next attempt to read the entry from the cache causes the individual cache entry to refresh.

Example: Controlling JNDI cache behavior from a program

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-null `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 V5 clients to access Versions 3.5.x and 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.iiop.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.iiop.ORB");
props.put("com.ibm.CORBA.ORBInitRef.NameService",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameService");
props.put("com.ibm.CORBA.ORBInitRef.NameServiceServerRoot",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceServerRoot");
ORB _orb = ORB.init((String[])null, props);
...
```

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, please 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 unqualifed name resolves only if the initial context is from the name server in which the object is bound.

The following examples show CosNaming resolve operations using qualified topology-based lookup names and an unqualified lookup name.

CosNaming resolve operation using a qualified name

The topology-based qualified name for an object depends on whether the object is bound in a single server or a server cluster. Examples of each follow.

Single Server

The following example shows the lookup of an EJB home that is running in a single server. The enterprise bean that is being looked up is running in the server, `MyServer`, on the node, `Node1`.

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
  "cell/nodes/Node1/servers/MyServer/mycompany/accounting/AccountEJB");
accountHome =
  (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Server Cluster

The following example shows a lookup of an EJB home that is running in a cluster. The enterprise bean being that is looked up is running in the cluster, Cluster1. The name can be resolved if any of the cluster members is running.

Usage scenario

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
   "cell/clusters/Cluster1/mycompany/accounting/AccountEJB");
accountHome =
   (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

ORB string_to_object operation using an unqualified stringified name

If the resolve operation is being performed on the name server that contains the object, the system name space does not need to be traversed, and you can use an unqualified lookup name. Note that this name does not resolve on other name servers. If an unqualified name is provided, the object key must be NameServiceServerRoot so that the correct initial context is selected. If a qualified name is provided, you can use the default key of NameService.

The following example shows a lookup of an EJB home. The enterprise bean that is being looked up is bound on the name server running on the host `myHost` on port `2809`. Note the object key of `NameServiceServerRoot`.

```
// Assume orb is an existing ORB instance
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = orb.string_to_object(
    "corbaname:iiop:myHost:2809/NameServiceServerRoot#mycompany/accounting");
accountHome =
    (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

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.

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

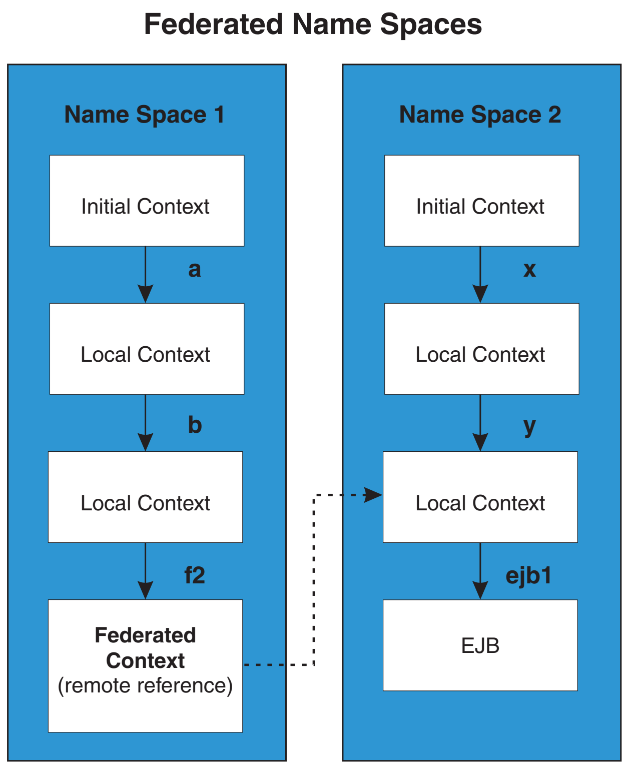
You can configure a binding of a string constant. The following data is required to configure a string constant binding:

- 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.
- 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 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

Configuring and viewing name space bindings

To view or configure an EJB, CORBA, Indirect lookup or string name space binding, complete the following:

1. Open the Administrative console.
2. Click **Environment**.
3. Click **Manage Name Space Bindings**.
4. 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**.
5. 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.
6. Edit the Binding identifier, the Name in name space, and the String value fields as desired.

   Note: All of these fields are required.
7. Click **Finish** to register the changes.

Configuring name servers

To configure a name server, complete the following:

1. Open the administrative console.
2. Click **Servers**.
3. Click **Application Servers**.
4. Click the application server you want to configure.
5. Click **Server Components**.
6. Click **Name Server**.
7. Edit the fields as desired.

   Note: All of these fields are mandatory.
8. To make other changes, click **Custom Properties**.
9. 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 one of the following paths:

- **Servers > Application Servers >***server_name* **> Server Components > Name Server**
- **Servers > JMS Servers >***server_name* **> 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

Troubleshooting name space problems

Many naming problems can be avoided by fully understanding the key underlying concepts of WebSphere Application Server naming.

1. Review the key concepts of WebSphere Application Server naming, especially Name space logical view and Lookup names support in deployment descriptors and thin clients.
2. Review the programming examples that are included in the sections explaining the JNDI and CosNaming interfaces.
3. Read Naming services component troubleshooting tips for additional general information.
4. If you Cannot look up an object hosted by WebSphere Application Server from a servlet, JSP file, or other client, read this article.

dumpNameSpace tool

You can use the dumpNameSpace tool to dump the contents of a name space accessed through a name server. When you invoke the dumpNameSpace tool, the naming service must be active. The dumpNameSpace tool cannot dump name spaces local to the server process, such as those with java:

and local: URL schemes. The local: name space contains references to enterprise beans with local interfaces. Use the name space dump utility for java:, local: and server name spaces to dump java: and local: name spaces.

Note that the server root context for the server at the specified host and port is dumped (unless a non-default starting context which precludes it is specified). The server root contexts for other servers are not dumped.

If you run the dumpNameSpace tool, a login prompt is displayed. If you cancel the login prompt, the dumpNameSpace tool continues outbound with an ″UNAUTHENTICATED″ credential. Thus, by default, an ″UNAUTHENTICATED″ credential is used that is equivalent to the ″Everyone″ access authorization policy. You can modify this default setting by changing the value for the `com.ibm.CSI.performClientAuthenticationRequired` property to `true` in the *`install_dir`*`/properties/sas.client.props` file. When you change this property to `true`, rerun the dumpNameSpace tool, and cancel the login prompt, the authorization fails and the command will not continue outbound.

Command line invocation descriptions of the dumpNameSpace tool follow. This section includes sample output.

You can also access this tool a through its program interface. Refer to the class com.ibm.websphere.naming.DumpNameSpace in the WebSphere Application Server API documentation. To invoke the tool through the command line, enter the following command from the *`WebSphere/AppServer/bin`* directory:

Platform	**Command**
UNIX	dumpNameSpace.sh [[-*keyword value*]...]
Windows NT	dumpNameSpace [[-*keyword value*]...]

Parameters

The keywords and associated values for the dumpNameSpace utility follow:

-host *myhost.austin.ibm.com*
Indicates the bootstrap host or the WebSphere Application Server host whose name space you want to dump. The value defaults to ***localhost***.

-port *nnn*
Indicates the bootstrap port which, if not specified, defaults to **2809**.

-root {cell | server | node | host | legacy | tree | default}
Indicates the root context to use as the initial context for the dump. The applicable root options and default root context depend on the type of name server from which the dump is being obtained. This information is provided in the following tables.

For WebSphere Application Servers V5 or later:

cell	DumpNameSpace default. Dump the tree starting at the cell root context.
server	Dump the tree starting at the server root context.
node	Dump the tree starting at the node root context. (Synonymous with host.)

For WebSphere Application Servers v4.0 or later:

legacy	DumpNameSpace default. Dump the tree starting at the legacy root context.
host	Dump the tree starting at the bootstrap host root context. (Synonymous with node.)

tree	Dump the tree starting at the tree root context.

For all WebSphere Application Servers and other name servers:

default	Dump the tree starting at the initial context which JNDI returns by default for that server type. This is the only -root choice that is compatible with WebSphere Application Servers prior to v4.0 and with non-WebSphere Application Server name servers.

-url *some provider URL*
Indicates the value for the java.naming.provider.url property used to get the initial JNDI context. This option can be used in place of the -host, -port, and -root options. If the -url option is specified, the -host, -port, and -root options are ignored.

-factory *com.ibm.websphere.naming.WsnInitialContextFactory*
Indicates the initial context factory to be used to get the JNDI initial context. The value defaults to: *com.ibm.websphere.naming.WsnInitialContextFactory* The default value generally does not need to be changed.

-startAt *some/subcontext/in/the/tree*
Indicates the path from the bootstrap host's root context to the top level context where the dump should begin. The utility recursively dumps subcontexts below this point. It defaults to an empty string, that is, the bootstrap host root context.

-format{jndi | ins}

Option	Description
jndi	The default. Displays name components as atomic strings.
ins	Shows name components parsed per INS rules (id.kind).

-report {short | long}

Option	Description
short	The default. Dumps the binding name and bound object type. This output is also provided by JNDI Context.list().
long	Dumps the binding name, bound object type, local object type, and string representation of the local object (that is, the IORs, string values, and other values that are printed). For objects of user-defined classes to display correctly with the long report option, it may be necessary to add their containing directories to the list of directories searched. Set the environment variable **WAS_USER_DIRS**. The value can include one or more directories, as for example: **Platform** **Command** **UNIX** `WAS_USER_DIRS=/usr/classdir1:/usr/classdir2 export WAS_USER_DIRS` **Windows NT** set `WAS_USER_DIRS=c:\classdir1;d:\classdir2` All zip, jar, and class files in the specified directories can then be resolved by the class loader when running **dumpNameSpace**.

-traceString *"some.package.name.to.trace.*=all=enabled"*
Represents the trace string with the same format as that generated by the servers. The output is sent to the file, `DumpNameSpaceTrace.out`.

Example: Invoking the name space dump utility

It is often helpful to view a dump of the name space to understand why a naming operation is failing. You can invoke the name space dump utility from the command line or from a program. Examples of each option follow.

Invoking name space dump utility from a command line

Invoke the name space dump utility from the command line by entering the following command:

```
dumpNameSpace -host myhost.mycompany.com -port 901
```

OR

```
dumpNameSpace -url corbaloc:iiop:myhost.mycompany.com:901
```

There are several command line options to choose from. For detailed help, enter the following command:

```
dumpNameSpace -help
```

Invoking name space dump utility from a Java program

You can dump name spaces from a program with the com.ibm.websphere.naming.DumpNameSpace API. Refer to the WebSphere Application Server API documentation for details on the DumpNameSpace program interface

The following example illustrates how to invoke the name space dump utility from a Java program:

```
{
   ...
   import javax.naming.Context;
   import javax.naming.InitialContext;
   import com.ibm.websphere.naming.DumpNameSpace;
   ...
   java.io.PrintStream filePrintStream = ...
   Context ctx = new InitialContext();
   // Starting context for dump
   ctx = (Context) ctx.lookup("cell/nodes/node1/servers/server1");
   DumpNameSpace dumpUtil =
      new DumpNameSpace(filePrintStream, DumpNameSpace.SHORT);
   dumpUtil.generateDump(ctx);
   ...
}
```

Name space dump utility for `java:`, `local:` and `server` name spaces

Sometimes it is helpful to dump the java: name space for a J2EE application. You cannot use the dumpNameSpace command line utility for this purpose because the application's java: name space is accessible only by that J2EE application. From the WebSphere Application Server scripting tool, you can invoke a NameServer MBean to dump the java: name space for any J2EE application running in that same server process.

There is another name space local to server process which you cannot dump with the dumpNameSpace command line utility. This name space has the URL scheme of local: and is used by the container to bind objects locally instead of through the name server. The local: name space contains references to enterprise beans with local interfaces. There is only one local: name space in a server process. You can dump the local: name space by invoking the NameServer MBean associated with that server process.

Name space dump options

Name space dump options are specified in the MBean invocation as a parameter in chararacter string format. The option descriptions follow.

-startAt *some/subcontext/in/the/tree*

Indicates the path from the name space root context to the top level context where the dump should begin. The utility recursively dumps subcontexts below this point. It defaults to an empty string, that is, the root context.

-report {short | long}

Option	Description
short	The default. Dumps the binding name and bound object type. This output is also provided by JNDI Context.list().
long	Dumps the binding name, bound object type, local object type, and string representation of the local object (that is, the IORs, string values, and other values that are printed).

-root {tree | host | legacy | cell | node | server | default}

Specify the root context of where the dump should start. The default value for `-root` is *cell*. This option is only valid for server name space dumps.

Option	Description
tree	Dump the tree starting at the tree root context.
host	Dump the tree starting at the server host root context (synonymous with ″node″).
legacy	Dump the tree starting at the legacy root context.
cell	Dump the tree starting at the cell root context. This is the default option.
node	Dump the tree starting at the node root context (synonymous with ″host″).
server	Dump the tree starting at the server root context. This is `-root` default.
default	Dump the tree starting at the initial context which JNDI returns by default for that server type.

-format {jndi | ins}

Specify the format to display name component as atomic strings or parsed according to INS rules (id.kind). This option is only valid for server name space dumps.

Option	Description
jndi	Display name components as atomic strings. This is `-format` default.
ins	Display name components parsed according to INS rules (id.kind).

NameServer MBean invocation

1. Enter the WebSphere Application Server scripting command prompt.

   Invoke a method on a NameServer MBean by using the WebSphere Application Server scripting tool. Enter the scripting command prompt by typing the following command:

Platform	Command
UNIX	wsadmin.sh
Windows NT	wsadmin

   Use the -help option for help on using the wsadmin command.
2. Select the NameServer MBean instance to invoke.

   Execute the following script commands to select the NameServer instance you want to invoke. For example,

```
set mbean [$AdminControl completeObjectName WebSphere:*,type=NameServer,cell=
    cellName,node=nodeName,process=serverName]
```

where *cellName*, *nodeName*, and *serverName* are the names of the cell, node, and server for the MBean you want to invoke. The specified server must be running before you can invoke a method on the MBean.

You can see a list of all NameServer MBeans current running by issuing the following query:

```
$AdminControl queryNames {*:*,type=NameServer}
```

3. Invoke the NameServer MBean.

java: name space
Dump a java: name space by invoking the dumpJavaNameSpace method on the NameServer MBean. Since each server application has its own java: name space, the application must be specified on the method invocation. An application is identified by the application name, module name, and component name. The method syntax follows:

```
$AdminControl invoke $mbean dumpJavaNameSpace {{appname}{modName}{compName}{opts}}
```

where *appName* is the application name, *modName* is the module name, and *compName* is the component name of the java: name space you want to dump. The value for *opts* is the list of name space dump options described earlier in this section. The list can be empty.

local: name space
Dump a java: name space by invoking the dumpLocalNameSpace method on the NameServer MBean. Since there is only one local: name space in a server process, you have to specify the name space dump options only.

```
$AdminControl invoke $mbean dumpLocalNameSpace {{opts}}
```

where *opts* is the list of name space dump options described earlier in this section. The list can be empty.

Server name space
Dump a server name space by invoking the dumpServerNameSpace method on an application server's NameServer MBean. This provides an alternative way to dump the name space on an application server, much like the dumpNameSpace command line utility.

```
$AdminControl invoke $mbean dumpServerNameSpace {{opts}}
```

where *opts* is the list of name space dump options described earlier in this section. The list can be empty.

Name space dump output

Name space dump output is sent to the console. It is also written to the file `DumpNameSpace.log`, in the server's log directory.

Example: Invoking the name space dump utility for `java:` and `local:` name spaces

It is often helpful to view the dump of a `java:` or `local:` name space to understand why a naming operation is failing. The NameServer MBean running in the application's server process can be invoked from the WebSphere Application Server scripting tool to generate a dump of these name spaces. Examples of NameServer MBean calls to generate dumps of java: and local: name spaces follow.

Dumping a `java:` name space

Assume you want to dump the java: name space of an application component running in server server1 on node node1 of the cell MyCell. The application name is AcctApp in module AcctApp.war, and the component name is Acct Servlet. The following script commands generate a long format dump of the application's java: name space of that application:

```
set mbean [$AdminControl completeObjectName WebSphere:*,type=NameServer,cell=MyCell,node=node1,process=server1]
$AdminControl invoke $mbean dumpJavaNameSpace {{AcctApp}{AcctApp.war}{Acct Servlet}{-report long}}
```

Dumping a `local:` name space

Assume you want to dump the local: name space for the server server1 on node node1 of cell MyCell. The following script commands will generate a short format dump of that server's local name space:

```
set mbean [$AdminControl completeObjectName WebSphere:*type=NameServer,cell=MyCell,node=node1,process=server1]
$AdminControl invoke $mbean dumpLocalNameSpace {{-report short}}
```

Name space dump sample output

Name space dump output looks like the following example, which is the **SHORT** dump format:

```
Getting the initial context
Getting the starting context

==============================================================================
Name Space Dump
   Provider URL: corbaloc:iiop:localhost:9810
   Context factory: com.ibm.websphere.naming.WsnInitialContextFactory
   Requested root context: cell
   Starting context: (top)=outpostNetwork
   Formatting rules: jndi
   Time of dump: Mon Sep 16 18:35:03 CDT 2002
==============================================================================



==============================================================================
Beginning of Name Space Dump
==============================================================================

    1 (top)
    2 (top)/domain                                      javax.naming.Context
    2    Linked to context: outpostNetwork
    3 (top)/cells                                       javax.naming.Context
    4 (top)/clusters                                    javax.naming.Context
    5 (top)/clusters/Cluster1                           javax.naming.Context
    6 (top)/cellname                                    java.lang.String
    7 (top)/cell                                        javax.naming.Context
    7    Linked to context: outpostNetwork
    8 (top)/deploymentManager                           javax.naming.Context
    8    Linked to URL: corbaloc::outpost:9809/NameServiceServerRoot
    9 (top)/nodes                                       javax.naming.Context
   10 (top)/nodes/will2                                 javax.naming.Context
   11 (top)/nodes/will2/persistent                      javax.naming.Context
   12 (top)/nodes/will2/persistent/SomeObject           SomeClass
   13 (top)/nodes/will2/nodename                        java.lang.String
   14 (top)/nodes/will2/domain                          javax.naming.Context
   14    Linked to context: outpostNetwork
   15 (top)/nodes/will2/cell                            javax.naming.Context
   15    Linked to context: outpostNetwork
   16 (top)/nodes/will2/servers                         javax.naming.Context
   17 (top)/nodes/will2/servers/server1                 javax.naming.Context
   18 (top)/nodes/will2/servers/will2                   javax.naming.Context
   19 (top)/nodes/will2/servers/member2                 javax.naming.Context
   20 (top)/nodes/will2/node                            javax.naming.Context
   20    Linked to context: outpostNetwork/nodes/will2
   21 (top)/nodes/will2/nodeAgent                       javax.naming.Context
```

```
22 (top)/nodes/outpost                                javax.naming.Context
23 (top)/nodes/outpost/node                           javax.naming.Context
23    Linked to context: outpostNetwork/nodes/outpost
24 (top)/nodes/outpost/nodeAgent                      javax.naming.Context
24    Linked to URL: corbaloc::outpost:2809/NameServiceServerRoot
25 (top)/nodes/outpost/persistent                     javax.naming.Context
26 (top)/nodes/outpost/nodename                       java.lang.String
27 (top)/nodes/outpost/domain                         javax.naming.Context
27    Linked to context: outpostNetwork
28 (top)/nodes/outpost/servers                        javax.naming.Context
29 (top)/nodes/outpost/servers/server1                javax.naming.Context
30 (top)/nodes/outpost/servers/server1/url            javax.naming.Context
31 (top)/nodes/outpost/servers/server1/url/CatalogDAOSQLURL
31                                                    java.net.URL
32 (top)/nodes/outpost/servers/server1/mail           javax.naming.Context
33 (top)/nodes/outpost/servers/server1/mail/PlantsByWebSphere
33                                                    javax.mail.Session
34 (top)/nodes/outpost/servers/server1/TransactionFactory
34                                                    com.ibm.ejs.jts.jts.ControlSet$LocalFactory
35 (top)/nodes/outpost/servers/server1/servername     java.lang.String
36 (top)/nodes/outpost/servers/server1/WSsamples      javax.naming.Context
37 (top)/nodes/outpost/servers/server1/WSsamples/TechSampDatasource
37                                                    TechSamp
38 (top)/nodes/outpost/servers/server1/thisNode       javax.naming.Context
38    Linked to context: outpostNetwork/nodes/outpost
39 (top)/nodes/outpost/servers/server1/cell           javax.naming.Context
39    Linked to context: outpostNetwork
40 (top)/nodes/outpost/servers/server1/eis            javax.naming.Context
41 (top)/nodes/outpost/servers/server1/eis/DefaultDatasource_CMP
41                                                    Default_CF
42 (top)/nodes/outpost/servers/server1/eis/WSsamples  javax.naming.Context
43 (top)/nodes/outpost/servers/server1/eis/WSsamples/TechSampDatasource_CMP
43                                                    TechSamp_CF
44 (top)/nodes/outpost/servers/server1/eis/jdbc       javax.naming.Context
45 (top)/nodes/outpost/servers/server1/eis/jdbc/PlantsByWebSphereDataSource_CMP
45                                                    PLANTSDB_CF
46 (top)/nodes/outpost/servers/server1/eis/jdbc/petstore
46                                                    javax.naming.Context
47 (top)/nodes/outpost/servers/server1/eis/jdbc/petstore/PetStoreDB_CMP
47                                                    PetStore_CF
48 (top)/nodes/outpost/servers/server1/eis/jdbc/CatalogDB_CMP
48                                                    Catalog_CF
49 (top)/nodes/outpost/servers/server1/jta            javax.naming.Context
50 (top)/nodes/outpost/servers/server1/jta/usertransaction
50                                                    java.lang.Object
51 (top)/nodes/outpost/servers/server1/DefaultDatasource
51                                                    Default Datasource
52 (top)/nodes/outpost/servers/server1/jdbc           javax.naming.Context
53 (top)/nodes/outpost/servers/server1/jdbc/CatalogDB CatalogDB
54 (top)/nodes/outpost/servers/server1/jdbc/petstore  javax.naming.Context
55 (top)/nodes/outpost/servers/server1/jdbc/petstore/PetStoreDB
55                                                    PetStoreDB
56 (top)/nodes/outpost/servers/server1/jdbc/PlantsByWebSphereDataSource
56                                                    PLANTSDB
57 (top)/nodes/outpost/servers/outpost                javax.naming.Context
57    Linked to URL: corbaloc::outpost:2809/NameServiceServerRoot
58 (top)/nodes/outpost/servers/member1                javax.naming.Context
59 (top)/nodes/outpost/cell                           javax.naming.Context
59    Linked to context: outpostNetwork
60 (top)/nodes/outpostManager                         javax.naming.Context
61 (top)/nodes/outpostManager/domain                  javax.naming.Context
61    Linked to context: outpostNetwork
62 (top)/nodes/outpostManager/cell                    javax.naming.Context
62    Linked to context: outpostNetwork
63 (top)/nodes/outpostManager/servers                 javax.naming.Context
64 (top)/nodes/outpostManager/servers/dmgr            javax.naming.Context
64    Linked to URL: corbaloc::outpost:9809/NameServiceServerRoot
```

```
   65 (top)/nodes/outpostManager/node                    javax.naming.Context
   65    Linked to context: outpostNetwork/nodes/outpostManager
   66 (top)/nodes/outpostManager/nodename                java.lang.String
   67 (top)/persistent                                   javax.naming.Context
   68 (top)/persistent/cell                              javax.naming.Context
   68    Linked to context: outpostNetwork
   69 (top)/legacyRoot                                   javax.naming.Context
   69    Linked to context: outpostNetwork/persistent
   70 (top)/persistent/AnotherObject                     AnotherClass

==============================================================================
End of Name Space Dump
==============================================================================
```

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.

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 instructions and examples
- Naming in WebSphere Application Server V5: Impact on Migration and Interoperability

Programming specifications
- Java Naming and Directory Interface^{TM} 1.2.1 Specification
- OMG CosNaming Interoperable Naming Specification

Chapter 14. Using the dynamic cache service to improve performance

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, or serves the content of the object from the dynamic cache.

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.

The dynamic caching documentation provides you with the following tasks to enable and configure the dynamic cache service, as well as advanced features, such as controlling external caches and building user-defined drop-in components to customize the cache operation.

1. Enable the dynamic cache service globally.
2. Configure servlet caching.
3. Configure Edge Side Include (ESI) caching.
4. Configure command caching.
5. Configure Web services caching.
6. Troubleshoot any problems with the dynamic cache service.

Dynamic cache

Caching the output of servlets, commands and JavaServer Pages (JSP) files, 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, which 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, results of servlet execution, and metadata.

Enabling the dynamic cache service

In order to use the dynamic cache service, you must first enable it.

1. Open the administrative console.
2. Click **Servers** > **Application Servers** in the administrative console navigation tree.
3. Click a server.
4. Click **Dynamic Cache Service** under Additional Properties.
5. Select **Enable service at server startup** in the **Startup state** field.
6. Click **Apply** or **OK**.
7. Restart WebSphere Application Server.

The dynamic cache service will now cache content for requests that have cache policies configured.

Dynamic cache service settings

Use this page to configure and manage the dynamic cache service settings.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service**.

© Copyright IBM Corp. 2003

Startup state
Specifies whether the dynamic cache is enabled.

Cache size
Specifies a positive integer as the value for the maximum number of entries the cache holds.
Enter the cache size value in this field between the range of 100 through 200,000.

Default priority
Specifies the default priority for cache entries, determining how long an entry stays in a full cache.

Default	1
Range	1 to 255

Disk offload
Specifies whether disk offload is enabled.
By default, the dynamic cache maintains the number of entries configured in memory. If new entries are created while the cache is full, the priorities configured for each cache entry and a least recently used algorithm, are used to remove entries from the cache. In addition to having a cache entry removed from memory when the cache is full, you can enable disk offload to have a cache entry copied to the file system (the location is configurable). Later, if that cache entry is needed, it is moved back to memory from the file system.

Cache replication
Specifies whether cache replication is enabled.
You can also configure advanced cache replication settings.

Configuring servlet caching

To enable servlet caching, you must enable the dynamic cache service.

1. Open the administrative console.
2. Click **Servers** > **Application Servers** in the console navigation tree.
3. Click a server.
4. Click **Web Container**.
5. Select the **Enable servlet caching** check box under the Configuration tab.
6. Click **Apply** or **OK**.

Servlet caching
After a servlet is invoked and generating the output to cache, a cache entry is created containing the output and the side effects of the invocation. For example, these side effects can include calls to other servlets or JavaServer Pages (JSP) files, as well as metadata about the entry, including timeout and entry priority information.
Unique entries are distinguished by an ID string generated from the HttpServletRequest object for each invocation of the servlet. You can then base servlet caching on:
- Request parameters and attributes the URI used to invoke the servlet
- Session information
- Other options, including cookies

Since JSP files are compiled by WebSphere Application Server into servlets, the dynamic cache function treats them the same, except in specifically documented situations.

Configuring the dynamic cache disk offload

By default, when the number of cache entries reaches the configured limit for a given WebSphere server, eviction of cache entries occurs, allowing new entries to enter the cache service. The dynamic cache includes an alternative feature named disk offload, that copies the evicted cache entries to disk for potential future access.

To configure disk offload:
1. Open the administrative server.
2. Click **Server** > **Application Server** in the administrative console navigation tree.
3. Click *server*.
4. Click **Dynamic Cache Service**.
5. Click the **Enable disk offload** check box in the **Disk offload** field. You can also set the disk offload location in this field.
6. Click **Apply** or **OK**.

Application servers must have different disk offload locations

When you have two or more application servers with servlet caching enabled and the application servers specify the same disk offload location for their caches through the dynamic cache service, the following exceptions might occur:

```
java.lang.NullPointerException
       at com.ibm.ws.cache.CacheOnDisk.readTemplate(CacheOnDisk.java:686)
       at com.ibm.ws.cache.Cache.internalInvalidateByTemplate(Cache.java:828)
```

or:

```
java.lang.NullPointerException
       at com.ibm.ws.cache.CacheOnDisk.readCacheEntry(CacheOnDisk.java:600)
       at com.ibm.ws.cache.Cache.getCacheEntry(Cache.java:341)
```

If one server is run as root and the other servers are run as nonroot, this problem could occur. For example, if `server1` runs as root and `server2` runs as `wasuser` or `wasgroup`, the cache files in the disk offload location might be created with root permissions. This situation causes the applications running on the nonroot servers to crash when they try to read or write to the cache.

The disk offload location must be unique for servers defined on the same node. If you have multiple servers defined on the same node, make sure the disk offload location is different for each server as defined on the **Dynamic Cache Service** panel, **Offload location** field.

Configuring Edge Side Include caching

Edge Side Include (ESI) is configured through the `plugin-cfg.xml` file.

The Web server plug-in contains a built-in ESI processor. The ESI processor has the ability to cache whole pages, as well as fragments, providing a higher cache hit ratio. The cache implemented by the ESI processor is an in-memory cache, not a disk cache, therefore, the cache entries are not saved when the Web server is restarted.

The basic operation of the ESI processor is as follows: When a request is received by the Web server plug-in, it is sent to the ESI processor, unless the ESI processor is disabled. It is enabled by default. If a cache miss occurs, a Surrogate-Capabilities header is added to the request and the request is forwarded to the WebSphere Application Server. If the dynamic servlet cache is enabled in the application server, and the response is edge cacheable, the application server returns a Surrogate-Control header in response to the WebSphere Application Server plug-in.

The value of the Surrogate-Control response header contains the list of rules which are used by the ESI processor in order to generate the cache ID. The response is then stored in the ESI cache, using the cache ID as the key. For each ESI include tag in the body of the response, a new request is processed such that each nested include results in either a cache hit or another request forwarded to the application server. When all nested includes have been processed, the page is assembled and returned to the client.

The ESI processor is configurable through the WebSphere Web server plug-in configuration file `plugin-cfg.xml`. The following is an example of the beginning of this file, which illustrates the ESI configuration options.

```
<?xml version-"1.0"?>
<Config>
  <Property Name="esiEnable" Value="true"/>
  <Property Name="esiMaxCacheSize" Value="1024"/>
  <Property Name="esiInvalidationMonitor" Value="false"/>
```

The first option, esiEnable, can be used to disable the ESI processor by setting the value to false. ESI is enabled by default. If ESI is disabled, then the other ESI options are ignored.

The second option, esiMaxCacheSize, is the maximum size of the cache in 1K byte units. The default maximum size of the cache is 1 megabyte. If the cache is full, the first entry to be evicted from the cache is the entry that is closest to expiration.

The third option, esiInvalidationMonitor, specifies whether or not the ESI processor should receive invalidations from the application server. ESI works well when the Web servers following a threading model is used, and only one process is started. When multiple processes are started, each process caches the responses independently and the cache is not shared. This could lead to a situation where, the system’s memory is fully used up by ESI processor.There are three methods by which entries are removed from the ESI cache: first, an entry’s expiration timeout could fire; second, an entry may be purged to make room for newer entries; or third, the application server could send an explicit invalidation for a group of entries. In order for the third mechanism to be enabled, the esiInvalidationMonitor property must be set to true and the DynaCacheEsi application must be installed on the application server. The DynaCacheEsi application is located in the installableApps directory and is named DynaCacheEsi.ear. If the ESIInvalidationMonitor property is set to true but the DynaCacheEsi application is not installed, then errors will occur in the webserver plugin and the request will fail.

The ESI processor’s cache can be monitored through the CacheMonitor application. In order for ESI processor’s cache to be visible in the CacheMonitor, the DynaCacheEsi application must be installed as described above and the ESIInvalidationMonitor property must be set to true in the `plugin-cfg.xml` file.

When WebSphere Application Server is used to serve static data, such as images and HTML on the application server, the URLs are also cached in the ESI processor. This data has a default timeout of 300 seconds. You can change the timeout value by adding the property com.ibm.servlet.file.esi.timeOut to your JVM’s command line parameters. The following example shows how to set a one minute timeout on static data cached in the plug-in:

```
-Dcom.ibm.servlet.file.esi.timeOut=60
```

For more information about the `plugin-cfg.xml` file see ″Chapter 14, “Using the dynamic cache service to improve performance,” on page 597.″

For information about configuring alternate URL, see ″Configuring alternate URL.″

Configuring alternate URL

Alternate URL is a method for edge caching JavaServer Pages (JSP) files and servlet responses that you can not request externally. Dynamic cache provides support to recognize the presence of an Edge Side Include (ESI) processor and to generate ESI include tags and appropriate cache policies for edge cacheable fragments. However, for a fragment to be edge cacheable, you must be able to externally request it from the application server. In other words, if a user types the URL in their browser with the appropriate parameters and cookies for the fragment, WebSphere Application Server must return the content for that fragment.

One of the standard J2EE programming architectures is the model-view-controller (MVC) architecture, where a call to a controller servlet might include one or more child JSP files to construct the view. When using the MVC programming model, the child JSP files are edge cacheable only if you can request these JSP files externally, which is not usually the case. For example, if a child JSP file uses one or more request attributes that are determined and set by the controller servlet, you cannot cache that JSP file on the edge. You can use alternate URL support to overcome this limitation by providing an alternate controller servlet URL used to invoke the JSP file.

The alternate URL for a JSP file or a servlet is set in the `cachespec.xml` file as a property with the name `alternate_url`. You can set the alternate URL either on a per cache-entry basis or on a per cache-id basis. It is valid only if the `EdgeCacheable` property is also set for that entry. If the `EdgeCacheable` property is not set, the `alternate_url` property is ignored. The following is a sample cache policy using the `alternate_url` property:

```
<cache-entry>
    <class>servlet</class>
    <name>/AltUrlTest2.jsp</name>
    <property  name="EdgeCacheable">true</property>
    <property  name="alternate_url">/alturlcontroller2</property>
        <cache-id>
            <timeout>600</timeout>
            <priority>2</priority>
        </cache-id>
</cache-entry>
```

For more information on the `cachespec.xml` file, see Cachespec.xml file.

Configuring external cache groups

The dynamic cache can control caches outside of the application server, such as IBM Edge Server, an IBM HTTP Server for distributed platforms' Fast Response Cache Accelerator (FRCA) cache, and a WebSphere HTTP Server for distributed platforms plug-in ESI Fragment Processor. When external cache groups are defined, the dynamic cache matches externally cacheable cache entries with those groups, and pushes cache entries and invalidations out to those groups. This allows WebSphere Application Server to manage dynamic content beyond the application server. The content can then be served from the external cache, instead of the application server, improving savings in performance.

1. Open the administrative console.
2. Enable the dynamic cache.
   a. Click **Servers** > **Application Servers** in the administrative console navigation tree.
   b. Click a *server*.
   c. Click **Dynamic Cache Service**.
   d. Select the check box in the **Startup state** field to enable the dynamic cache.
3. Define the external cache group in which WebSphere Application Server should control.
   a. Click **External Caching Groups** from the Dynamic Cache administrative console page.
   b. Click **New** or choose an external cache group from the list.
4. Configure cache group members.
   a. Click **External cache groups** from the Dynamic Cache administrative console page. Then click **New** or choose an external cache group from the list.

b. Click **External cache group members** > **New** or choose an external cache group member from the list.
c. Type the configuration string in the **Address** field.
d. Type the adapter bean name in the **Adapter Bean Name** field.
e. **Save** the configuration.
f. Click **Apply** or **OK**.

External cache group collection

Use this page to define sets of external caches controlled by WebSphere Application Server on Web servers, such as IBM Edge Server and IBM HTTP Server.
To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service** > **External Cache Groups**.

Name:

Specifies the external cache group name.

The external cache group name needs to match the externalcache property as defined in the servlet or JSP `cachespec.xml` file.

When external caching is enabled, the cache matches pages with its URIs and pushes matching pages to the external cache. The entries can then be served from the external cache, instead of the application server.

Type:

Specifies the external cache group type.

External cache group settings

Use this page to configure sets of external caches controlled by WebSphere Application Server on Web servers, such as IBM Edge Server and IBM HTTP Server.
To view this administrative console page, click **Servers** > **Application Server** > *server* > **Dynamic Cache Service** > **External Cache groups** > *external_cache_group*.

Name:

Specifies the external cache group name.

The external cache group name needs to match the externalcache property as defined in the servlet or JavaServer Pages (JSP) `cachespec.xml` file.

When external caching is enabled, the cache matches pages with its URIs and pushes matching pages to the external cache. The entries can then be served from the external cache, instead of the application server. This ability creates a significant savings in performance.

Type:

Specifies the external cache group type.

External cache group member collection

Use this page to define specific caches that are members of a cache group.
To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service** > **External Cache groups** > *external_cache_group* > **External cache group members**.

Address:

Specifies a configuration string used by external cache adapter bean to connect to the external cache.

AdapterBeanName:

Specifies the adapter bean name.

Example adapter bean names supported in WebSphere Application Server are:

AFPA
AdapterBeanName: com.ibm.ws.cache.servlet.Afpa
Address: Port on which afpa listens
ESI
AdapterBeanName: com.ibm.websphere.servlet.cache.ESIInvalidatorServlet
Address: local host
WTE
AdapterBeanName: com.ibm.websphere.edge.dynacache.WteAdapter
Address: hostname:port (host name and port on which WTE is listening)

External cache group member settings

Use this page to configure specific caches that are members of a cache group.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service** > **External Cache groups** > *external_cache_group* > **External cache group members** > *external_cache_group_member*.

Address:

Specifies a configuration string used by external cache adapter bean to connect to the external cache.

AdapterBeanName:

Specifies the adapter bean name.

Example adapter bean names supported in WebSphere Application Server are:

AFPA
AdapterBeanName: com.ibm.ws.cache.servlet.Afpa
Address: Port on which afpa listens
ESI
AdapterBeanName: com.ibm.websphere.servlet.cache.ESIInvalidatorServlet
Address: local host
WTE
AdapterBeanName: com.ibm.websphere.edge.dynacache.WteAdapter
Address: hostname:port (host name and port on which WTE is listening)

Configuring high-speed external caching through the Web server

IBM HTTP Server for Windows NT and Windows 2000 operating systems contains a high-speed cache referred to as the *Fast Response Cache Accelerator*, or *cache accelerator*.

The Fast Response Cache Accelerator is available on Windows NT and Windows 2000 operating systems and AIX platforms. However, support to cache dynamic content is only available on Windows NT and Windows 2000 operating systems.

You can enable cache accelerator to cache static and dynamic content. To enable cache accelerator for caching static content, add the following directives to the `http.conf` configuration file, in the IBM HTTP Server `conf` directory:
- `AfpaEnable`
- `AfpaCache on`
- `AfpaLogFile "`*`install_root`*`\IBMHttpServer\logs\afpalog" V-ECLF`

To enable cache accelerator for caching dynamic content, such as servlets and Java Server Pages (JSP) files, configure the WebSphere Application Server and the IBM HTTP Server for distributed platforms:
1. Configure WebSphere Application Server to enable Fast Response Cache Accelerator:
   a. Turn on servlet caching for each application server that uses the cache accelerator.
   b. Configure an external cache group on the application server:
      - Click **Servers > Application Servers > server1**.
      - Click **Dynamic Cache Service** in the **Additional Properties** window.
      - Click **External Cache Groups** in the **Additional Properties** window.
      - Click **New** on the External cache group administrative console page to define an external cache group named `afpa` for each application server that uses the cache accelerator.
      - Type `afpa` in the External cache group field.
      - Click **Apply**.
      - Add a member to the group with an adapter bean name of `com.ibm.ws.cache.servlet.Afpa`:

      Click **Afpa > External cache group members**. Click **New** on the External cache group members administrative console page. Type `com.ibm.ws.cache.servlet.Afpa` in the AdapterBean name field. Enter an unused port number in the Address field.
   c. Add a cache policy in the `cachespec.xml` file for the servlet or JSP file you want to cache. Add the following property to the cache policy:
      ```
      <property name="ExternalCache">afpa</property>
      ```

   It is important to follow all the steps for every application server in the cluster.
2. Enable cache accelerator on the IBM HTTP Server for distributed platforms:
   a. Add the following directives to the end of the `httpd.conf` file:
      - `AfpaEnable`
      - `AfpaCache on`
      - `AfpaLogFile "`*`install_root`*`\IBMHttpServer\logs\afpalog" V-ECLF`
      - `LoadModule afpaplugin_module` *`install_root`*`/bin/afpaplugin.dll`
      - `AfpaPluginHost` *`WAS_Hostname:port`*, where *`WAS_Hostname`* is the host name of the application server and *`port`* is the port you specified in the Address field while configuring the external cache group member

   The LoadModule directive loads the IBM HTTP Server plug-in that connects the Fast Response Cache Accelerator to the WebSphere Application Server fragment cache. If multiple IBM HTTP Servers are routing requests to a single application server, add the directives above to the `http.conf` file of each of these IBM HTTP Servers for distributed platforms. If one IBM HTTP Server is routing requests to a cluster of application servers, add the `AfpaPluginHost WAS_Hostname:port` directive to the `http.conf` file for each application server in the cluster. For example, if there are three application servers in the cluster, add the following directives to the `http.conf` file:
   - `LoadModule afpaplugin_module` *`install_root`*`/bin/afpaplugin.dll`
   - `AfpaPluginHost WAS1_Hostname:port1`
   - `AfpaPluginHost WAS2_Hostname:port2`
   - `AfpaPluginHost WAS3_Hostname:port3`

Configuring Fast Response Cache Accelerator cache size through a distributed platforms Web server:

In the default IBM HTTP Server for distributed platforms configuration, the maximum Fast Cache Accelerator dynamic cache size is calculated as 1/8 of physical pin-able memory. On a machine with 384 megabytes of RAM, it allows a maximum of approximately 50 megabytes for the Fast Cache Accelerator dynamic cache. When this limit is reached, the Cache Accelerator then deletes older entries to cache new ones.

Follow these steps to configure the Cache Accelerator:
Using the IBM HTTP Server for distributed platforms' AfpaDynaCacheMax directive, tune the maximum allowed cache size:

1. Place the directive in the global server configuration scope, along with the other default Fast Cache Accelerator directives.
2. Enable Fast Cache Accelerator. To enable the Fast Cache Accelerator, update the following directives in this IBM HTTP Server's `http.conf` file:

   ```
   AfpaEnable
   AfpaCache on
   AfpaLogFile "c:/Program Files/IBM HTTP Server/logs/afpalog" V-ECLF
   AfpaDynaCacheMax 10
   ```

   These above settings limit the dynamic cache size to 10 megabytes. If you use these directives to increase cache size, do not make the cache so large that all the physical memory is consumed. Determine how much memory is available when all applications are running, by using the Windows Task Manager.

   Assign no more than 50% of available physical memory to the dynamic cache. Specifying too large a cache not only decreases the performance of other applications, but also puts you at a risk for completely running out of memory.

   The default configuration does not include the AfpaDynaCacheMax directive where the cache size is automatically calculated as 1/8 of physical memory.

Displaying cache information

The dynamic cache monitor is an installable Web application that displays simple cache statistics, cache entries, and cache policy information.

1. Use the administrative console to install the cache monitor application from the `*install_root*/installableApps` directory. The application is named `CacheMonitor.ear`. Install the cache monitor onto the application server you are trying to monitor. Installing the cache monitor on the *admin_host* (port 909*x*) is more secure than installing it on the *default_host* (908*x*), and so it is preferable to install it onto the *admin_host*.
2. Access the Web application using a Web browser and the URL `http://`*`your host_name`*`:`*`your port_number`*`/cachemonitor`, where *your port_number* is the port associated with the host on which you installed the cache monitor application.
3. Verify that the cache monitor is working properly.
   a. View the Statistics page and verify the cache configuration and cache data. Click **Reset Statistics** to reset the counters
   b. View the Cache Policies page to see which cache policies are currently loaded in the dynamic cache. Click on a template to view the cache ID rules for the template.
   c. View the Cache Contents page to examine the contents currently cached in memory.

   d. View the ESI Statistics page to view data about the current ESI processors configured for caching. Click **Refresh Statistics** to see the latest statistics or content from the ESI processors. Click **Reset Statistics** to reset the counters.
   e. View the Disk Offload page to view content currently off-loaded from memory to disk.

   When viewing contents on memory or disk, click on a template to view all entries for that template, click on a dependency ID to view all entries for the ID, or click on the cache ID to view the entire data cached for that entry.
4. Use the cache monitor to perform basic operations on data in a cache.

   Remove an entry from cache
      Click **Invalidate** when viewing a cache entry.

   Remove all entries for a certain dependency ID
      Click **Invalidate** when viewing entries for a dependency ID.

   Remove all entries for a certain template
      Click **Invalidate** when viewing entries for a template.

   Move an entry to the front of the Least Recently Used queue to avoid eviction
      Click **Refresh** when viewing a cache entry.

   Move an entry from disk to cache
      Click **Send to Memory** when viewing a cache entry on disk.

   Clear the entire contents of the cache
      Click **Clear Cache** while viewing statistics or contents.

   Clear the contents on the ESI processors
      Click **Clear Cache** while viewing ESI statistics or contents.

   Clear the contents of the disk cache
      Click **Clear Disk** while viewing disk contents.

Configuring cacheable objects with the cachespec.xml file

Define cacheable objects inside the `cachespec.xml`, found inside the Web module `WEB-INF` or enterprise bean `META-INF` directory.

You can place a global `cachespec.xml` in the application server properties directory, but the recommended method is to place the cache configuration file with the deployment module. The root element of the `cachespec.xml` file is *<cache>*, which contains *<cache-entry>* elements.

Within a <cache-entry>...</cache-entry> element are parameters that allow you to complete the following tasks to enable the dynamic cache with the `cachespec.xml` file:

1. Develop a `cachespec.xml` file.
   a. Create a caching configuration file.

      In the `<install_root>/properties` directory, locate the `cachespec.sample.xml` file.
   b. Copy the `cachespec.sample.xml` file to `cachespec.xml` in Web module `WEB-INF` or enterprise bean `META-INF` directory.
2. Define the cache-entry elements necessary to identify the cacheable objects. See the topic ″Cachespec.xml file″ for a list of elements.
3. Develop cache-ID rules.

   To cache an object, WebSphere Application Server must know how to generate unique IDs for different invocations of that object. The <cache-id> element performs that task. Each cache entry can have multiple cache-ID rules that execute in order until either a rule returns non-empty cache-ID or no more rules remain to execute. If none of the cache-ID generation rules produce a valid cache ID, then the object is not cached. Develop these IDs in one of two ways:
   - Use the <component> element defined in the cache policy of a cache entry (recommended)
   - Write custom Java code to build the ID from input variables and system state

To configure the cache entry to use the IdGenerator, specify your IdGenerator in the XML file by using the `<idgenerator>` tag, for example:

```
<cache-entry>
     <class>servlet</class>
     <name>/servlet/CommandProcessor</name>
<cache-id>
     <idgenerator>com.mycompany.SampleIdGeneratorImpl</idgenerator>
     <timeout>60</timeout>
</cache-id>
</cache-entry>
```

4. Specifying dependency ID rules. Use dependency ID elements to specify additional cache group identifiers that associate multiple cache entries to the same group identifier.

   The dependency ID is generated by concatenating the dependency ID base string with the values returned by its component elements. If a required component returns a null value, then the entire dependency ID does not generate and is not used. You can validate the dependency IDs explicitly through the WebSphere Dynamic Cache API, or use another cache-entry <invalidation> element. Multiple dependency ID rules can exist per cache-entry. All dependency ID rules separately execute. See the topic ″Cachespec.xml file″ for a list of <component> elements.
5. Invalidate other cache entries as a side effect of this object execution, if relevant. You can define invalidation rules in exactly the same manner as dependency IDs. However, the IDs that generate by invalidation rules are used to invalidate cache entries that have those same dependency IDs.

   The invalidation ID is generated by concatenating the invalidation ID base string with the values returned by its component element. If a required component returns a null value, then the entire invalidation ID is not generated and no invalidation occurs. Multiple invalidation rules can exist per cache-entry. All invalidation rules separately execute.
6. Verify the cacheable page.

Typically you declare several <cache-entry>...</cache-entry> elements inside a `cachespec.xml` file.

The dynamic cache responds to changes in this file. When new versions of the `cachespec.xml` are detected, the old policies are replaced. Objects cached through the old policy file are not automatically invalidated from the cache; they are either reused with the new policy or eliminated from the cache through its replacement algorithm.

For each of the three IDs (cache, dependency, invalidation) generated by cache entries, a <cache-entry> can contain multiple elements. The dynamic cache will execute the <cache-id> rules in order, and the first one that successfully generates an ID will be used to cache that output. If the object is to be cached, each one of the <dependency-id> elements will be executed to build a set of dependency IDs for that cache entry. Finally, each of the <invalidation> elements will be executed, building a list of IDs that the dynamic cache will invalidate, whether or not this object is cached.

Verifying the cacheable page

Verify the cacheable page by following these steps:
1. View the snoop servlet in the default application by accessing the URI: `/snoop`
2. Invoke and reload the URI several times using a different Web browser or using different parameters. This action returns the same output for the snoop servlet. The snoop servlet is now operating incorrectly, because it displays the request information from its first invocation rather than from the current request.
3. Inspect the entry in the cache with the dynamic cache monitor.

Cachespec.xml file

The cache parses the `cachespec.xml` file on startup, and extracts from each <cache-entry> element a set of configuration parameters. Every time a new servlet or other cacheable object initializes, the cache attempts to match each of the different cache-entry elements, to find the configuration information for that object. Different cacheable objects have different <class> elements. You can define the specific object a cache policy refers to using the <name> element.

Location

The `cachespec.xml` file is found inside the `WEB-INF` directory of a Web module.

You can place a global `cachespec.xml` file in the application server properties directory, but the recommended method is to place the cache configuration file with the deployment module. (To place the cache configuration file with the deployment module, use the Assembly ToolkitApplication Assembly Tool (AAT) to define the cacheable objects.

The root element of the `cachespec.xml` file is *`cache`*, which contains *`cache-entry`* elements.

The `cachespec.dtd` file is available in the application server properties directory.

Usage notes

Each cache entry must specify certain basic information that the dynamic cache uses to process that entry. This section explains the function of each cache entry element of the `cachespec.xml` file including:
- class
- name
- sharing-policy
- property
- cache-id

class

```
<class>command | servlet | webservice</class>
```

This element is required and governs how the application server interprets the remaining cache policy definition. The value `servlet` refers to servlets and JavaServer Pages (JSP) files deployed in the WebSphere Application Server servlet engine. The `object` class extends the servlet with special component types for Web services requests. Finally, the value `command` refers to classes using the WebSphere command programming model. The following examples illustrate the `class` element:

```
<class>command</class>
<class>servlet</class>
<class>webservice</class>
```

name

<name>*name*</name>

where *name* is the fully qualified class name of the command, servlet, or object.

There are two ways to use <name> to specify a cacheable object:
- For commands and objects, this required element must include the package name, if any, and class name, including a trailing `.class`, of the configured object.
- For servlets and JSP files, if the `cachespec.xml` file is in the WebSphere Application Server properties directory, this required element must include the full URI of the JSP file or servlet to cache. For servlets and JSP files, if the `cachespec.xml` file is in the Web application, this required element can be relative to the specific Web application context root.

  Note: The preferred location of the `cachespec.xml` file is in the Web application, not the properties directory.

You can specify multiple <name> elements within a <cache-entry> if you have different mappings that refer to the same servlet.

The following examples illustrate the `name` element:

```
<name>com.mycompany.MyCommand.class</name>
<name>default_host:/servlet/snoop</name>
<name>com.mycompany.beans.MyJavaBean</name>
<name>mywebapp/myjsp.jsp</name>
```

sharing-policy

```
<sharing-policy> not-shared | shared-push | shared-pull </sharing-policy>
```

When working within a cluster with a distributed cache, these values determine the sharing characteristics of entries created from this object. If this element is not present, a not-shared value is assumed. Also, in non-distributed environments, not-shared is the only valid value. This property does not affect distribution to Edge servers through the Edge fragment caching property.

Value	Description
not-shared	Cache entries for this object are not shared among different application servers. These entries can contain non-serializable data. For example, a cached servlet can place non-serializable objects into the request attributes, if the <class> type supports it.
shared-push	Cache entries for this object are automatically distributed to the dynamic caches in other application servers or cooperating Java virtual machines (JVMs). Each cache has a copy of the entry at the time it is created. These entries cannot store non-serializable data.
shared-pull	Cache entries for this object are shared between application servers on demand. If an application server gets a cache miss for this object, it queries the cooperating application servers to see if they have the object. If no application server has a cached copy of the object, the original application server executes the request and generates the object. These entries cannot store non-serializable data. This mode of sharing is not recommended.
shared push-pull	Cache entries for this object are shared between application servers on demand. When an application server generates a cache entry, it broadcasts the cache ID of the created entry to all cooperating application servers. Each server then knows whether an entry exists for any given cache ID. On a given request for that entry, the application server knows whether to generate the entry or pull it from somewhere else. These entries cannot store non-serializable data.

The following example shows a sharing policy:

```
<sharing-policy>not-shared</sharing-policy>
```

property

<property name="*key*">*value*</property>

where *key* is the name of the property defined for this cache entry element, and *value* is the corresponding value.

You can set optional properties on a cacheable object, such as a description of the configured servlet. The class determines valid properties of the cache entry. At this time, the following properties are defined:

Property	**Valid classes**	**Value**
ApplicationName	All	Overrides the J2EEName application ID so that multiple applications can share a common cache ID namespace.
EdgeCacheable	Servlet	True or false. Default is false. If the property is true, then the given servlet or JSP file is externally requested from an Edge Server. Whether or not the servlet or JSP file is cacheable, depends on the rest of the cache specification.
ExternalCache	Servlet	Specifies the external cache name. The external cache name needs to match the external cache group name.
consume-subfragments	Servlet or Web service	True or false. Default is false. When a servlet is cached, only the content of that servlet is stored, and includes placeholders for any other fragments to which it includes or forwards. Consume-subfragments (CSF) tells the cache not to stop saving content when it includes a child servlet. The parent entry, the one marked CSF, includes all the content from all fragments in its cache entry, resulting in one big cache entry that has no includes or forwards, but the content from the whole tree of entries. This can save a significant amount of application server processing, but is typically only useful when the external HTTP request contains all the information needed to determine the entire tree of included fragments.
alternate_url	Servlet	Specifies the alternate URL used to invoke the servlet or JSP file. The property is valid only if the EdgeCacheable property also is set for the cache entry.
persist-to-disk	All	True or false. Default is true. When this property is set to false, the cache entry is not written to the disk when overflow or server stopping occurs.
save-attributes	Servlet	True or false. Default is true. When this property is set to false, the request attributes are not saved with the cache entry.

cache-id

To cache an object, the application server must know how to generate a unique ID for different invocations of that object. These IDs are built either from user-written custom Java code or from rules defined in the cache policy of each cache entry. Each cache entry can have multiple cache ID rules that are executed in order until either:

- A rule returns a non-empty cache ID, or
- No more rules are left to execute.

If none of the cache ID generation rules produce a valid cache ID, the object is not cached.

Each `cache-id` element defines a rule for caching an object and is composed of the sub-elements component, timeout, priority, and property. The following example illustrates a cache-id:

```
<cache-id>component*| timeout? | priority? | property* </cache-id>
```

component sub-element

Use the component sub-element to generate a portion of the cache ID. Each component sub-element consists of the attributes `id`, `type`, and `ignore-value`, and the elements `method`, `field`, `required`, `value`, and `not-value`.

- Use the `id` attribute to identify the component.
- Use the `type` attribute to identify the type of component. The following table lists the values for the type.

Type	Valid classes	Meaning
method	command	Calls the indicated method on the command or object
field	command	Retrieves the named field in the command or object
parameter	servlet	Retrieves the named parameter value from the request object
parameter-list	servlet	Retrieves a list of values for the named parameter
session	servlet	Retrieves the named value from the HTTPSession
cookie	servlet	Retrieves the named cookie value
attribute	servlet	Retrieves the named request attribute
header	servlet and Web service	Retrieves the named request header
pathInfo	servlet	Retrieves the pathInfo from the request
servletpath	servlet	Retrieves the servlet path
locale	servlet	Retrieves the request locale
SOAPEnvelope	Web service	Retrieves the SOAPEnvelope from a Web services request. An ID attribute of `Hash` uses a Hash of the SOAPEnvelope, while `Literal` uses the SOAPEnvelope as received.
SOAPAction	Web service	Retrieves the SOAPAction header, (if available), for a Web services request.
serviceOperation	Web service	Retrieves the service operation for a Web services request

serviceOperationParameter	Web service	Retrieves the specified parameter from a Web services request

- Use the `ignore-value` attribute to specify whether or not to use the value returned by this component in cache ID formation. This is an optional attribute with a default value of false. If the value is true, only the ID of the component is used when creating a cache ID, or no output is used when creating a dependency or invalidation ID.
- Use the **method** element to call a void method on a returned object. You can infinitely nest method and field objects in any combination. The method must be public and is not valid for edge-cacheable components. For example:

  ```
  <component id="getUser" type="method"><method>getUserInfo
  <method>getName</method></method></component>
  ```

  This method is equivalent to getUser().getUserInfo().getName()
- Use the **field** element to access a field in a returned object. You can infinitely nest method and field objects in any combination. The field must be public. Not valid for edge-cacheable components. For example:

  ```
  <component id="getUser" type="method"><method>getUserInfo
  <field>name</field></method></component>
  ```

  This method is equivalent to getUser().getUserInfo().name
- Use the **required** element to specify whether or not this component must return a non-null value for this cache ID for it to represent a valid cache. If set to `true`, this component must return a non-null value for this cache ID to represent a valid cache ID. If set to `false`, the default, a non-null value is used in the formation of the cache ID and a null value means that this component is not used at all in the ID formation. For example:

  ```
  <required>true</required>
  ```
- Use the **value** element to specify values that must match to use this component in cache ID formation. For example:

  ```
  <component id="getUser" type="method"><value>blue</value>
  <value>red</value> </component>
  ```
- Use the **not-value** element to specify values that must not match to use this component in cache ID formation. This method is similar to `<value>`, but instead prescribes the defined values from caching. You can use multiple `<not-value>` elements when there is more than one invalid value. For example:

  ```
  <component id="getUser" type="method">
  <required>true</required>
  <not-value>blue</not-value>
  <not-value>red</not-value></component>
  ```

The component element can have either a `method` or a `field` element, or a `value` or a `not-value` element. The `method` and `field` elements apply only to commands. The following example illustrates the attributes of a component element:

```
<component id="isValid" type="method" ignore-value="true"><component>
```

timeout sub-element

The timeout sub-element is used to specify a time-to-live (TTL) value for the cache entry. For example,

```
<timeout>value</timeout>
```

where *value* is the amount of time, in seconds, to keep the cache entry. If 0, or a negative value is specified, the cache entry is kept indefinitely.

priority sub-element

Use the priority sub-element to specify the priority of a cache entry in a cache. The priority weighting is used by the least recently used (LRU) algorithm, of the cache to decide which entries to remove from the cache if the cache runs out of storage space. For example,

```
<priority>value</priority>
```

where *value* is a positive integer between 1 and 255 inclusive.

property sub-element

Use the property sub-element to specify generic properties for the cache entry. For example,

```
<property name="key">value</property>
```

where *key* is the name of the property to define, and *value* is the corresponding value.

For example:

```
<property name="description">The Snoop Servlet</property>
```

Property	Valid classes	Meaning
sharing-policy/timeout/priority	All	Overrides the settings for the containing cache entry when the request matches this cache ID.
EdgeCacheable	servlet	Overrides the settings for the containing cache entry when the request matches this cache ID.

idgenerator and metadatagenerator elements

Use the `idgenerator` element to specify the class name loaded for the generation of the cache ID. The IdGenerator must implement the `com.ibm.websphere.servlet.cache.IdGenerator` interface. The IdGenerator must build and set cache IDs, group IDs and invalidation IDs. An example of the `idgenerator` element follows:

```
<idgenerator> classname classname </idgenerator>
```

(where classname= Fully qualified name of the class to use)

Use the `metadatagenerator` element to specify the class name loaded for the metadata generation cache ID. The MetadataGenerator class must implement the `com.ibm.websphere.servlet.cache.MetaDataGenerator` interface. The MetadataGenerator defines properties like timeout, external caching or generic properties. An example of the `metadatagenerator` element follows:

```
<metadatagenerator> classname classname </metadatagenerator>
```

(where classname= Fully qualified name of the class to use)

Configuring command caching

Cacheable commands are stored in the cache for re-use with a similar mechanism for servlets and Java Server Pages (JSP) files. However, in this case, the unique cache IDs are generated based on methods and fields present in the command as input parameters. For example, a **GetStockQuote** command can have a symbol as its input parameter.

A unique cache ID can generate from the name of the command, plus the value of the symbol.

To use command caching you must:
Create a command.

1. Define an interface. The Command interface specifies the most basic aspects of a command.

   You must define the interface that extends one or more of the interfaces in the command package. The command package consists of three interfaces:
   - TargetableCommand
   - CompensableCommand
   - CacheableCommand

   In practice, most commands implement the TargetableCommand interface, which allows the command to execute remotely. The code structure of a command interface for a targetable command follows:

   ```
   ...
   import com.ibm.websphere.command.*;
   public interface MyCommand extends TargetableCommand {
         // Declare application methods here
   }
   ```
1. Provide an implementation class for the interface. Write an interface that extends the CacheableCommandImpl class and implements your command interface. This class contains the code for the methods in your interface, the methods inherited from extended interfaces like the CacheableCommand interface, and the required or abstract methods in the CacheableCommandImpl class.

   You can also override the default implementations of other methods provided in the CacheableCommandImpl class.

Command class

To write a command interface, extend one or more of the three interfaces included in the command package. The base interface for all commands is the Command interface. This interface provides only the client-side interface for generic commands and declares three basic methods:
- **isReadyToCallExecute.** This method is called on the client side before the command passes to the server for execution.
- **execute.** This method passes the command to the target and returns any data.
- **reset.** This method reverts any output properties to the values they had before the execute method was called so that you can reuse the object.

The implementation class for your interface must contain implementations for the isReadyToCallExecute and reset methods.

CacheableCommandImpl class

Commands are implemented by extending the class CacheableCommandImpl, which implements the CacheableCommand interface.

The CacheableCommandImpl class is an abstract class that provides implementations for some of the methods in the CacheableCommand interface, for example, setting return values. This class declares additional methods that the application must implement, for example, how to execute the command.

The code structure of an implementation class for the CacheableCommand interface follows:

```
...
import com.ibm.websphere.command.*;
public class MyCommandImpl extends CacheableCommandImpl
implements MyCommand {
// Set instance variables here      ...
// Implement methods in the MyCommand interface      ...
// Implement abstract methods in the CacheableCommandImpl class
...
}
```

Example: Caching a command object

This example of command caching is a simple stock quote command.

The following is a stock quote command bean. It accepts a ticker as an input parameter and produces a price as its output parameter.

```
public class QuoteCommand extends CacheableCommandImpl
{
    private String ticker;
    private double price;
    // called to validate that command input parameters have been set
    public boolean isReadyToCallExecute() {
      return (ticker!=null);
    }
    // called by a cache-hit to copy output properties to this object
    public void setOutputProperties(TargetableCommand fromCommand) {
        QuoteCommand f = (QuoteCommand)fromCommand;
        this.price = f.price;
    }

   // business logic method called when the stock price must be retrieved
    public void performExecute()throws Exception {...}

    //input parameters for the command
    public void setTicker(String ticker) { this.ticker=ticker;}
    public String getTicker() { return ticker;}

    //output parameters for the command
    public double getPrice()  { return price;};
}
```

To cache the above command object using the stock ticker as the cache key and using a 60 second time-to-live, use the following cache policy:

```
<cache>
 <cache-entry>
  <class>command</class>
  <sharing-policy>not-shared</sharing-policy>
  <name>QuoteCommand</name>
   <cache-id>
    <component type="method" id="getTicker">
     <required>true</required>
    </component>
    <priority>3</priority>
    <timeout>60</timeout>
   </cache-id>
 </cache-entry>
</cache>
```

Example: Caching Web services

The following is a example of building a set of cache policies for a simple Web services application. The application in this example stores stock quotes, and has operations to read, update the price of, and buy a given stock symbol.

Following are two SOAP message examples that the application can receive, with accompanying HTTP Request headers.

The first message sample contains a SOAP message for a GetQuote operation, requesting a quote for IBM. This is a read-only operation that gets its data from the back-end, and is very cacheable. In this example the SOAP messasge is cached and a timeout is placed on its entries to guarantee the quotes it returns are not too out of date.

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>
```

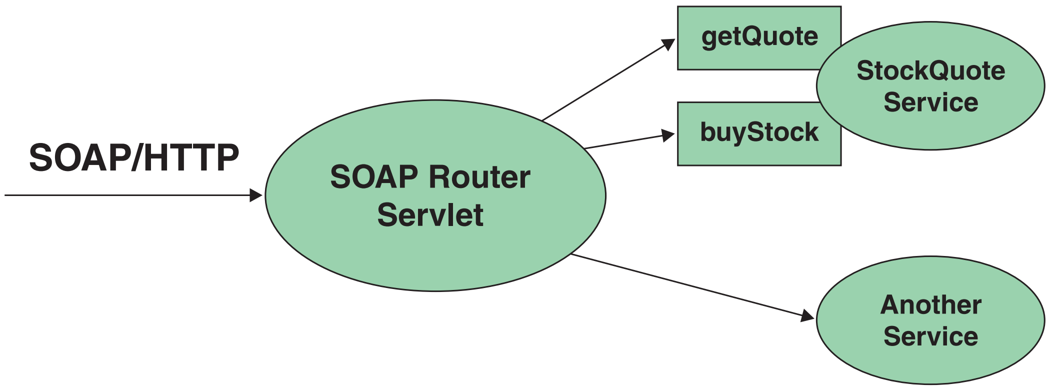
The SOAPAction HTTP header in the request is defined in the SOAP specification and is used by HTTP proxy servers to dispatch requests to particular HTTP servers. WebSphere Application Server dynamic cache can use this header in its cache policies to build IDs without having to parse the SOAP message.

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:getQuote>
</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 our 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 we have a servlet which is used to manage a simple news site. This servlet uses the query parameter ″action″ to determine whether the request is being used to ″view″ news or ″update″ news (used by the administrator). Further, another query parameter ″category″ is used to select the news category. Further, suppose that this site supports an optional customized layout, which 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 with `cachespec.xml`, using the information provided to you:

1. Define the cache-entry elements necessary to identify the servlet. In this case, the servlet's URI is ″newscontroller″ so this will be our cache-entry's name element. Also, since we are caching a servlet/JavaServer Page (JSP), the cache-entry class is ″servlet″.

   ```
   <cache-entry>
   <name> /newscontroller </name>
   <class>servlet  </class>
    </cache-entry>
   ```

2. Define cache ID generation rules. For this servlet, we only want to cache when action=view, so one component of the cache ID will be 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 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>
</cache-entry>
```

3. Define dependency ID rules. For this servlet, a dependency ID will be added for the category. Later, when the category is invalidated due to an update event, all views of that news category will be invalidated. After adding our dependency-id, the 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>
</cache-entry>
```

4. Define invalidation rules. Since we defined a category dependency ID, we will now 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 will not add to the output of the invalidation ID, but will only determine whether or not the invalidation ID is created and executed. 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>
```

Cache monitor

Cache monitor is an installable Web application that provides a real-time view of the current state of dynamic cache. You use it to help verify that dynamic cache is operating as expected. The only way to manipulate the data in the cache is by using the cache monitor. It provides a GUI interface to manually change data.

Cache monitor provides a way to:

- **Verify the configuration of dynamic cache**

  The WebSphere Application Server adminstrative console provides ways to enable the dynamic cache service and configure properties, such as maximum size of the cache and disk offload location, as well as advanced features such as controlling external caches. Cache monitor offers a way for dynamic cache users to verify the configuration of the dynamic cache by providing a convenient view of the configured features and properties in the cache monitor.
- **Verify the cache policies**

  To cache an object, WebSphere Application Server must know how to generate unique IDs for different invocations of that object. This is performed by providing rules for each cacheable object in the `cachespec.xml` file, found inside the Web module `WEB-INF` or `enterprise bean META-INF` directory. Each cacheable object can have multiple cache ID rules that execute in sequence until either a rule returns a cache ID or no more rules remain to execute. If none of the cache ID generation rules produce a valid cache ID, then the object is not cached. Since there can be multiple `cachespec.xml` files with multiple cache ID rules, cache monitor provides a convenient way to verify the policies of each object. It offers a view of all the cache polices currently loaded in dynamic cache. This view is also convenient to verify that the `cachespec.xml` file was read by the dynamic cache without errors.
- **Monitor cache statistics**

  Cache monitor provides a view of the essential cache data, such as number of cache hits, cache misses, and number of entries in cache. This helps to tune the cache configuration optimally to get the best performance improvement out of dynamic cache. For example, if the number of used entries is often high, and entries are being removed and recreated, one might consider increasing the maximum size of the cache or enabling disk offload.
- **Monitor the data flowing through the cache**

  Once a cacheable object is invoked, dynamic cache creates a cache entry for it that contains the output of the execution and metadata, such as time to live, sharing policy, etc. Entries are distinguished by a unique ID string that is based on the rules specified in the `cachespec.xml` file for this objects name. Objects with the same name may generate multiple cache IDs for different invocations, based on request parameters and attributes for each invocation. Cache monitor provides a view of all the cache entries currently in cache, based on the unique ID. It also provides a view of the group of cache entries that share a common name (also known as template). Cache entries can also be grouped together by a dependency ID, which is used to invalidate the entire group of entries dependent on a common entity. Therefore, cache monitor also provides a view of the group of cache entries that share a common dependency ID.

  For each entry, cache monitor also displays metadata, such as time to live, priority and sharing-policy, and provides a view of the output that has been cached. This helps the customer to verify which pages have been cached, that the pages have been cached with the right attributes such as time to live, priority, etc., and that the pages have the right content.
- **Monitor the data in the edge cache**

  Dynamic cache provides support to recognize the presence of an Edge Side Include (ESI) processor and to generate ESI include tags and appropriate cache policies for edge cacheable fragments. The ESI processor has the ability to cache whole pages, as well as fragments, providing a higher cache hit ratio. There can be multiple ESI processors running on multiple hosts configured for caching.

  Cache monitor provides a list of all ESI processes and their hosts that are enabled for caching. It also provides a way to select a host or a processor, and view its edge cache statistics as well as current cache entries.
- **View the data offloaded to the disk**

By default, when the number of cache entries reaches the configured limit for a given server, eviction of cache entries occurs, allowing new entries to enter the cache service. The dynamic cache includes the disk offload feature that copies the evicted cache entries to disk for future access. Cache monitor offers a view of the content offloaded to disk that corresponds to the view of contents cached in memory.

- **Manage the data in the cache**

  Besides displaying cache content, cache monitor also provides some basic operations on the data in the cache:
  - Removing an entry from the cache
  - Removing all entries for a certain dependency ID
  - Removing all entries for a certain name (template)
  - Moving an entry to the front of the least recently used queue to avoid eviction
  - Moving an entry from the disk to the cache
  - Clearing the entire contents of the cache
  - Clearing the contents of the disk cache

  These functions are useful for dynamic cache customers, as they provide a way to manually change the state of the cache without having to restart the server.

Edge cache statistics

Cache monitor provides a view of the edge cache statistics.

The following statistics are available:
- **ESI Processors**. Number of processes configured as edge caches.
- **Number of Edge Cached Entries**. Number of entries currently cached on all edge servers and processes.
- **Cache Hits**. Number of requests that match entries on edge servers.
- **Cache Misses By URL.** A cache policy does not exist on the edge server, for the requested template.

  Note:
  - The initial ESI request for a template that has a cache policy on WebSphere Application Server will result in a miss.
  - Every request for a template that does not have a cache policy on WebSphere Application Server will result in a miss by URL on the edge server.
- **Cache Misses By Cache ID**. The policy for the requested template exists on the edge server, and a cache ID is created, based on the ID rules and the request attributes, but the cache entry for this ID does not exist.

  Note: If the policy exists on the edge server for the requested template, but a cache ID match is not found, based on the ID rules and the request attributes, it is not treated as a cache miss.
- **Cache Timeouts**. Number of entries removed from the edge cache, based on the timeout value.
- **Evictions**. Number of entries removed from the edge cache, due to invalidations received from WebSphere Application Server.

Troubleshooting the dynamic cache service

Complete the steps below to resolve problems that you think are related to the dynamic cache service.

1. Review the JVM logs for your application server. Messages prefaced with *DYNA* result from dynamic cache service operations.
   a. View the JVM logs for your application server. Each server has its own JVM log file. For example, if your server is named *Member_1*, the JVM log is located in the subdirectory *install_root*/logs/Member_1. To use the administration console to review the JVM logs, click **Troubleshooting > Logs and Trace > *server_name* > JVM Logs > Runtime > View**.
   b. Find any messages prefaced with *DYNA* in the JVM logs, and write down the message IDs. A sample message having the message ID *DYNA0030E* follows:

```
DYNA0030E: "property" element is missing required attribute "name".
```

c. Find the message for each message ID in the WebSphere Application Server InfoCenter. In the InfoCenter navigation tree, click ***product_name*** **> Reference > Messages > DYNA** to view dynamic cache service messages.

d. Read the message **Explanation** and **User Action** statements. A search for the message ID `DYNA0030E` displays a page having the following message:

```
DYNA0030E: "property" element is missing required attribute "name".
  Explanation: A required attribute was missing in the cache configuration.
  User Action: Add the required attribute to your cache configuration file.
```

This explanation and user action suggests that you can fix the problem by adding or correcting a required attribute in the cache configuration file.

e. Try the solutions stated under **User Action** in the *DYNA* messages.

2. Use the cache monitor to determine whether the dynamic cache service is functioning as expected. The cache monitor is an installable Web application that displays simple cache statistics, cache entries, and cache policy information.
3. If you have completed the preceding steps and still cannot resolve the problem, contact your IBM software support representative. Use the collector tool (collector.bat or collector.sh located in the `bin` directory) to gather trace information and other configuration information for the support team to diagnose the problem. The collector tool gathers dynamic cache service files and packages them into a JAR file. The IBM representative can specify when and where to send the JAR file. The IBM representative might ask you to complete a diagnostic trace. To enable tracing in the administrative console, click **Troubleshooting > Logs and Trace >** ***server_name*** **> Diagnostic Trace** and specify **Enable trace with the following specification**. The IBM representative can tell you what trace specification to enter. Note that dynamic cache trace files can become large in a short period of time; you can limit the size of the trace file by starting the trace, immediately recreating the problem, and immediately stopping the trace.

   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.

Troubleshooting tips for the dynamic cache service

The dynamic cache service works within an application server Java virtual machine (JVM), intercepting calls to cacheable objects. This article describes some common run-time and configuration problems and remedies.

Servlets are not cached

Recommended response	Enable servlet caching. On the Web container page of the administrative console, select the **Enable servlet caching** check box.

Cache entries are not written to disk

Explanation	Cache entries are written to disk when the cache is full and new entries are added to the memory cache. Cache entries also are written to disk when the `flushToDiskOnStop` system property is set and the server is stopped.
Recommended response	Verify that **Disk offload** is enabled on the Dynamic Cache Service page of the administrative console. Also verify that cache entries written to disk are serializable and do not have the `PersistToDisk` configuration set to `false`.

Some servlets are not replicated or written to disk

Recommended response	Ensure that the attributes and response are serializable. If the you do not want to store the attributes, use the following property in your cache policy: `<property name="save-attributes">false</property>`

Dynamic cache service does not cache fragments on the Edge

Recommended response	Set the `EdgeCacheable` property to `true` in the cache policy for those entries that are to be cached on the Edge. `<property name="EdgeCacheable">true</property>`

Dynamic cache invalidations are not sent to the IBM HTTP Server (IHS) plug-in

Explanation	The `DynaCacheEsi.ear` file is required to send invalidations to external caches.
Recommended response	Install `DynaCacheEsi.ear` using the administrative console.

Cache entries are evicted often

Problem	The cache is full and new entries are added to the cache.
Explanation	Cache entries are evicted when the cache is full and new entries are added to the cache. A LRU eviction mechanism removes the least recently used entry to make space for the new entries.
Recommended response	Either enable **Disk offload** on the Dynamic Cache Service page of the administrative console so the entries are written to disk. Or, increase the cache size to accommodate more entries in the cache.

Cache entries in disk with timeout set to 0 expire after one day

Explanation	The maximum lifetime of an entry in disk cache is 24 hours. A timeout of `0` in the cache policy configures these entries to stay in disk cache for one whole day, unless they are evicted earlier.
Recommended response	Set the timeout for the cache policy to a number greater than `0`.

I cannot monitor cache entries on the Edge

Explanation	Use the cache monitor for monitoring contents in memory cache, disk cache and external caches (Edge cache). For the ESI processor's cache to be visible in the cache monitor, the `DynaCacheEsi.ear` application must be installed and the `esiInvalidationMonitor` property must be set to `true` in the `plugin-cfg.xml` file.
Recommended response	Install the DynaCacheEsi.ear application and set the `esiInvalidationMonitor` property to `true` in the `plugin-cfg.xml` file.

I want to cache static contents using the dynamic cache service

Explanation	You can cache static contents using the dynamic cache service. Static contents in WebSphere application server are served by the `SimpleFileServlet` file.

Recommended response	Create a cache policy for the class `com.ibm.ws.webcontainer.servlet.SimpleFileServlet.class` to cache static contents. It is advisable to use the dynamic cache service for caching more expensive dynamic contents than static contents.

I want to tune cache for my environment

Recommended response	Use the Tivoli Performance viewer to study the caching behavior for your applications. Also, do the following: • Increase the priority of cache entries that are expensive to regenerate. • Modify timeout of entries so that they stay in memory as long as they are valid. • Enable disk offload to store LRU evicted entries. • Increase the cache size.

Chapter 15. Managing user profiles

Note: User Profile Manager API is deprecated in the current release, and there is no replacement available.

IBM WebSphere Application Server provides a service for processing user profiles, called the *User Profile Manager*. The service is provided in the form of an EJB entity bean that servlets can call whenever they are required to access a user profile.

The key activities for implementing user profiles are summarized.

1. Customize the user profile support as necessary. Options include:
   - Using the data representation class with the name-value pairs it currently supports (no action required)
   - Extending the data representation class to support additional, arbitrary name-value pairs
   - Adding columns to the base user profile representation
   - Extending the User Profile enterprise bean to import existing databases

   Evaluate whether the user profile representation provided by IBM represents the kind of data you want to keep about your users. You might find it desirable to customize the IBM user profile support.
2. Create or modify servlets to use the User Profile Manager and related user profile support classes to maintain user profiles on behalf of Web applications.
3. Assemble your application.
4. Deploy your application.
5. Ensure the administrator appropriately configures User Profile Managers using userprofile.xml file. If the programmer and administrator are not the same person, the programmer might need to provide settings information to the administrator, based on how the programmer implemented user profiles.

User profile

Some applications collect data about the users with which they interact. The data is stored in a database. The next time the user interacts with the application, the application recalls the data.

Because the application already knows something about the user, it can provide the user with a more personalized experience.

User profiles provide a means by which a company can maintain and manage database tables containing fields for demographic data and use those tables to interact with a database of individual customers or other users on the company system.

For example, when a repeat user logs onto a Web site that supports user profiles, the Web site can display headlines and advertising tailored to the shopping preferences of that user. The site can address the user by logon name.

An application implementing user profiles requires database access for storing the user profile data it gathers.

UserProfileManager class

Servlets and other application building blocks requiring user profile support should make calls to the class:

```
com.ibm.websphere.userprofile.UserProfileManager
```

The class supports the following functions:

- Creating and deleting user profiles

- Getting and updating (cached and immediate) to and from the database
- Getting user profiles for read-only tasks
- Performing queries on database columns

User profile development options

The application developer has a few options for customizing the user profile support provided by IBM WebSphere Application Server. The Related information provides instructions and additional details about each option.

Extending the data represented in user profiles

Web applications can maintain several pieces of data about users. You can extend the data representation to allow the collection of arbitrary name-value pairs.

Use the following interface with the com.ibm.websphere.userprofile.UserProfileExtender class to extend a user profile hash table:

```
com.ibm.websphere.userprofile.UserProfileProperties
```

This action enables you to place arbitrary name-value pairs in the user profile. Extending the hash table is similar to using the java.util.Dictionary class in the base JDK 1.x, or any of the classes that extend it.

Adding columns to the base user profile implementation

Application developers can customize user profiles by adding columns to the base user profile implementation. Adding new columns is accomplished by implementing the interface:

```
com.ibm.websphere.userprofile.UserProfileExtender
```

and extending the base class:

```
com.ibm.servlet.personalization.userprofile.UserProfile
```

The application developer can add columns to but not delete columns from the base implementation.

Adding columns is a two-step process, as follows:
1. Extend the UserProfile class.
2. Modify your existing servlets to use the new columns.

Several examples are available to demonstrate how to extend the base user profile implementation and utilize the extension with a servlet.

Example	Description
UPServletExample.java	Demonstrates how a servlet opens a user profile and prints the fields contained within.
UserProfileExtendedSample.java	Shows how to extend the UserProfile class to add a column to the user profile for a cellular phone number. The WebSphere Application Server administrator configures the User Profile Manager to point to the extended class.
UPServletExampleExtended.java	Shows how to modify the UPServletExample servlet to include the cellular phone number in the output.
UserProfileExtended.java	Shows how to extend a hash table to place arbitrary name-value pairs into the user profile.

Example	Description
UPServletExtended.java	Shows how to extend the servlet. When any of the newly added columns are removed or replaced, look for the table named ″USERPROFILE″ in the database to which the user profile is configured and drop that table.

The examples are encoded in HTML for viewing in a browser. The documentation directory also contains nonHTML versions (.java files) that are ready for use.

Extending the User Profile enterprise bean and importing legacy databases

Application developers can extend the User Profile enterprise bean itself and import legacy databases into the user profile. The main advantage in extending the User Profile enterprise bean is to gain the ability to import existing databases into the user profile. You can also extend this enterprise bean to add columns to the base user profile implementation.

Example: UPServletExample.java

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.ibm.websphere.userprofile.UserProfile;
import com.ibm.websphere.userprofile.UserProfileManager;
import com.ibm.websphere.userprofile.UserProfileCreateException;
import com.ibm.websphere.userprofile.UserProfileFinderException;
import com.ibm.websphere.userprofile.UserProfileRemoveException;

//Creates a Userprofile using the new API

public class UPServlet_ReadWrite extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException {

       PrintWriter out;
       res.setContentType("text/html");
       out = res.getWriter();


       UserProfileManager manager = UserProfileManager.getUserProfileManager();
       UserProfile userprofile;



       try {

          //Try creating the UserProfile
          userprofile = manager.addUserProfile("bpink");

       } catch(UserProfileCreateException e1) {

         try {

           //Try finding the existing in readWrite mode.
           //Second argument indicates whether we want to get userprofile
           //in read only mode or read write mode.

           userprofile = manager.getUserProfile("bpink",true);
         } catch(UserProfileFinderException e) {
            e.printStackTrace();
            return;
         }

       }


       //Set the properties

       userprofile.setAddress1("myaddress1");
       userprofile.setAddress2("myaddress2");
       userprofile.setFirstName("Pinkowski");
       userprofile.setSurName("Ben");
       userprofile.setDayPhone("555-6677");
       userprofile.setNightPhone("556-6765");
       userprofile.setCity("MYCITY");
       userprofile.setNation("myCountry");
       userprofile.setEmployer("MyEmployer");
       userprofile.setFax("7823470");

       userprofile.setLanguage("mylanguage");
       userprofile.setEmail("MyEmail@email");
       userprofile.setStateOrProvince("myState");
       userprofile.setPostalCode("xxxxx");

       //Freeing resources held by userprofile
       manager.releaseResources(userprofile);
       userprofile=null;

       //Checking whether it updated the info

       try {

          //Getting the existing userprofile in ReadOnly mode.

          userprofile = manager.getUserProfile("bpink",false);

       } catch(UserProfileFinderException e1) {

          out.println("Error finding ");
          e1.printStackTrace();
          return;
       }


       //Displaying the properties of userprofile

       out.println(userprofile.getAddress1()+"<br>");
       out.println(userprofile.getAddress2()+"<br>");;
       out.println(userprofile.getFirstName()+"<br>");;
       out.println(userprofile.getSurName()+"<br>");
       out.println(userprofile.getDayPhone()+"<br>");;
       out.println(userprofile.getNightPhone()+"<br>");;
       out.println(userprofile.getCity()+"<br>");
       out.println(userprofile.getNation()+"<br>");;
       out.println(userprofile.getEmployer()+"<br>");;
       out.println(userprofile.getFax()+"<br>");;
       out.println(userprofile.getLanguage()+"<br>");;
       out.println(userprofile.getEmail()+"<br>");;
       out.println(userprofile.getStateOrProvince()+"<br>");;
       out.println(userprofile.getPostalCode()+"<br>");

       //Freeing resources held by userprofile
       manager.releaseResources(userprofile);
   }
}
```

Example: UserProfileExtendedSample.java

```
/* ----------------------------------------------------------------
** Copyright 1997-99 IBM Corporation.  All rights reserved.
**
** ----------------------------------------------------------------
*/
package com.ibm.servlet.personalization.userprofile;


import com.ibm.servlet.personalization.userprofile.UserProfile;
import com.ibm.websphere.userprofile.UserProfileExtender;


//Extensions of UserProfile to add new Columns should implement UserProfileExtender
public class UserProfileExtendedSample
   extends com.ibm.servlet.personalization.userprofile.UserProfile
   implements UserProfileExtender {

   //New column that is being added by this
   //derived class.
   public String cellPhone;

   //Manager Class will call this  method to append new Column types.
   //If UserProfile class is extended to append new columns
   //TOTAL COLUMNS: Base Class columns + columns returned by this class

   public  String[]  getNewColumns() {
      //If variable name is "cellPhone," you need to
      //return "cellPhone" in array format. JDBC equivalent will be
      //generated automatically. You can add muliple columns.
      //For multiple columns: String newCol={"fieldName1","fieldName2",...};
      String[] newCol={"cellPhone"};
      return newCol;
   }

   public String getCellPhone() {
      // Need to call this method to
      // get the things from persistence store.
      return(String)getByType("cellPhone");
   }

   public  void setCellPhone(String value) {
      cellPhone = value;
      //Call this method to store the
      //things in persistence store
      setByType("cellPhone", value);
   }
}
```

Example: UPServletExampleExtended.java

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.ibm.websphere.userprofile.UserProfile;
import com.ibm.websphere.userprofile.UserProfileManager;
import com.ibm.websphere.userprofile.UserProfileCreateException;
import com.ibm.websphere.userprofile.UserProfileFinderException;
import com.ibm.websphere.userprofile.UserProfileRemoveException;
import com.ibm.servlet.personalization.userprofile.UserProfileExtendedSample;
```

```
public class UPServletExtendedSample extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        UserProfileManager manager = UserProfileManager.getUserProfileManager();
        UserProfile userprofile;

        PrintWriter out;

        res.setContentType("text/html");
        out = res.getWriter();


        try {

            //try Creating the UserProfile

            userprofile = manager.addUserProfile("bpink");

        } catch( UserProfileCreateException e1) {

            try {  //try finding the existing in readWrite mode

                userprofile = manager.getUserProfile("bpink",true);
            } catch(UserProfileFinderException e) {
                e.printStackTrace();
                return;
            }


        }


        userprofile.setAddress1("myaddress1");
        userprofile.setAddress2("myaddress2");
        userprofile.setFirstName("Pinkowski");
        userprofile.setSurName("Ben");
        userprofile.setDayPhone("555-6677");
        userprofile.setNightPhone("556-6765");
        userprofile.setCity("MYCITY");
        userprofile.setNation("myCountry");
        userprofile.setEmployer("MyEmployer");
        userprofile.setFax("7823470");
        userprofile.setLanguage("mylanguage");
        userprofile.setEmail("MyEmail@email");
        userprofile.setStateOrProvince("myState");
        userprofile.setPostalCode("xxxxx");

        //calling setCellPhone
        ((com.ibm.servlet.personalization.userprofile.UserProfileExtendedSample)
           userprofile).setCellPhone("346-4588");


        //Freeing resources held by userprofile
         manager.releaseResources(userprofile);
         userprofile=null;


        //Checking whether it updated the info

        try {

            //Getting the existing userprofile
            userprofile = manager.getUserProfile("bpink",false);
        } catch( UserProfileFinderException e1) {

            out.println("Error finding ");
            e1.printStackTrace();
            return;
        }



        out.println(userprofile.getAddress1()+"<br>");
        out.println(userprofile.getAddress2()+"<br>");;
        out.println(userprofile.getFirstName()+"<br>");;
        out.println(userprofile.getSurName()+"<br>");
        out.println(userprofile.getDayPhone()+"<br>");;
        out.println(userprofile.getNightPhone()+"<br>");;
        out.println(userprofile.getCity()+"<br>");
        out.println(userprofile.getNation()+"<br>");;
        out.println(userprofile.getEmployer()+"<br>");;
        out.println(userprofile.getFax()+"<br>");;
        out.println(userprofile.getLanguage()+"<br>");;
        out.println(userprofile.getEmail()+"<br>");;
        out.println(userprofile.getStateOrProvince()+"<br>");;
        out.println(userprofile.getPostalCode()+"<br>");


        //Calling getCellPhone
        out.println(((UserProfileExtendedSample)userprofile).getCellPhone()+"<br>");

        //Freeing resources held by userprofile
         manager.releaseResources(userprofile);
         userprofile=null;

        //For getting values by cellPhone
        out.println("<br><br>Retreiving by Cell Phone <br>");
        Enumeration enum = manager.findUserProfiles("cellPhone","346-4588");
        while(enum.hasMoreElements()) {

            com.ibm.websphere.userprofile.UserProfile up =
               (com.ibm.websphere.userprofile.UserProfile)enum.nextElement();
            out.println("first name :"+up.getFirstName()+"<br>");

            //Freeing resources held by userprofile
            manager.releaseResources(up);
        }
    }
}
```

Example: UserProfileExtended.java

```
package com.ibm.servlet.personalization.userprofile;
/* -----------------------------------------------------------------
** Copyright 1997-99 IBM Corporation.  All rights reserved.
**
** -----------------------------------------------------------------
*/
import java.util.*;

import com.ibm.servlet.personalization.userprofile.UserProfile;

import com.ibm.websphere.userprofile.UserProfileExtender;
import com.ibm.websphere.userprofile.UserProfileProperties;

public class UserProfileExtended extends UserProfile implements UserProfileExtender,
   UserProfileProperties {
    //New column that is being added by this
    //derived class.
    public Hashtable properties;
    static String  propCol ="properties";

    //Manager Class will call this  method to append new Column types
    //to SQL Strings. If UserProfile class is extended to append new columns
    //it should implement UserProfileExtender.
    //COLUMNS: Base Class columns + columns returned by this class

    public  String[]  getNewColumns() {
        //if variable name is properties, you need to
        //return "properties" . JDBC equivalent will be
        //generated automatically.
        String[] newCol={propCol};
        return newCol;
    }

    public Object getValue(String key) {
        // Need to call this method to
        // get the things from persistent store
        properties = (Hashtable) getByType(propCol);

        if(properties != null)
            return properties.get(key);

        else return null;

    }

    public  void putValue(String key, Object value) {

        properties =(Hashtable) getByType(propCol);

        if(properties == null)
            properties = new Hashtable();

        properties.put(key,value);

        //store in persistent store
        setByType(propCol, properties);
    }

    public void removeValue(String key) {
        properties = (Hashtable) getByType(propCol);

        if(properties == null)
            return;

        properties.remove(key);

        //store in persistent store
        setByType(propCol, properties);
    }
}
```

Example: UPServletExtended.java

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.ibm.websphere.userprofile.UserProfile;
import com.ibm.websphere.userprofile.UserProfileManager;
import com.ibm.websphere.userprofile.UserProfileCreateException;
import com.ibm.websphere.userprofile.UserProfileFinderException;
import com.ibm.websphere.userprofile.UserProfileRemoveException;
import com.ibm.websphere.userprofile.UserProfileProperties;


public class UPServletExtended extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        UserProfileManager manager = UserProfileManager.getUserProfileManager();
        UserProfile userprofile;

        PrintWriter out;

        res.setContentType("text/html");
        out = res.getWriter();

        try {

            //try Creating the UserProfile

            userprofile = manager.addUserProfile("bpink");

        } catch( UserProfileCreateException e1) {

            try {  //try finding the existing in readWrite mode

                userprofile = manager.getUserProfile("bpink",true);
            } catch(UserProfileFinderException e) {
                e.printStackTrace();
                return;
            }


        }

        userprofile.setAddress1("myaddress1");
        userprofile.setAddress2("myaddress2");
        userprofile.setFirstName("Pinkowski");
        userprofile.setSurName("Ben");
        userprofile.setDayPhone("555-6677");
        userprofile.setNightPhone("556-6765");
        userprofile.setCity("MYCITY");
        userprofile.setNation("myCountry");
        userprofile.setEmployer("MyEmployer");
        userprofile.setFax("7823470");
        userprofile.setLanguage("mylanguage");
        userprofile.setEmail("MyEmail@email");
        userprofile.setStateOrProvince("myState");
        userprofile.setPostalCode("xxxxx");

        //calling putValue

        ((UserProfileProperties)userprofile).putValue("name","HHHHHHH");
        ((UserProfileProperties)userprofile).putValue("Date",new java.util.Date());

        //Freeing resources held by userprofile
        manager.releaseResources(userprofile);
        userprofile=null;

        //Checking whether it updated the info

        try {

          //Getting the existing userprofile

          userprofile = manager.getUserProfile("bpink",false);
          out.println(userprofile.getAddress1()+"<br>");
          out.println(userprofile.getAddress2()+"<br>");;
          out.println(userprofile.getFirstName()+"<br>");;
          out.println(userprofile.getSurName()+"<br>");
          out.println(userprofile.getDayPhone()+"<br>");;
          out.println(userprofile.getNightPhone()+"<br>");;
          out.println(userprofile.getCity()+"<br>");
          out.println(userprofile.getNation()+"<br>");;
          out.println(userprofile.getEmployer()+"<br>");;
          out.println(userprofile.getFax()+"<br>");;
          out.println(userprofile.getLanguage()+"<br>");;
          out.println(userprofile.getEmail()+"<br>");;
          out.println(userprofile.getStateOrProvince()+"<br>");;
          out.println(userprofile.getPostalCode()+"<br>");


          //Getting the values

          out.println(((UserProfileProperties)userprofile).getValue("name")+"<br>");
          out.println(((UserProfileProperties)userprofile).getValue("Date")+"<br>");
          out.println("Removing Values ");
          ((UserProfileProperties)userprofile).removeValue("name");
          ((UserProfileProperties)userprofile).removeValue("Date");
          out.println(((UserProfileProperties)userprofile).getValue("name")+"<br>");
          out.println(((UserProfileProperties)userprofile).getValue("Date")+"<br>");

          //Freeing resources held by userprofile
           manager.releaseResources(userprofile);

        } catch( UserProfileFinderException e1) {

            out.println("Error finding ");
            e1.printStackTrace();
            return;
        }
    }
}
```

userprofile.xml

To *installation_root*/properties, add a file named userprofile.xml in the following format. Specify enterprise bean class names; data wrapper class name; and JNDI names for the read-only bean, read/write bean, and data source (from step 2). You must also add user ID and password information for the JNDI data source

The following example file contains class names as provided in WebSphere Application Server. If data wrapper and enterprise bean class names are extended programmatically, change them accordingly.

```
<?xml version="1.0"?>

   <userprofile>
      <userprofile-enabled>true</userprofile-enabled>
      <userprofile-wrapper-class>
         <classname>
            com.ibm.servlet.personalization.userprofile.UserProfile
         </classname>
      </userprofile-wrapper-class>
      <userprofile-manager-name>
         User Profile Manager
      </userprofile-manager-name>
      <userprofile-bean>
         <readonly-interface>
            com.ibm.servlet.personalization.userprofile.UP_ReadOnly
         </readonly-interface>
         <readwrite-interface>
            com.ibm.servlet.personalization.userprofile.UP_ReadWrite
         </readwrite-interface>
         <readonlyhome-interface>
            com.ibm.servlet.personalization.userprofile.UP_ReadOnlyHome
         </readonlyhome-interface>
         <readwritehome-interface>
            com.ibm.servlet.personalization.userprofile.UP_ReadWriteHome
         </readwritehome-interface>
         <readonly-JNDI-lookupName>UP_ReadOnlyHome</readonly-JNDI-lookupName>
<readwrite-JNDI-lookupName>UP_ReadWriteHome</readwrite-JNDI-lookupName>
      </userprofile-bean>

      <userprofile-store>
         <database-userid></database-userid>
         <database-password></database-password>
         <database-datasource></database-datasource>
      </userprofile-store>

   </userprofile>
```

Chapter 16. Assembling applications with the Assembly Toolkit

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 (EJB) JAR files, and application client archives (JAR). 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.

The Assembly Toolkit replaces the Application Assembly Tool (AAT). The Assembly Toolkit consists of the J2EE Perspective of the WebSphere Studio Application Developer product. With the Assembly Toolkit, you can create and modify J2EE applications and modules, edit deployment descriptors, and map databases.

The Assembly Toolkit is one of the tools provided by the Application Server Toolkit (ASTK). Follow instructions available with the ASTK to install the Assembly Toolkit.

Gather the code artifacts that you want to package into one or more assembled modules. Code artifacts include these items that you have created and unit tested in your favorite integrated development environment:
- Enterprise beans
- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (*connector*) implementations
- Application clients
- Other supporting classes and files

The Assembly Toolkit provides extensive online documentation. The articles on Assembly Toolkit provided in this InfoCenter supplement that documentation.

1. Start the Assembly Toolkit.
2. Optional: Read the online documentation for the Assembly Toolkit.
   - Read the section **Assembly Tool** on the Welcome to the Application Server Toolkit page. To access this page, click **Help > Welcome > Application Server Toolkit**.
   - Click **Help > Help Contents > Assembly Toolkit information**. The displayed documentation provides extensive information about the Assembly Toolkit.
   - Press F1 to access information specific to an Assembly Toolkit view or window.
   - Visit the IBM WebSphere Studio Application Developer InfoCenter at `http://publib.boulder.ibm.com/infocenter/wsphelp/index.jsp`. Click **WebSphere Studio Application Developer > J2EE development**. The documentation in the WebSphere Studio InfoCenter is similar to that in the Assembly Toolkit online information.
   - See the article ″"Assembly Toolkit: Resources for learning" on page 651″ for additional sources.
3. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
4. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
5. Migrate EAR, WAR, enterprise bean JAR files, application client JAR files, or resource adapter RAR files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import the files to the Assembly Toolkit.
6. Optional: Migrate a project from J2EE 1.2 to J2EE 1.3 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.

© Copyright IBM Corp. 2003

   a. In the J2EE Hierarchy 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.
7. 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 application client.
   - Create an enterprise bean (EJB) project.
   - Create a resource adapter (connector) project.
8. Edit the deployment descriptors as needed. You can edit deployment descriptors for enterprise application, Web, application client, and enterprise bean (EJB) modules.
9. Optional: Generate enterprise bean (EJB) to relational database (RDB) mappings for EJB modules.
10. Verify the archive files.
11. Generate code for deployment for EJB modules or for enterprise applications that use EJB modules.
12. Generate code for deployment for Web services-enabled modules or for enterprise applications that use Web service modules.
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 Assembly Toolkit to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important

    Note: Use **Run On Server** for unit testing only. Application Server Toolkit controls the WebSphere Application Server installation and, when an application is published remotely, the Toolkit overwrites the server configuration file for that server. Do not use on production servers.

    For instructions on remote testing, see the article ″Setting Up a Remote WebSphere Application Server in WebSphere Studio V5″ at http://www7b.boulder.ibm.com/wsdd/techjournal/0303_yuen/yuen.html.

After assembling your applications, use a systems management tool to deploy the EAR or WAR files onto the 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 the Assembly Toolkit. The tool locates the required external resources that the application uses, such as enterprise beans and databases.

Select a tool to use:
- Administrative console installation pages (GUI)
- Java administrative programs (programming)
- wsadmin AdminApp `install` command (scripting)

If you are uncertain of which systems management tool to use, try using the administrative console.

If your application has a large number of modules, it might not install successfully onto a server. 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 the Assembly Toolkit.

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 the Assembly Toolkit, the tool generates the deployment descriptor for you. While the Assembly Toolkit 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.

Archive support in Version 5.0

The following archives and Web components are supported in Version 5.0:
- Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 and 1.3 enterprise archive (EAR) files
- Enterprise bean (EJB) 2.0 JAR files
- Servlet 2.3 Web archive (WAR) files
- Application client 1.2 and 1.3 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

Starting the Assembly Toolkit

The Assembly Toolkit provides a graphical interface for packaging code artifacts into various archives (modules) and configuring related Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 or 1.3 compliant deployment descriptors.

The Assembly Toolkit is a component of the Application Server Toolkit (ASTK). To install the Assembly Toolkit, follow the installation instructions for the ASTK and, when prompted by the ASTK installation program, select to install the application assembly toolkit.

If you have installed the Assembly Toolkit component of ASTK previously and you install the Assembly Toolkit again, you must delete the workspace of the previous installation of ASTK before starting the Assembly Toolkit. The default workspace directory is `*my_directory*\IBM\astk\workspace`. If you do not delete the workspace for a previous installation of ASTK, you might encounter error messages such as the following when starting the Assembly Toolkit:

```
Problems during startup. Check the ".log" file in the ".metadata"
directory of your workspace.
```

1. Run the `astk` executable.
2. In the Application Server Toolkit window, 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.

Consider whether you have any existing J2EE 1.2 application modules that you would like to migrate to J2EE 1.3.

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.

astk command

The **astk** command starts the Application Server Toolkit. The command file is the astk executable file.

Location of the command file

The astk executable file resides in the main installation directory for Application Server Toolkit (ASTK).

Command syntax

- Issue the **astk** command:

  `ASTK_install_root/astk`

  Or, double-click the **ASTK** icon.
- The **astk** command has no options or command-line parameters.

Usage notes

- Is the astk executable file read-only?

  Yes
- Is this file updated by a product component?

  No
- How and when are the contents of this file used?

  The `astk` executable file provides the Assembly Toolkit component for the Application Server Toolkit. Run the `astk` executable file to start the Assembly Toolkit. There are no parameters or command-line options.

Migrating code artifacts to the Assembly Toolkit

You can migrate enterprise archive (EAR), Web archive (WAR), enterprise bean JAR, application client JAR, resource adapter archive (RAR) files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import the files to the Assembly Toolkit.

1. Start the Assembly Toolkit.

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*.
7. Verify the archive files.
8. Generate code for deployment.

Importing enterprise applications

You can import an enterprise archive (EAR) file and define a new enterprise application project using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **Enterprise Applications** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the EAR file and drop it on a view.
3. In the Import dialog, specify the EAR file to import and the project name:
   a. Click **EAR file > Next**.
   b. Specify the EAR file to import. Use **Browse** to locate the EAR file and specify its full path name.
   c. 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.
   d. Click **Finish**.
4. Verify the contents of the new enterprise application project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and view the new project.
   - Click **Window > Show View > Navigator** to see the associated files for the enterprise application project in a Navigator view.

Importing WAR files

Your can import a Web application archive (WAR) file and define a new enterprise archive (EAR) project and Web module for the WAR file using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **Web Modules** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the WAR file and drop it on a view.
3. In the Import dialog, specify a WAR file and a Web project name:
   a. Click **WAR file > Next**.
   b. Specify a WAR file. Use **Browse** to locate the WAR file and specify its full path name.
   c. Specify a Web project. 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.
   d. Optional: To add the WAR file and Web project to an enterprise application, enable **Configure advanced options** and click **Next**. On the J2EE Settings page, specify the EAR project, the context root (Web project), and the J2EE 1.2 or 1.3 specification to use for the Web module. The J2EE 1.3 specification (the default) includes the Servlet 2.3 specification and the JSP 1.2 specification; applications developed for the J2EE 1.3 specification typically target a WebSphere Application Server Version 5.*x* server. Then, click **Next**. On the Features page, specify a feature for the Web project. For example, enable **Default synchronization policy for CVS repository** to have a `.cvsignore` file generated for the `WEB-INF/classes` directory. Then, click **Next** or **Finish**.
   e. Click **Finish**.

4. Verify the contents of the new Web module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Web Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the Web module in a Navigator view.

Importing client applications

You can import an application client JAR file into a new or existing enterprise application using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **Application Client Modules** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the application client JAR file and drop it on a view.
3. In the Import dialog, specify the application client file and project name:
   a. Click **App Client JAR file > Next**.
   b. Specify the application client JAR file to be imported. Use **Browse** to locate the JAR file and specify its full path name.
   c. 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.
   d. Specify the enterprise archive (EAR) file into which to import the application client project.
   e. Click **Finish**.
4. Verify the contents of the new application client module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Application Client Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the application client module in a Navigator view.

Importing EJB files

You can import an enterprise bean (EJB) JAR file and define a new enterprise archive (EAR) project and EJB module for the enterprise bean JAR file using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **EJB Modules** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the enterprise bean JAR file and drop it on a view.
3. In the Import dialog, specify the EJB JAR file and project name:
   a. Click **EJB JAR file > Next**.
   b. Specify the enterprise bean JAR file to import. Use **Browse** to locate the JAR file and specify its full path name.
   c. 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 or 2.0 specification (EJB 2.0 is the default), and click **Next**.
   d. Name the enterprise archive (EAR) file into which to import the enterprise bean JAR file. The name must be unique among EAR files in the directory.
   e. Click **Finish**.
4. Verify the contents of the new EAR file and EJB module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Enterprise Applications** or **EJB Modules** and view the new modules.

- Click **Window > Show View > Navigator** to see the associated files for the EAR file and EJB module in a Navigator view.

Importing RAR files or connectors

You can import a resource adapter archive (RAR) file, or connector, and define a new enterprise archive (EAR) project and connector module using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Right-click **Connector Modules** in a view such as the J2EE Hierarchy view and click **Import > Import Connector Module**.
3. In the Import dialog, specify the connector file and project name:
   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. If you want the connector project not to be part of an enterprise application, specify **Standalone connector project**.
   d. If the connector project is not to stand alone, name the enterprise archive (EAR) file into which to import the RAR file. The name must be unique among EAR files in the directory.
   e. If you want the Assembly Toolkit to overwrite existing resource files without first warning you that the files are changing, specify **Overwrite existing resources without warning**. The default is not to overwrite files without warning.
   f. Click **Finish**.
4. Verify the contents of the new connector module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Enterprise Applications** or **Connector Modules** and view the new modules.
   - Click **Window > Show View > Navigator** to see the associated files for the connector module in a Navigator view.

Creating enterprise applications

Before you can deploy your archive files onto an application server, you must assemble them in an enterprise application archive (EAR) file. This article describes how to create a Java 2 Platform, Enterprise Edition (J2EE) enterprise application project using the Assembly Toolkit. After you create an enterprise application project, you can add (import) 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 the Assembly Toolkit.
2. Click **File > New > Project**. Or, if you are working in the J2EE perspective, click **File > New > Enterprise Application Project** and skip step 3a below.
3. In the New Project dialog, create an enterprise application project:
   a. Click **J2EE > Enterprise Application Project > Next**.
   b. Specify whether you want an EAR file that supports J2EE 1.2 or 1.3, and click **Next**.
   c. On the Enterprise Application Project page, specify an EAR file name and location. To change the default project location, click **Browse** and specify a new location. Then, click **Next**.
   d. Optional: 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, click **New Module**. 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. Then, click **Finish** to create the project modules and add their names to the list of available modules on the EAR Module Projects page.

   e. Click **Finish**.

   f. Optional: Confirm that you want to view the J2EE Hierarchy view.

4. Verify the contents of the new enterprise application in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Application** and view the new EAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the enterprise application in a Navigator view.

Creating Web applications

In the Assembly Toolkit, 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, JavaServer Pages (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 imbedded in enterprise application projects. Creating a Web project in the Assembly Toolkit requires that an enterprise application (EAR) project exist, or the Assembly Toolkit 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 new, 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 the Assembly Toolkit. For instructions on how to create a static Web project, see the Assembly Toolkit online help. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > Web development > Tasks > Working with Web projects > Creating new static Web projects**.

1. Start the Assembly Toolkit.
2. Optional: View an animation file that shows how to create a dynamic Web project using the Assembly Toolkit. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > Web development > Tasks > Working with Web projects > Creating new dynamic Web projects > Show Me**.
3. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
4. Click **File > New > Dynamic Web Project**.
5. On the Dynamic Web Project page of the New Web Project dialog:
   a. Specify a Web project (WAR file) name.

   b. Specify a location for the WAR file. To change the default WAR files location, click **Browse** and specify a new location.

   c. Decide whether you want to accept the defaults associated with a dynamic Web project or configure advanced options. If you want to accept the defaults, deselect **Configure advanced options**. Otherwise, select **Configure advanced options** and **Next**. Step 6 describes the defaults and advanced options for a dynamic Web project.
6. If you selected **Configure advanced options**, you can customize the Web project options:
   a. Specify a new or existing enterprise application (EAR) project to be associated with your new Web project for purposes of deployment. 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.

   b. 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. You can change the context root after you create a project using the project Properties dialog, which you access from the project's context menu.

   c. From the J2EE Level drop-down list, 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.
   d. Click **Next**.
   e. Optional: On the Features Page, select one or more of the Web project features and click **Next**.
   f. Optional: Select **Use a default Page Template for the Web Site** if you want your entire Web site to share a common page template. If you want to use one of the sample templates provided, select **Sample Template** and then choose one of the templates shown in the **Thumbnail** box. If you want to use a template of your own, select **User-defined Template** and then click **Browse** to select the template from the file system. The default is not to use a page template.
7. 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.
8.
9. Verify the contents of the new Web project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your Web project to view the new WAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the Web project in a Navigator view.

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.

Creating application clients

Application client projects contain programs that run on networked client systems. An application client project is deployed as a JAR file.

In the Assembly Toolkit, you can create and add an application client project to a new or existing enterprise application project.

1. Start the Assembly Toolkit.
2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Click **File > New > Application Client Project**.
4. In the Application Client project creation dialog:
   a. Select the Java 2 Platform, Enterprise Edition (J2EE) specification version to which you want your project to adhere, and click **Next**.
   b. 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, the project creation will fail.
   c. 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 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 enterprise application, you must click **New** and create a new enterprise application.

d. 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.

e. Click **Finish** to create the application client project.

5. Verify the contents of the new application client project in either of the following ways:
   - In the**J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your application client project to view the new JAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the application client project in a Navigator view.

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 detailed instructions on adding enterprise bean, resource, or resource environment references, see the Assembly Toolkit online help. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > J2EE application development > Tasks > Configuring application client modules with the client deployment descriptor editor**. Similar information is in the IBM WebSphere Studio Application Developer InfoCenter at `http://publib.boulder.ibm.com/infocenter/wsphelp/index.jsp`. Click **WebSphere Studio Application Developer > J2EE development > Tasks > Configuring application client modules with the client deployment descriptor editor**.

Creating EJB modules

In the Assembly Toolkit, 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 the Assembly Toolkit, you can set deployment descriptor and assembly properties for enterprise beans.

This article describes how to create an EJB project (or EJB module) using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Click **File > New > EJB Project**.
4. In the New EJB Project dialog:
   a. Select the EJB specification version to which you want your EJB project to adhere, and click **Next**. If you plan on using EJB 2.0 enterprise beans, you must specify an EJB 2.0 project. You can add EJB 1.1 enterprise beans to an EJB 2.0 project. An EJB 2.0 project must exist in a J2EE 1.3 enterprise application project. Your available options can differ, depending on the J2EE preferences defined.
   b. 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, the project creation will fail.
   c. 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 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 enterprise application, you must click **New** and create a new enterprise application.

   d. 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. Note that this page is available only if you are using an existing enterprise application project.
   e. Click **Finish** to create the EJB project.
5. Verify the contents of the new EJB project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your EJB project to view the new JAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the EJB project in a Navigator view.

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, see the Assembly Toolkit online help. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > Enterprise JavaBeans (EJB) development > Tasks**. Similar information is in the IBM WebSphere Studio Application Developer InfoCenter at `http://publib.boulder.ibm.com/infocenter/wsphelp/index.jsp`. Click **WebSphere Studio Application Developer > J2EE development > Tasks > Developing EJB applications**.

Creating connector modules

A *connector* is a J2EE component that provides access to Enterprise Information Systems (EIS), and must comply to 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 project using the Assembly Toolkit.
1. Start the Assembly Toolkit.
2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Click **File > New > Connector Project**.
4. In the New Connector Project dialog:
   a. If you want your connector project to be a stand-alone project, select **Standalone connector project**. Selecting that the connector project stand alone disables the enterprise archive (EAR) project option at the bottom of the page.
   b. Name the connector project and specify its location. To change the default project location, click **Browse** and specify a new location.

c. If you specified that the connector project stand alone, specify a new or existing enterprise application (EAR) project to be associated with your new 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. 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 enterprise application, you must click **New** and create a new enterprise application.

d. 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.

e. Click **Finish** to create the connector project.

5. Verify the contents of the new connector project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your connector project to view the new JAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the connector project in a Navigator view.

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. When you create a module, the Assembly Toolkit 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 Toolkit 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	• `application.xml` • `ibm-application-bnd.xmi` • `ibm-application-ext.xmi`
Web deployment descriptor editor	• `WEB-INF/web.xml` • Binding information • IBM binding and extensions information such as `ibm-web-bnd.xmi` and `ibm-web-ext.xmi` files
Enterprise bean (EJB) deployment descriptor editor	• `ejb-jar.xml` • `ibm-ejb-jar-bnd.xml` • `ibm-ejb-jar-ext.xml` • `ibm-ejb-access-bean.xml`
Client deployment descriptor editor	• `application-client.xml` • `ibm-application-client-bnd.xmi` • `ibm-application-client-ext.xmi`
Web services editor	• `webservices.xml` • `ibm-webservices-bnd.xmi` • `ibm-webservices-ext.xmi`
Web services client editor	• `webservicesclient.xml` • `ibm-webservicesclient-bnd.xmi` • `ibm-webservicesclient-ext.xmi`

1. Ensure that you are working in the J2EE Perspective. Click **Window > Open Perspective > J2EE**.

2. In a J2EE Hierarchy view (**Window > Show View > J2EE Hierarchy**), 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 displays 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.
3. Edit the deployment descriptor values as desired. For information on fields in the deployment descriptor editor, press F1 and click a topic.
4. 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.

Mapping enterprise beans to database tables

You can map enterprise bean JAR files (EJB modules) to relational database (RDB) tables using the EJB to RDB Mapping wizard of the Assembly Toolkit. 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.

1. In the J2EE Hierarchy view, right-click the EJB module.
2. Click **Generate > EJB to RDB Mapping**.
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.

Verifying archive files

The Assembly Toolkit 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 are compliant with the EJB 2.0 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 J2EE Hierarchy 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 J2EE Hierarchy 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 application module contains EJB modules, generate code for deployment. Otherwise, you are ready to deploy the application module (or stand-alone Web module) onto the application server.

Generating code for EJB deployment

This task assumes you have already assembled an enterprise bean (EJB) module, added it to an application, saved the application, and verified the application.

Before installing your application in WebSphere Application Server, you must generate deployment code for the application. This step is required for EJB modules and for any enterprise application archive (EAR) files that contain EJB modules. During code generation, the Assembly Toolkit prepares entity bean (JAR) files for deployment in a run-time environment. If your EJB project contains container-managed persistence (CMP) beans that have not been mapped, generating deployment code creates a default top-down mapping.

1. If you have turned automatic validation off, manually validate your enterprise beans before generating deployment code for them. If validating your beans results in compilation errors or validation errors, fix the errors before generating deployment code. However, if validating your beans results in warning or information messages, you can generate deployment code.
2. If you have changed the class path of your EJB project, ensure that the source folder for your EJB project is at the beginning of the class path of the project. Generating deployment code imports both the JAR file and the source code of the JAR file, so entries on the class path must be in the correct order.

3. In the J2EE Hierarchy view of the Assembly Toolkit, right-click on the enterprise application (EAR file) or EJB module (enterprise bean JAR file) for which you want to generate code for deployment.
4. Click a **Generate Deployment Code** option.
   - For EAR files, click **Generate Deployment Code**.
   - For enterprise bean JAR files, click **Generate > Deployment and RMIC Code >** *EJB_module* **> Finish**.

   Alternatively, you can generate deployment code for enterprise bean JAR files using the deployment tool for Enterprise JavaBeans (ejbdeploy) from a command prompt. For a detailed list of available options in the EJBDeploy tool, enter `ejbdeploy` from a command prompt.

Code is generated into the folder where your enterprise beans are located. Problems with the generation of Java RMI stub compiler (RMIC) 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.

Generating code for Web service deployment

This task assumes you have already assembled a module enabled with Web services, added it to an application, saved the application, and verified the application.

Before installing your application in WebSphere Application Server, you must generate deployment code for the application. This step is required for Web services-enabled modules and for any enterprise application archive (EAR) files that contain Web services-enabled modules.

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 J2EE Hierarchy view of the Assembly Toolkit, 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 **Web Services > Deploy Web Service**. 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.

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.

Assembly Toolkit: Resources for learning

Use the following links to find relevant supplemental information about the Assembly Toolkit. 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 instructions and examples
- Programming specifications
- Administration

Programming instructions and examples
- 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
- WebSphere Studio Version 5 Tips and Techniques
- Java 2 Enterprise Edition: Books index

Programming specifications
- J2EE 1.3 specification
- EJB specifications
- Servlet specifications

Administration
- Application Client files
- Connector RAR files

Chapter 17. Deploying and managing applications

After you develop an enterprise application and configure an application server, you can use the administrative console to install application files on the server and manage the activity of deployed applications.

1. Install your application on your application server.
2. Start and stop applications.
3. Edit the administrative configuration for an application. Go to the settings page for an application, change the values for settings as needed, and click **OK**.
4. Export applications.
5. Export DDL files.
6. Update application binary files.
7. Uninstall applications.

After making changes to administrative configurations of your applications, ensure that you click **Save** on the administrative console taskbar to save the changes.

Enterprise 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.

Installing a new application

To install an enterprise application to a WebSphere Application Server configuration, you can use the administrative console or the wsadmin tool. The steps below describe how to use the administrative console to install an application, EJB component, or Web module.

Note: Once 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 install pages is shown.
2. On the first Preparing for application install page:
   a. Specify the full path name of the source 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 stand-alone WAR or JAR file for installation.
   b. If you are installing a stand-alone WAR file, specify the context root.
   c. Click **Next**.

© Copyright IBM Corp. 2003

3. On the second Preparing for application install 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 JNDI prefix for all the 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 657″ describes available customizations and provides sample bindings.
   b. Click **Next**. The Install New Application pages are now shown. If you chose to generate default bindings, you can proceed to the Summary step (step 23 below). ″“Example: Installing an EAR file using the default bindings” on page 660″ provides sample steps.
4. On the **Step: Provide options to perform the installation** 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 JSP files as a part of installation. The default is not to precompile JSP files.
   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\installedApps\cell_name.

      Note: If an installation directory is not specified when an application is installed on a single-server (base) configuration, the application is installed in APP_INSTALL_ROOT/*base_cell_name*. When the base server is made a part of a Network Deployment configuration (using the addNode utility), the cell name of the new configuration becomes the cell name of the deployment manager node. If the `-includeapps` option is used for the addNode utility, then the applications that are installed prior to the addNode operation still use the installation directory APP_INSTALL_ROOT/*base_cell_name*. However, an application that is installed after the base server is added to the network configuration uses the default installation directory APP_INSTALL_ROOT/*network_cell_name*. To move the application to the APP_INSTALL_ROOT/*network_cell_name* location upon running the addNode operation, you should explicitly specify the installation directory as `${APP_INSTALL_ROOT}/${CELL}` during installation. In such a case, the application files can always be found under APP_INSTALL_ROOT/*current_cell_name*.
   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.
   e. For **Deploy EJBs**, specify whether the EJBDeploy tool runs during application installation. The tool generates code needed to run EJB files. The default is not to run the EJBDeploy tool. You must enable this setting if the EAR file was assembled using theAssembly Toolkit and the EJBDeploy tool was not run during assembly, if the EAR file was not assembled using theAssembly Toolkit tool, or if the EAR file was assembled using versions of the Application Assembly Tool (AAT) previous to Version 5. Note that enabling this setting might cause the installation program to run for several minutes.
   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.

i. For **Reload Interval**, 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. This setting takes effect only if class reloading is enabled.

   The reload interval specified here overrides the value specified in the IBM extensions for each 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).

j. For **Deploy WebServices**, 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 **Web Services > Deploy Web Service** menu of the Assembly Toolkit or from a command line.

5. 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`.
6. 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.
7. 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** (step 23) and the installation is cancelled.
8. 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.
9. 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. 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.
10. 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.
11. On the **Step: Map modules to application servers** panel, for every module select a target server or a cluster from the **Clusters and Servers** list. Select the check box beside **Module** to select all of the application modules or select individual modules.
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 the check box beside **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 shown, 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 the check box beside **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 for messaging beans**, provide a listener port name for every message driven bean. If a name is not specified for each bean, then a validation error displays after you click on **Finish** on the Summary panel.
16. If your application uses EJB modules that contain CMP beans that are based on the EJB 2.0 specification, for **Step: Provide default datasource mapping for modules containing 2.0 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. 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 2.0 specification, on the **Step: Map datasources for all 2.0 CMP** panel, for each of the 2.0 CMP beans specify a JNDI name and the type of resource authorization for data sources to be used. 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 shown after you click **Finish** and installation is cancelled.
18. If your application contains EJB 2.0 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 2.0 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 EJBs** setting is enabled on the **Provide options to perform the installation** 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 path, 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: Mapping 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, a validation error is shown after you click **Finish**.
21. If your application defines **Run-As Identity** as *System Identity*, for **Step: Replacing 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. On the Summary panel, verify the cell, node, and server onto which the application modules will install. Beside the **Cell/Node/Server** option, click **Click here** and verify the settings. Then click **Finish**.

    Note: After clicking **Finish**, 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.
24. Associate any shared libraries that the application needs to the application.
25. Click **Save** on the administrative console taskbar to 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 the single-server (base) installation, application files are copied to the destination directory when you click **Save**; for the Network Deployment installation, files are copied to remote nodes when the configuration on the deployment manager synchronizes with the configuration on individual nodes.
26. Start the application.
27. 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 path** if the browser and application files are on the same machine (whether or not the server is on that machine, too).

Use **Server path** if the application file resides on any node in the current cell context.You can browse the entire file system of a node if the node agent or deployment manager is running on that selected node. Only .ear, .jar, or .war files are shown during the browsing.

During application installation, application files are typically uploaded from a client machine running the browser to the server machine running the administrative console, where they are deployed. In such cases, the Web browser running the administrative console is used 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 **Server path** option.

You can also use this option to specify an application file already residing on the machine running the application server. For example, the field value on Windows NT might be `C:\WebSphere\AppServer\installableApps\test.ear`. If you are installing a stand-alone WAR module, then you also must specify the context root.

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.
- Message-driven bean (MDB) 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 CMPs. 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 alter 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 to override existing bindings.
If this check box is checked, the existing bindings are overridden by the generated ones.

EJB 1.1 CMP bindings

Specifies the default data source JNDI name.

If the **Default Bindings for EJB 1.1 CMPs** radio button is selected, specify the JNDI name for the default data source to be used with the CMP 1.1 beans. Also specify the user ID and password for this default data source.

Connection Factory Bindings

Specifies the default data source JNDI name.

If the **Default connection factory bindings** radio button 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 WAR modules.

Specific bindings file

Specifies a bindings file that overrides the default binding.

Alter the behavior of the default binding with an XML document (*aka* custom strategy). Custom strategies extend the default strategy so you only need to customize those areas where the default strategy is insufficient. That is, 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>
<!-- this name must match the module name in the .ear file -->
      <ejb-bindings>
        <ejb-binding>
         <ejb-name>HelloEjb</ejb-name>
<!-- this must match the <ejb-name> entry in the EJB jar DD -->
         <jndi-name>com/acme/ejb/HelloHome</jndi-name>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

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>
<!-- this matches the ejb-name tag in the DD -->
          <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>
```

Overriding a Resource Ref Binding from a WAR, EJB JAR file, or J2EE client JAR file

Example code for overriding a Resource Ref Binding from a WAR file follows. Use similar code to override a Resource Ref Binding from an enterprise bean (EJB) JAR file or a J2EE client JAR file.

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <war-binding>
      <jar-name>hello.war</jar-name>
      <resource-ref-bindings>
        <resource-ref-binding>
          <!-- the following must match the resource-ref in the DD -->
          <resource-ref-name>jdbc/MyDataSrc</resource-ref-name>
          <jndi-name>war/override/dataSource</jndi-name>
        </resource-ref-binding>
      </resource-ref-bindings>
    </war-binding>
  </module-bindings>
</dfltbndngs>
```

Overriding MDB JMS listener ports

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>YourEjbJar.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourMDB</ejb-name>
          <listener-port>yourMdbListPort</listener-port>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Example: Installing an EAR file using the default bindings

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 an Application** in the console navigation tree.
2. For **Path**, specify the full path name of the .ear file. For this example, the base file name is my_appl.ear and the file resides on a server at C:\sample_apps.
   a. Select the **Server path** radio button and click **Browse**.

   b. On the Browse Remote Filesystems page, click on the name of the node that holds the my_appl.ear file, **C:\**, **sample_apps**, **my_appl.ear**, and then **OK**.
3. Now that a value is given for **Path**, on the first Preparing for application install page, click **Next**.
4. On the second Preparing for application install page, place a checkmark beside the **Generate Default Bindings** check box 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 directly jump to the Summary step.
5. On the Install New Application page, click on **Summary**, the last step.
6. 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 application servers** panel, select the server onto which the application files will install from the **Clusters and Servers** list, place a checkmark in the check box beside **Module** to select all of the application modules, and click **Next**.

   Because my_appl.ear does not require any additional settings to complete an installation, the Summary panel displays again.
7. On the Summary panel, click **Finish**.

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*).
1. Go to the Enterprise Applications page. Click **Applications > Enterprise Applications** in the console navigation tree.
2. Check 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 from *Stopped* to *Started*.
Stop	Stops the processing of the application and changes the state of the application from *Started* to *Stopped*.

   To restart a running application, place a check mark in the check box for 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.

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 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**.
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 and click **OK**.

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 the DDL files (Table.ddl) in the EJB modules of the application 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 checkmark 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* displays at the top of the page. If the application has DDL files in its EJB modules, then a page displays listing DDL files in the format appname.ear/_module.jar_Table.ddl.
3. Click on a file in the list to download the file to your machine.

Updating applications

You can update an application deployed on a server. The steps below describe how to update a deployed application using the administrative console.

Note: You can also update applications using the wsadmin tool, which provides updating capabilities identical to those available using the administrative console. Further, in some situations, you can update applications without needing to restart the application server.

1. Update the contents of the application and reassemble it, using the Assembly Toolkit. Typical tasks include adding or editing assembly properties, adding or importing modules into an application, and adding enterprise beans, Web components, and files.
2. Go to the Applications page of the administrative console. Click **Applications > Enterprise Applications** in the console navigation tree.
3. Back up the application. Place a checkmark in the check box beside the application you want uninstalled and click **Export** to export the application to an EAR file and preserve the binding information.
4. With a checkmark beside the application, click **Update**. The binding information of the updated (new) version of the application merges with the binding information from the installed (old) version. Then, the older version uninstalls from the configuration and the new version installs.
5. Complete the steps in the Preparing for application update page and the pages that follow it. See 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. Map the installed application or module to servers or clusters. Use the Map modules to application servers page of the Install New Application pages displayed during updating the application. Or, after updating the application, use the Map modules to application servers page accessed from the Enterprise Applications page.
   a. Go to the Map modules to application servers page. Click **Applications > Enterprise Applications** in the console navigation tree, click the application name, and then click **Map modules to application servers**.
   b. Specify the application server where you want to install modules contained in your application and click **OK**.
7. Click **Save** on the administrative console taskbar to save the changes to your configuration. In the single server (base) product, after you click **Save** 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. In the Network

Deployment product, the old application files are deleted and new files are copied when the configuration on the deployment manager synchronizes with the configuration on the node where the application is installed. If the application is running when you update it, the application stops running before its files are copied to the destination directory of the node and restarts after the copy operation completes. Thus, the application is unavailable on the node during the time the node is synchronizing its configuration with the deployment manager.

8. Restart the application so the changes take effect. If the application is updated while it is running, WebSphere Application Server stops the application, updates the application logic and restarts the application.
   a. Click **Applications > Enterprise Applications** in the console navigation tree to go to the Enterprise Applications page.
   b. Check the check box for the updated application.
   c. Click **Start**.
9. Optional: If the application you are updating is deployed on a server that has its application class loader policy set to SINGLE, restart the server.

Hot deployment and dynamic reloading

You can make various changes to applications and their contents without having to stop the server and start it again. Making these types of changes is known as *hot deployment and dynamic reloading*.

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

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. CAUTION: If you are running WebSphere Application Server on a group of machines using Network Deployment and you are changing an application on a particular node, disable automatic synchronization.
   a. Click **System Administration > Node Agents** in the administrative console navigation tree, click on a node agent name, and then click **File Synchronization Service**.
   b. On the File Synchronization Service page, remove the checkmark from the check box for **Automatic Synchronization** and click **OK**.

   When you run WebSphere Application Server on a group of machines using Network Deployment and you change a file on the disk in the expanded application directory for a particular node, you can lose those changes the next time node synchronization occurs. In the Network Deployment environment, the configuration stored by the deployment manager is the master copy and any changes detected between that master copy and the copy on a particular machine trigger the master copy to be downloaded to the node.
4. 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 661″ provides information on using the administrative console to start, stop, or restart an application. ″Example: Starting an application using wsadmin″ and ″Example: Stopping running applications on a server using wsadmin″ provide information on using the wsadmin scripting tool.
6. If you disabled automatic synchronization in step 3, return to the File Synchronization Service page, enable **Automatic Synchronization**, and click **OK**.

Changing or adding application files

You can change or add application files on application servers without having to stop the server and start it again. This file describes--
- Updating an existing application on a running server (providing a new EAR file)
- Adding a new application to a running server
- Removing an existing application from a running server
- Adding a new EJB or Web module to an existing, running application
- 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

- Update 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

- Add 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

- Remove an existing application from a running server. Stop the application and then uninstall it from the server. Use the administrative console to stop the application and then uninstall it. Or run the wasadmin `stopApplication` command and then the `uninstall` command.

Hot deployment:	Yes
Dynamic reloading:	No

- Add a new EJB or Web module to an existing, running application.
  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

- Change 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

- Change 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

- Change 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

- Change 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 WAR files on application servers without having to stop the server and start it again. This file describes--

- Changing an existing 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

- Change 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

- Add 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

- Change 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

- Change 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

- Add 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

- Add a new servlet, including a new definition of the servlet in the web.xml deployment descriptor for the application. 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.
- 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

- Change 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

- Change 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 will no longer automatically detect 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

- Change 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 EJB Jar files on application servers without having to stop the server and start it again. This file describes--

- Changing the ejb-jar.xml file of an EJB Jar file
- Changing the ibm-ejb-jar-ext.xmi or ibm-ejb-jar-bnd.xmi file of an EJB Jar file
- Changing the Table.ddl file for an EJB Jar file
- Changing the Map.mapxmi or Schema.dbxmi file for an EJB Jar file
- Updating the implementation class for an EJB file or a dependent class of the implementation class for an EJB file
- Updating the Home/Remote interface class for an EJB file
- Adding a new EJB file to an existing EJB Jar file

- Change the ejb-jar.xml file of an EJB Jar file. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment:	Not applicable
Dynamic reloading:	Yes

- Change the ibm-ejb-jar-ext.xmi or ibm-ejb-jar-bnd.xmi file of an EJB Jar file. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment:	Not applicable
Dynamic reloading:	Yes

- Change 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

- Change 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

- Update 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

- Update 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

- Add 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. This file describes--
- Changing the application.xml file to change the context root of a 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. Regenerate the plug-in configuration file using the administrative console or by running the GenPluginCfg.bat/sh script.

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. Regenerate the plug-in configuration file using the administrative console or by running the GenPluginCfg.bat/sh script.

If the Web application has file serving enabled or has a servlet mapping of /, you do not have to regenerate the plug-in configuration. In all other cases the 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. Regenerate the plug-in configuration file using the administrative console, by running the GenPluginCfg.bat/sh script, or by running a wsadmin command.

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. Stop the application. Select the application you want uninstalled and click **Stop**.
3. Back up the application. Select the application you want uninstalled and click **Export** to export the application to an EAR file and preserve the binding information.
4. Select the application you want uninstalled and click **Uninstall**.
5. Click **Save** on the console taskbar to save changes made to the administrative configuration.

In the single-server (base) product, application binaries are deleted after you click **Save.**In the Network Deployment product, application binaries are deleted when configuration changes on the deployment manager synchronize with configurations for individual nodes.

Deploying and managing applications: Resources for learning

Use the following links to find relevant supplemental information about deploying and managing 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.

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 Bank 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^{TM} 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^{TM} 2 Platform, Enterprise Edition

Administration
- Listing of all IBM WebSphere Application Server Redbooks

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

© Copyright IBM Corp. 2003

Trademarks and service marks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- AIX
- CICS
- Cloudscape
- DB2
- DFSMS
- Everyplace
- iSeries
- IBM
- IMS
- Informix
- iSeries
- Language Environment
- MQSeries
- MVS
- OS/390
- RACF
- Redbooks
- RMF
- SecureWay
- SupportPac
- ViaVoice
- VisualAge
- VTAM
- WebSphere
- z/OS
- zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.